



Artix™

Getting Started with Artix

Version 3.0, October 2005

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: 28-Oct-2005

Contents

Preface	v
What is Covered in this Book	v
Who Should Read this Book	v
Organization of this Book	v
Finding Your Way Around the Library	vi
Searching the Artix Library	vii
Online Help	vii
Additional Resources	viii
Document Conventions	viii
List of Figures	xi
Chapter 1 Introduction	1
What is Artix?	2
Solving Problems with Artix	7
Using the Artix Documentation	9
Chapter 2 Artix Concepts	13
The Artix Runtime Components	14
The Artix Bus	15
Artix Endpoints	16
Artix Contracts	17
Artix Services	19
Chapter 3 Understanding WSDL	21
WSDL Basics	22
Abstract Data Type Definitions	25
Abstract Message Definitions	28
Abstract Interface Definitions	31
Mapping to the Concrete Details	34

Chapter 4 Using Artix Designer	35
Introduction	36
Creating Artix Designer Projects	41
Creating a WSDL File	43
Defining the WSDL Elements	45
Defining Types	46
Defining Messages	50
Defining Port Types	52
Defining Bindings	55
Defining a Service	58
Developing the Applications	62
Creating code generation configurations	63
Adding Logic to the Code	68
The C++ Code	69
The Java Code	72
Running the Applications	74
Glossary	77
Index	85

Preface

What is Covered in this Book

Getting Started with Artix provides an introduction to IONA's Artix technology. It gives a brief overview of the architecture and functionality of Artix, and an introduction to Web Services Description Language (WSDL).

This book takes you through the process of creating a WSDL file and generating starting point code in both C++ and Java using the Artix Designer development tool.

This book also provides guidance for finding your way around the Artix product library.

Who Should Read this Book

Getting Started with Artix is for anyone who needs to understand the concepts and terms used in IONA's Artix product.

Organization of this Book

This book contains conceptual information about Artix and WSDL:

- [“Introduction” on page 1](#) introduces the Artix product and the types of problems it is designed to solve, and provides an introduction walkthrough of the Artix documentation library.
- [“Artix Concepts” on page 13](#) explains the main concepts used in Artix.
- [“Understanding WSDL” on page 21](#) explains the basics of WSDL.
- [“Using Artix Designer” on page 35](#) explains the basics of using the Artix GUI to edit Artix contracts and other Artix project artifacts.

There is also a [“Glossary” on page 77](#) of this book to help you with any unfamiliar terms.

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 might 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 and 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
```

<i>Italic</i>	Italic words in normal text represent <i>emphasis</i> and introduce <i>new terms</i> .
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).

PREFACE

List of Figures

Figure 1: Artix High-Performance Architecture	4
Figure 2: Artix Runtime Components	14
Figure 3: The Empty JavaHello Project in the Navigator View	42
Figure 4: The CppHello Project With a Link to the HelloWorld.wsdl File	44
Figure 5: The Select Source Resources Panel	46
Figure 6: The Define Type Properties Panel	47
Figure 7: The Define Element Data Panel	48
Figure 8: The Define Message Parts Panel	50
Figure 9: The Define Port Type Operations Panel	53
Figure 10: The Define Operation Messages Panel	54
Figure 11: The Select Binding Type Panel	55
Figure 12: The Set Binding Defaults Panel	56
Figure 13: The Define Service Panel	58
Figure 14: The Define Port Panel	59
Figure 15: The Define Port Properties panel	59
Figure 16: The Artix Generator Window	63
Figure 17: The General Tab	64
Figure 18: The Generation Tab	65
Figure 19: The WSDL Details Tab	66
Figure 20: The Terminate and Remove All Terminated Launches Buttons	75

LIST OF FIGURES

Introduction

This chapter introduces the main features of Artix, and describes where to look in the documentation for further information.

In this chapter

This chapter discusses the following topics:

What is Artix?	page 2
Solving Problems with Artix	page 7
Using the Artix Documentation	page 9

What is Artix?

Overview

Artix is an extensible enterprise service bus (ESB). It provides the tools for rapid application integration that exploits the middleware technologies and the products already present within your enterprise.

The approach taken by Artix relies heavily on existing Web service standards and extends these standards to provide rapid integration solutions that increase operational efficiencies, capitalize on existing infrastructure, and enable the adoption or extension of a service-oriented architecture (SOA).

Web services and SOAs

The information services community generally regards Web services as application-to-application interactions that use SOAP over HTTP.

Web services have the following advantages:

- The data encoding scheme and transport semantics are based on standardized specifications.
- The XML message content is human readable.
- The contract defining the service is XML-based and can be edited by any text editor.
- They promote loosely coupled architectures.

Service-oriented architectures take the Web services concept and extend it to the entire enterprise. Using a service-oriented architecture, your infrastructure becomes a collection of loosely coupled services. Each service becomes an endpoint defined by a contract written in Web Services Description Language (WSDL). Clients, or service consumers, can then access the services by reading a service's contract.

Artix and services

Using IONA's proven *Adaptive Runtime Technology* (ART), Artix extends the Web service standards to include more than just SOAP over HTTP. Thus, Artix allows organizations to define their existing applications as services without worrying about the underlying middleware. It also provides the ability to expose those applications across a number of middleware technologies without writing any new code.

Artix also provides developers with the tools to write new applications in C++ or Java that can be exposed as middleware-neutral services. These tools aid in the definition of the new service in WSDL and in the generation of stub and skeleton code.

Just like the WSDL contracts used to define a service, the code that Artix generates adheres to industry standards.

Benefits of Artix

Artix's extensible nature provides a number of benefits compared to other ESB products and older enterprise application integration (EAI) products. Chief among these is its speed and flexibility. In addition, Artix provides enterprise levels of service such as session management, service discovery, security, and cross-middleware transaction propagation.

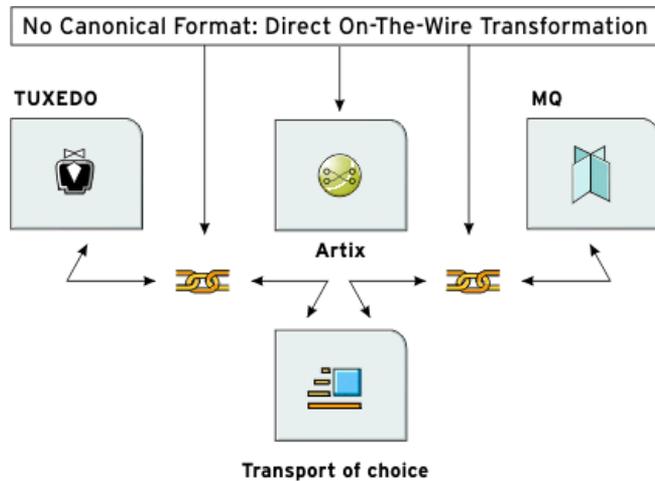
EAI products typically use a proprietary, canonical message format in a centralized EAI hub. When the hub receives a message, it transforms the message to this canonical format and then transforms the message to the format of the target application before sending it to its destination. Each application requires two adapters that are typically proprietary and that translate to and from the canonical format.

By contrast, the Artix bus does not require a hub architecture, nor does it use any intermediate message format. When a message is received by the bus, it is transformed directly into the target application's message format.

Because Artix uses a standardized means of defining its services, the plug-ins used to connect applications to the bus are reusable.

Figure 1 shows an example Artix integration between BEA Tuxedo and IBM WebSphere MQ.

Figure 1: *Artix High-Performance Architecture*



Because Artix is built on top of ART, it is modular in nature. This means that it is highly configurable and that it is easily extendable. You can configure Artix to only load the pieces that you need for the functionality you require. If Artix does not provide a transport or message format that you need, you can easily develop your own plug-in, extend the contract definitions, and configure Artix to load it.

Using Artix

There are two ways to use Artix in your enterprise:

- You can use Artix to develop new applications using the Artix Application Programming Interface (API). In this situation, developers generate Artix stubs and skeletons from an Artix contract and Artix becomes a part of your development environment.
- You can use the Artix bus to integrate two existing applications, built on different middleware technologies, into a single application. In this situation, developers simply create an Artix contract defining the integration of the systems. In most cases, no new code is needed.

Becoming proficient with Artix

To become an effective Artix developer you need an understanding of the following:

1. The syntax for WSDL files and the Artix extensions to the WSDL specification.
2. The relationship between Artix WSDL extensions, ART plug-ins, and setting configuration entries.
3. The Artix APIs that you can use in your application.
4. Artix Designer, a GUI tool that enables you to write, generate, and edit WSDL files, and to generate, compile, and run code.

This book introduces these four concepts. The other books in the Artix documentation library covers the same concepts in greater detail.

Artix features

Artix includes the following unique features:

- Support for multiple transports and message data formats
- C++ and Java development
- Message routing
- Cross-middleware transaction support
- Asynchronous Web services
- Deployment of services as plug-ins via the Artix container
- Role-based security, single sign-on, and security integration
- Session management and stateful Web services
- Look-up services
- Load-balancing
- High-availability service clustering
- Integration with EJBs
- Easy-to-use development tools
- Support for Microsoft .NET
- Integration with enterprise management tools such as IBM Tivoli and BMC Patrol
- Support for XSLT-based message transformation
- No need to hard code WSDL references into applications

Supported transports and protocols

A *transport* is an on-the-wire format for messages; whereas a *protocol* is a transport that is defined by an open specification. For example, MQ and Tuxedo are transports, while HTTP and IIOP are protocols.

In Artix, 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
- CORBA
- Java Messaging Service

Supported payload formats

A *payload format* controls the layout of a message delivered over a transport. Artix can automatically transform between the following payload formats:

- CORBA Common Data Representation (CDR)
- G2++
- Fixed record length (FRL)
- SOAP
- Pure XML
- Tagged (variable record length)
- TibrvMsg (a TIBCO Rendezvous format)
- Tuxedo's Field Manipulation Language (FML)

Further information

For more information about supported transports and payload formats, see [Designing Artix Solutions](#).

For information about Artix mainframe support, see the documentation for Artix Advanced for z/OS, available at:

<http://www.iona.com/support/docs/index.xml>.

Solving Problems with Artix

Overview

Artix enables you to easily solve problems arising from the integration of existing back-end systems using a service-oriented approach. Artix enables you to develop new services using C++ or Java, and to retain all of the enterprise levels of service that you require.

In general, there are three phases to an Artix project:

1. The design phase, where you define your services and define how they are integrated using Artix contracts.
2. The development phase, where you write the application code required to implement new services.
3. The deployment phase, where you configure and deploy your Artix solution.

Design phase

In the design phase, you define the logical layout of your system in an Artix contract. The logical definition of a system includes the services that it contains, the operations each service offers, and the data the services will use to exchange information.

Once you have defined the logical aspects of your system, you then add the physical network details to the contracts.

The physical details of your system include the transports and payload formats used by your services, as well as any routing schemes needed to connect services that use different transports or payload formats.

Artix Designer and the Artix command-line tools automate the mapping of your service descriptions into WSDL-based Artix contracts. These tools enable you to:

- Import existing WSDL documents
- Create Artix contracts from scratch
- Generate Artix contracts from:
 - ◆ CORBA IDL
 - ◆ A description of tagged data
 - ◆ A description of fixed data
 - ◆ A COBOL copybook

- ◆ A Java class
 - Add the following bindings to an Artix contract:
 - ◆ CORBA
 - ◆ Fixed
 - ◆ SOAP
 - ◆ Tagged
 - ◆ XML
-

Development phase

You need to write Artix application code if your solution involves creating new applications or a custom router, or involves using the locator or session management features. The first step in writing Artix code is to generate client stub code and server skeleton code from the Artix contracts that you created in the design phase. You can generate this code using Artix Designer or the Artix command-line tools.

After you have generated the client stub code and server skeleton code, you can develop the code that implements the business logic you require. For most applications, Artix-generated code allows you to stick to using standard C++ or Java code for writing business logic.

Artix Designer is integrated with the Eclipse IDE, but you are not required to use Eclipse for the whole project. Once the stub code is generated, you can switch to your favorite development environment to develop and debug the application code.

Artix also provides advanced APIs for directly manipulating messages, for writing message handlers, and for other advanced features your application might require. These can be plugged into the Artix runtime for customized processing of messages.

Deployment phase

In the deployment phase, you configure the Artix runtime to fine-tune the Artix bus for your new Artix system. This involves modifying the Artix configuration files and editing the Artix contracts that describe your solution to fit the exact circumstances of your deployment environment.

This phase also includes the managing of the deployed system. This might involve, for example, using an enterprise management tool such as Tivoli along with the Artix command interface. These tools allow you to further fine-tune your system.

Using the Artix Documentation

Overview

The Artix library consists of a number of guides to help you use Artix. The guides are organized in groups that reflect the three phases of Artix problem solving. This section gives a brief overview of each guide and suggests an order in which to read the library.

If you are new to Artix

If you are approaching Artix for the first time, work through the library in the following order:

1. Getting Started with Artix (this book)
2. [Artix Technical Use Cases](#)
3. [Designing Artix Solutions](#)
4. [Developing Artix Applications in C++](#), or
[Developing Artix Applications in Java](#)
5. [Deploying and Managing Artix Solutions](#)

In addition, the following publications provide useful background information:

- *Understanding Web Services: XML, WSDL, SOAP, and UDDI*, by Eric Newcomer
 - *Understanding SOA with Web Services*, by Eric Newcomer and Greg Lomow
 - The W3C XML Schema page at www.w3.org/XML/Schema
 - THE W3C WSDL specification at www.w3.org/TR/wsdl
-

Designing Artix contracts

This section of the library has a single guide called [Designing Artix Solutions](#). This guide discusses how to describe your services and their integration using Artix contracts. The guide provides descriptions of the tools provided by the Artix design environment as well as the WSDL used in the contracts.

While there is some discussion of the logical portions of your service in WSDL, this guide does not go into depth about WSDL and the elements used to describe the physical portion of the contract. It does, however, describe in detail all of the Artix extensions used to describe non-SOAP payload formats, transports, and routing rules.

Developing Artix applications

The Artix documentation suite includes three main development guides:

- [Developing Artix Applications in C++](#)
- [Developing Artix Applications in Java](#)
- [Developing Artix Plug-Ins in C++](#)

Both of the *Developing Artix Applications* guides describe how to develop clients and servers using the Artix APIs. They discuss how the Artix code generators map XML Schema types to code and how to work with the generated types. They also provide examples of using Artix advanced functionality such as transactions, locator services, session management, and dynamic configuration.

The Java guide also explains how to develop message handlers, plug-ins, and custom message transports. For C++ developers this material is covered in the *Developing Artix Plug-Ins* guide.

As well as the primary developer guides, the Artix documentation suite includes Java API reference material in JavaDoc format and C++ API reference material in Doxygen format.

Deploying and managing Artix solutions

[Deploying and Managing Artix Solutions](#) explains how to configure and deploy all aspects of an Artix solution. It describes the Artix configuration file, where to locate the contracts that control your Artix services, and how to run Artix applications. It also explains how to configure and deploy the Artix locator and session manager.

The [Artix Configuration Guide](#) explains how to configure your Artix environment and provides reference information on Artix configuration variables.

In addition, the [IONA Tivoli Integration Guide](#) and the [IONA BMC Patrol Integration Guide](#) explain Artix integration with IBM's Tivoli enterprise management system and BMC Patrol.

Lastly, Artix provides the [Artix Security Guide](#) for security configuration and management.

Latest updates

The latest updates to the Artix 3.0 documentation can be found at:
<http://www.iona.com/support/docs/artix/3.0/index.xml>.

Artix Concepts

This chapter introduces the key concepts used in the Artix product.

In this chapter

This chapter discusses the following topics:

The Artix Runtime Components	page 14
The Artix Bus	page 15
Artix Endpoints	page 16
Artix Contracts	page 17
Artix Services	page 19

The Artix Runtime Components

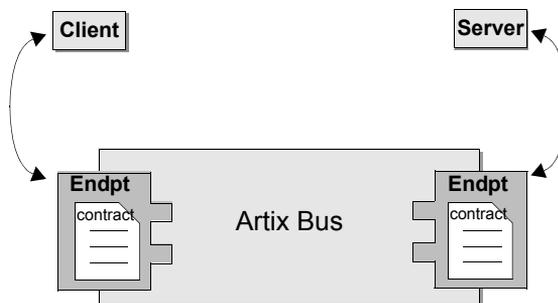
How it fits together

Artix is comprised of a group of components that are built on the Adaptive Runtime Technology (ART) platform:

- [The Artix Bus](#) is at the core of Artix, and provides the transport and payload format support.
- [Artix Contracts](#) describe your applications in such a way that they become services that can be deployed as [Artix Endpoints](#).
- [Artix Services](#) include a number of advanced services, such as the Locator and Session Manager. Each Artix service is defined with an Artix contract and can be deployed as an Artix endpoint.

[Figure 2](#) illustrates how the Artix elements fit together.

Figure 2: *Artix Runtime Components*



Plugability

Because Artix is built on [ART](#), all Artix services are implemented as plug-ins. You can also deploy your own services as plug-ins. This means that you can host any Artix service either as a standalone application or as a plug-in to another Artix application.

Each separate service, regardless of how it is deployed, becomes a separate endpoint.

The Artix Bus

Overview

The Artix bus is at the heart of the Artix architecture. It is the component that hosts the services that you create and connects your applications to those services.

The bus is also responsible for translating data from one format into another. This translation process works as follows:

1. Reader plug-ins accept incoming data in one format.
2. The Artix bus directly translates the data into another format.
3. Writer plug-ins write the data back out to the wire in the new format.

In this way, Artix enables all of the services in your company to communicate, without needing to communicate in the same way. It also means that clients can contact services without understanding the native language of the server handling requests.

Benefits

While other products provide some ability to expose applications as services, they frequently require a good deal of coding. The Artix bus eliminates the need to modify your applications or write code by directly translating the application's native communication protocol into any of the other supported protocols.

For example, by deploying an Artix instance with a SOAP-over-WebSphere MQ endpoint and a SOAP-over-HTTP endpoint, you can expose a WebSphere MQ application directly as a Web service. The WebSphere MQ application does not need to be altered or made aware that it was being exposed using SOAP over HTTP.

The Artix bus translation facility also makes it a powerful integration tool. Unlike traditional [EAI](#) products, Artix translates directly between different middlewares without first translating into a canonical format. This saves processing overhead and increases the speed at which messages are transmitted.

Artix Endpoints

Overview

An Artix endpoint is where a service or a service consumer connects to the Artix bus. Endpoints are described by a contract describing the services offered and the physical representation of the data on the network.

Reconfigurable connection

An Artix endpoint provides an abstract connection point between applications, as shown in [Figure 2 on page 14](#). The benefit of using this abstract connection is that it allows you to change the underlying communication mechanisms without recoding any of your applications. You simply need to modify the contract describing the endpoint.

For example, if one of your back-end service providers is a Tuxedo application and you want to swap it for a CORBA implementation, you simply change the endpoint's contract to contain a CORBA connection to the Artix bus. The clients accessing the back-end service provider do not need to be aware of the change.

Artix Contracts

Overview

Artix contracts are written in WSDL. In this way, a standard language is used to describe the characteristics of services and their associated Artix endpoints. By defining characteristics such as service operations and messages in an abstract way—independent of the transport or protocol used to implement the endpoint—these characteristics can be bound to a variety of protocols and formats.

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 pair of systems with a shared interface, payload format, and transport. Artix contracts, however, can define very complex integration scenarios.

WSDL elements

Understanding Artix contracts requires some familiarity with WSDL. The key WSDL elements are as follows:

WSDL types provide data type definitions used to describe messages.

A WSDL message is an abstract definition of the data being communicated. Each part of a message is associated with a defined type.

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 portType is a set of abstract operation descriptions.

A WSDL binding associates a specific data format for operations defined in a `portType`.

A WSDL port specifies the transport details 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 is conceptually divided into logical and physical components.

The logical contract

The logical contract specifies components 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. It enables Artix to generate skeletons and stubs without having to define the physical characteristics of the connection (transport and wire format).

The logical contract includes the `types`, `message`, `operation`, and `portType` elements of the WSDL file.

The physical contract

The physical component of an Artix contract defines the format and transport-specific details. For example:

- 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

The physical contract includes the `binding`, `port`, and `service` elements of the WSDL file.

Artix Services

Overview

In addition to the core Artix components, Artix also provides the following services:

- [Locator](#)
- [Session manager](#)
- [Transformer](#)
- [Container](#)
- [Bootstrapping](#)

These services provide advanced functionality that Artix deployments can use to gain even more flexibility.

Locator

The Artix locator provides service look-up and load balancing functionality to an Artix deployment. It isolates clients from changes in a server's contact information.

The Artix WSDL contract defines how the client contacts the server, and contains the address of the Artix locator. The locator provides the client with a reference to the server.

Servers are automatically registered with the locator when they start, and service endpoints are automatically made available to clients without the need for additional coding.

Session manager

The Artix session manager is a group of plug-ins that work together to manage the number of concurrent clients that access a group of services. This enables you to control how long each client can use the services in the group before having to check back with the session manager.

In addition, the session manager has a pluggable policy callback mechanism that enables you to implement your own session management policies.

Transformer

The Artix transformer provides Artix with a way to transform operation parameters on the wire using rules written in Extensible Style Sheet Transformation (XSLT) scripts. The transformer can be used to provide a

simple means of transforming data. For example, it can be used to develop an application that accepts names as a single string and returns them as separate first and last name strings.

The transformer can also be placed between two applications where it can transform messages as they pass between the applications. This functionality allows you to connect applications that do not use exactly the same interfaces and still realize the benefits of not using a canonical format.

Container

The Artix container provides a consistent mechanism for deploying and managing Artix services. It allows you to write Web service implementations as Artix plug-ins and then deploy your services into the Artix container.

Using the container eliminates the need to write your own C++ or Java server mainline. Instead, you can deploy your service by simply passing the location of a generated deployment descriptor to the Artix container's administration client.

Bootstrapping

Bootstrapping in Artix refers to enabling client and server applications to find WSDL service contracts and references. Using the Artix bootstrapping service avoids the need to hard code WSDL into your client and server applications.

For more information

For more information on Artix services, see [Deploying and Managing Artix Solutions](#).

Understanding WSDL

Artix contracts use WSDL documents to describe services and the data they use.

In this chapter

This chapter discusses the following topics:

WSDL Basics	page 22
Abstract Data Type Definitions	page 25
Abstract Message Definitions	page 28
Abstract Interface Definitions	page 31
Mapping to the Concrete Details	page 34

WSDL Basics

Overview

Web Services Description Language (WSDL) is an XML document format used to describe services offered over the Web. WSDL is standardized by the World Wide Web Consortium (W3C) and is currently at revision 1.1. You can find the standard on the W3C website at www.w3.org/TR/wsdl.

Elements of a WSDL document

A WSDL document is made up of the following elements:

- `import` allows you to import another WSDL or XSD file.
- Logical contract elements:
 - ◆ `types`
 - ◆ `message`
 - ◆ `operation`
 - ◆ `portType`
- Physical contract elements:
 - ◆ `binding`
 - ◆ `port`
 - ◆ `service`

These elements are described in “WSDL elements” on page 17.

Abstract operations

The abstract definition of *operations* and *messages* is separated from the concrete data formatting definitions and network protocol details. As a result, the abstract definitions can be reused and recombined to define several endpoints. For example, a service can expose identical operations with slightly different concrete data formats and two different network addresses. Alternatively, one WSDL document could be used to define several services that use the same abstract messages.

The portType

A *portType* is a collection of abstract operations that define the actions provided by an endpoint.

Concrete details

When a portType is mapped to a concrete data format, the result is a concrete representation of the abstract definition, in the form of reusable *binding*. A *port* is defined by associating a network address with a reusable binding, in the form of an endpoint. A collection of ports (or endpoints) define a service.

Because WSDL was intended to describe services offered over the Web, the concrete message format is typically SOAP and the network protocol is typically HTTP. However, WSDL documents can use any concrete message format and network protocol. In fact, Artix contracts bind operations to several data formats and describe the details for a number of network protocols.

Namespaces and imported descriptions

WSDL supports the use of XML namespaces defined in the `definition` element as a way of specifying predefined extensions and type systems in a WSDL document. WSDL also supports importing WSDL documents and fragments for building modular WSDL collections.

Example

[Example 1](#) shows a simple WSDL document.

Example 1: Simple WSDL example

```
<definitions name="HelloWorld.wsdl"
  targetNamespace="http://www.iona.com/artix/HelloWorld"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.iona.com/artix/HelloWorld"
  xmlns:wSDL="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <types>
    <schema
      targetNamespace="http://www.iona.com/artix/HelloWorld"
      xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="InElement" type="string"/>
      <element name="OutElement" type="string"/>
    </schema>
  </types>
```

Example 1: *Simple WSDL example (Continued)*

```

<message name="RequestMessage">
  <part element="tns:InElement" name="InPart"/>
</message>
<message name="ResponseMessage">
  <part element="tns:OutElement" name="OutPart"/>
</message>
<portType name="HelloWorldPT">
  <operation name="sayHi">
    <input message="tns:RequestMessage"
      name="sayHiRequest"/>
    <output message="tns:ResponseMessage"
      name="sayHiResponse"/>
  </operation>
</portType>
<binding name="HelloWorldPTSOAPBinding"
  type="tns:HelloWorldPT">
  <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="HelloWorldService">
  <port binding="tns:HelloWorldPTSOAPBinding"
    name="HelloWorldPort">
    <soap:address
      location="http://localhost:9000/HelloWorldService/HelloWorldPort"
    "/>
  </port>
</service>
</definitions>

```

Abstract Data Type Definitions

Overview

Applications typically use data types that are more complex than the primitive types, like `int`, defined by most programming languages. WSDL documents represent these complex data types using a combination of schema types defined in referenced external XML schema documents and complex types described in `types` elements.

Complex type definitions

Complex data types are described in a `types` element. The W3C specification states that XSD is the preferred canonical type system for a WSDL document. Therefore, XSD is treated as the intrinsic type system. Because these data types are abstract descriptions of the data passed over the wire and not concrete descriptions, there are a few guidelines on using XSD schemas to represent them:

- Use elements, not attributes.
- Do not use protocol-specific types as base types.
- Define arrays using the SOAP 1.1 array encoding format.

WSDL does allow for the specification and use of alternative type systems within a document.

Example

The structure, `personalInfo`, defined in [Example 2](#), contains a `string`, an `int`, and an `enum`. The `string` and the `int` both have equivalent XSD types and do not require special type mapping. The enumerated type `hairColorType`, however, does need to be described in XSD.

Example 2: *personalInfo* structure

```
enum hairColorType {red, brunette, blonde};

struct personalInfo
{
    string name;
    int age;
    hairColorType hairColor;
}
```

Example 3 shows one mapping of `personalInfo` into XSD. This mapping is a direct representation of the data types defined in **Example 2**.

`hairColorType` is described using a named `simpleType` because it does not have any child elements. `personalInfo` is defined as an `element` so that it can be used in messages later in the contract.

Example 3: XSD type definition for `personalInfo`

```
<types>
  <xsd:schema targetNamespace="http://iona.com/personal/schema"
    xmlns:xsd1="http://iona.com/personal/schema"
    xmlns="http://www.w3.org/2000/10/XMLSchema"/>
    <simpleType name="hairColorType">
      <restriction base="xsd:string">
        <enumeration value="red"/>
        <enumeration value="brunette"/>
        <enumeration value="blonde"/>
      </restriction>
    </simpleType>
    <element name="personalInfo">
      <complexType>
        <element name="name" type="xsd:string"/>
        <element name="age" type="xsd:int"/>
        <element name="hairColor" type="xsd1:hairColorType"/>
      </complexType>
    </element>
</types>
```

Another way to map `personalInfo` is to describe `hairColorType` in-line as shown in **Example 4**. With this mapping, however, you cannot reuse the description of `hairColorType`.

Example 4: Alternate XSD Mapping for `personalInfo`

```
<types>
  <xsd:schema targetNamespace="http://iona.com/personal/schema"
    xmlns:xsd1="http://iona.com/personal/schema"
    xmlns="http://www.w3.org/2000/10/XMLSchema"/>
    <element name="personalInfo">
      <complexType>
        <element name="name" type="xsd:string"/>
        <element name="age" type="xsd:int"/>
      </complexType>
    </element>
</types>
```

Example 4: *Alternate XSD Mapping for personInfo (Continued)*

```
<element name="hairColor">
  <simpleType>
    <restriction base="xsd:string">
      <enumeration value="red"/>
      <enumeration value="brunette"/>
      <enumeration value="blonde"/>
    </restriction>
  </simpleType>
</element>
</complexType>
</element>
</types>
```

Abstract Message Definitions

Overview

WSDL is designed to describe how data is passed over a network. It describes data that is exchanged between two endpoints in terms of abstract messages described in `message` elements.

Each abstract message consists of one or more parts, defined in `part` elements.

These abstract messages represent the parameters passed by the operations defined by the WSDL document and are mapped to concrete data formats in the WSDL document's `binding` elements.

Messages and parameter lists

For simplicity in describing the data consumed and provided by an endpoint, WSDL documents allow abstract operations to have only one input message, the representation of the operation's incoming parameter list, and only one output message, the representation of the data returned by the operation.

In the abstract message definition, you cannot directly describe a message that represents an operation's return value. Therefore, any return value must be included in the output message.

Messages allow for concrete methods defined in programming languages like C++ to be mapped to abstract WSDL operations. Each message contains a number of `part` elements that represent one element in a parameter list.

Therefore, all of the input parameters for a method call are defined in one message and all of the output parameters, including the operation's return value, are mapped to another message.

Example

For example, imagine a server that stores personal information as defined in [Example 2 on page 25](#) and provides a method that returns an employee's data based on an employee ID number.

The method signature for looking up the data would look similar to [Example 5](#).

Example 5: *Method for Returning an Employee's Data*

```
personalInfo lookup(long empId)
```

This method signature could be mapped to the WSDL fragment shown in [Example 6](#).

Example 6: *WSDL Message Definitions*

```
<message name="personalLookupRequest">
  <part name="empId" type="xsd:int" />
</message>
<message name="personalLookupResponse">
  <part name="return" element="xsd:personalInfo" />
</message>
```

Message naming

Each message in a WSDL document must have a unique name within its namespace. Choose message names that show whether they are input messages (requests) or output messages (responses).

Message parts

Message parts are the formal data elements of the abstract message. Each part is identified by a `name` attribute and by either a `type` or an `element` attribute that specifies its data type. The data type attributes are listed in [Table 1](#).

Table 1: *Part Data Type Attributes*

Attribute	Description
<code>type="type_name"</code>	The datatype of the part is defined by a <code>simpleType</code> or <code>complexType</code> called <code>type_name</code>
<code>element="elem_name"</code>	The datatype of the part is defined by an <code>element</code> called <code>elem_name</code> .

Messages are allowed to reuse part names. For instance, if a method has a parameter, `foo`, which is passed by reference or is an in/out, it can be a part in both the request message and the response message as shown in [Example 7](#).

Example 7: *Reused Part*

```
<message name="fooRequest">
  <part name="foo" type="xsd:int"/>
</message>
<message name="fooReply">
  <part name="foo" type="xsd:int"/>
</message>
```

Abstract Interface Definitions

Overview

WSDL `portType` elements define, in an abstract way, the operations offered by a service. The operations defined in a port type list the input, output, and any fault messages used by the service to complete the transaction the operation describes.

Port types

A `portType` can be thought of as an interface description. In many Web service implementations there is a direct mapping between port types and implementation objects. Port types are the abstract unit of a WSDL document that is mapped into a concrete binding to form the complete description of what is offered over a port.

Port types are described using the `portType` element in a WSDL document. Each port type in a WSDL document must have a unique name, specified using the `name` attribute, and is made up of a collection of operations, described in `operation` elements. A WSDL document can describe any number of port types.

Operations

Operations, described in `operation` elements in a WSDL document, are an abstract description of an interaction between two endpoints. For example, a request for a checking account balance and an order for a gross of widgets can both be defined as operations.

Each operation within a port type must have a unique name, specified using the required `name` attribute.

Elements of an operation

Each operation is made up of a set of elements. The elements represent the messages communicated between the endpoints to execute the operation.

The elements that can describe an operation are listed in [Table 2](#).

Table 2: *Operation Message Elements*

Element	Description
<code>input</code>	Specifies a message that is received from another endpoint. This element can occur at most once for each operation.

Table 2: *Operation Message Elements (Continued)*

Element	Description
output	Specifies a message that is sent to another endpoint. This element can occur at most once for each operation.
fault	Specifies a message used to communicate an error condition between the endpoints. This element is not required and can occur an unlimited number of times.

An operation is required to have at least one `input` or `output` element. The elements are defined by two attributes listed in [Table 3](#).

Table 3: *Attributes of the Input and Output Elements*

Attribute	Description
name	Identifies the message so it can be referenced when mapping the operation to a concrete data format. The name must be unique within the enclosing port type.
message	Specifies the abstract message that describes the data being sent or received. The value of the <code>message</code> attribute must correspond to the <code>name</code> attribute of one of the abstract messages defined in the WSDL document.

It is not necessary to specify the `name` attribute for all input and output elements; WSDL provides a default naming scheme based on the enclosing operation's name.

If only one element is used in the operation, the element name defaults to the name of the operation. If both an `input` and an `output` element are used, the element name defaults to the name of the operation with `Request` or `Response` respectively appended to the name.

Return values

Because the port type is an abstract definition of the data passed during an operation, WSDL does not provide for return values to be specified for an operation. If a method returns a value, it is mapped into the output message as the last part of that message. The concrete details of how the message parts are mapped into a physical representation are described in [“Bindings” on page 34](#).

Example

For example, in implementing a server that stores personal information in the structure defined in [Example 2 on page 25](#), you might use an interface similar to the one shown in [Example 8](#).

Example 8: *personalInfo Lookup Interface*

```
interface personalInfoLookup
{
    personalInfo lookup(in int empID)
    raises(idNotFound);
}
```

This interface could be mapped to the port type in [Example 9](#).

Example 9: *personalInfo Lookup Port Type*

```
<types>
...
  <element name="idNotFound" type="idNotFoundType">
    <complexType name="idNotFoundType">
      <sequence>
        <element name="ErrorMsg" type="xsd:string"/>
        <element name="ErrorID" type="xsd:int"/>
      </sequence>
    </complexType>
  </types>
<message name="personalLookupRequest">
  <part name="empId" type="xsd:int" />
</message>
<message name="personalLookupResponse">
  <part name="return" element="xsdl:personalInfo" />
</message>
<message name="idNotFoundException">
  <part name="exception" element="xsdl:idNotFound" />
</message>
<portType name="personalInfoLookup">
  <operation name="lookup">
    <input name="empID" message="personalLookupRequest" />
    <output name="return" message="personalLookupResponse" />
    <fault name="exception" message="idNotFoundException" />
  </operation>
</portType>
```

Mapping to the Concrete Details

Overview

The abstract definitions in a WSDL document are intended to be used in defining the interaction of real applications that have specific network addresses, use specific network protocols, and expect data in a particular format. To fully define these real applications, the abstract definitions discussed in the previous section must be mapped to concrete representations of the data passed between applications. The details describing the network protocols in use must also be added.

This is accomplished in the WSDL `bindings` and `ports` elements. WSDL binding and port syntax is not tightly specified by the W3C. A specification is provided that defines the mechanism for defining these syntaxes. However, the syntaxes for bindings other than SOAP and for network transports other than HTTP are not defined in a W3C specification.

Bindings

Bindings describe the mapping between the abstract messages defined for each port type and the data format used on the wire. Bindings are described in `binding` elements in the WSDL file. A binding can map to only one port type, but a port type can be mapped to any number of bindings.

It is within the bindings that you specify details such as parameter order, concrete data types, and return values. For example, a binding can reorder the parts of a message to reflect the order required by an RPC call. Depending on the binding type, you can also identify which of the message parts, if any, represent the return type of a method.

Services

To define an endpoint that corresponds to a running service, the `port` element in the WSDL file associates a binding with the concrete network information needed to connect to the remote service described in the file. Each port specifies the address and configuration information for connecting the application to a network.

Ports are grouped within `service` elements. A service can contain one or many ports. The convention is that the ports defined within a particular service are related in some way. For example all of the ports might be bound to the same port type, but use different network protocols, like HTTP and WebSphere MQ.

Using Artix Designer

This chapter introduces Artix Designer, and outlines how you can use it to build a WSDL file and to generate starting point code.

In this chapter

This chapter discusses the following topics:

Introduction	page 36
Creating Artix Designer Projects	page 41
Creating a WSDL File	page 43
Defining the WSDL Elements	page 45
Developing the Applications	page 62
Adding Logic to the Code	page 68
Running the Applications	page 74

Introduction

Overview

Artix Designer is a GUI development tool that ships as a series of plug-ins to the Eclipse platform. Eclipse is an open source development platform and application framework for building software, as described at eclipse.org.

Artix Designer enables you to write and edit the WSDL files that describe Artix resources and their integration, and to generate starting point code for a Web service.

Generating WSDL

Artix Designer features a number of wizards that enable you to create WSDL files based on:

- CORBA IDL files
- Java classes
- XSD schemas
- fixed data
- tagged data
- COBOL copybook files

Using the WSDL editor

Although there are other XML editors that you can use to write WSDL, Artix Designer has an understanding of the Artix WSDL extensions and is a much easier way to write the WSDL files used in an Artix application. For example, Artix Designer automatically adds the required namespace declarations and prefix definitions when you build Artix applications that involve other data marshalling schemas, transport protocols, or routing.

The Artix Designer WSDL editor provides a number of wizards that take you through the process of creating and editing `type`, `message`, `portType`, `binding`, `service`, and `route` elements in your WSDL files.

See “[Defining the WSDL Elements](#)” on page 45 for more on using the WSDL editing wizards.

Generating code

Artix Designer's Artix Generator tool is integrated with the Artix command-line tools, such as `wsdltocpp` and `wsdltojava`, so that you can use it to generate starting point code in C++ and Java based on your WSDL files.

In addition, integration with the Eclipse Java Development Tools (JDT) and C/C++ Development Tools (CDT) means that any code you create is compiled automatically after you generate it, and is recompiled when you make any changes to your source.

Note: The **Build Automatically** option must be enabled in the Eclipse **Project** menu for code to be compiled automatically.

Artix Generator allows you to create a variety of code generation configurations, which you can save and reuse. For example, you can create configurations for:

- client and server applications
- Artix switch applications
- CORBA IDL
- Artix service plug-ins
- Container applications for hosting service plug-ins

See [“Developing the Applications” on page 62](#) for an example of the Artix Generator at work.

Launching Artix Designer

To launch Artix Designer in Windows, select **Start | (All) Programs | IONA | Artix 3.0 | Artix Designer**.

To launch Artix Designer in Linux:

1. Change to the following directory:

```
InstallDir/artix/3.0/bin
```

2. Run the following command:

```
./start_eclipse
```

The Eclipse platform launches with the Artix Designer plug-ins loaded.

Note: You can install Artix Designer into an existing Eclipse 3.0.x installation, as described in “Configuring Eclipse for Artix Designer” in the [Artix 3.0 Installation Guide](#). Artix Designer 3.0 does *not* support Eclipse versions 3.1 or later.

Artix Designer projects

In Eclipse, all development is performed within a project. Artix Designer provides the following project types:

- Basic Web services projects, either empty or template-based
- CORBA Web services projects

An empty project simply creates an Eclipse `.project` file and the directory structure that you are likely to use when developing your Web services project.

A template-based project creates all the starting point code and configuration information needed for your Web services application.

The following templates are available:

- Artix switch
- C++ client
- C++ server
- C++ client and server
- Java client
- Java server
- Java client and server

Note: You must have a valid WSDL file ready before you can create a Web services project from a template.

A CORBA Web services project creates a WSDL file and a switch configuration based on a CORBA IDL data source.

See “[Creating Artix Designer Projects](#)” on page 41 for more information.

The Artix perspective

When you create an Artix Designer project, you are prompted to switch to the Artix perspective. Perspectives are predefined layouts of the windows, views, and tools in the Eclipse window.

The Artix perspective provides you with all the tools that you need to develop an Artix project in Eclipse. It includes the following features:

- The Artix toolbar
- The Navigator view
- The Outline view

Using the Artix toolbar

The Artix toolbar gives you quick access to the main Artix Designer functionality.

It contains the following buttons:

Table 4: *Artix Designer toolbar buttons*

Button	Description
	Add <code>import</code> element to currently selected WSDL file
	Add <code>type</code> element to WSDL
	Add <code>message</code> element
	Add <code>portType</code> element
	Add <code>binding</code> element
	Add <code>service</code> element
	Add <code>route</code> element
	Run Artix Generator. ^a

a. If a code generation configuration already exists, clicking this button launches the last-used configuration. Click the down arrow next to the button to run other configurations.

Cheat sheets

Artix Designer comes with a number of cheat sheets that guide you through common tasks, such as creating an Artix Designer project, creating a WSDL file, and generating code.

Each cheat sheet lists the steps required to complete a particular task. As you progress from one step to the next, the cheat sheet automatically launches the required tools for you.

To view the available Artix Designer cheat sheets, select **Help | Cheat Sheets**.

Online Help

Help on Artix Designer is available from within the Eclipse online Help system.

Select **Help | Help Contents** to view the Eclipse Help. The Artix Designer Help section is listed in the table of contents frame on the left.

In addition, you can access context-sensitive Help from within the Artix Designer wizards and the Artix Generator window by pressing **F1**.

Creating Artix Designer Projects

Overview

This section outlines how to create Artix Designer projects in Eclipse.

Here you will create two empty, basic Web services projects: one for your Java code and one for your C++ code.

Note: You must keep your C++ and Java code in separate projects, because Eclipse does not support the use of the JDT and the CDT in the same project.

Creating the Web services projects

To create a basic Web services project:

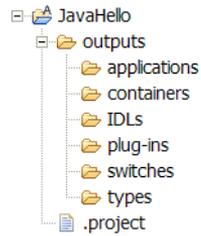
1. In Eclipse, select **File | New | Project**.
2. In the New Project dialog box, expand the **IONA Artix Designer** folder.
3. Select **Basic Web Services Project** and then click **Next**.
4. Type **JavaHello** in the **Project name** text box.
5. Leave the **Use default** checkbox selected, unless you want to store the project somewhere other than your Eclipse workspace.
6. Artix Designer allows you to specify a template when creating a new Web services project, and the template sets up files and a directory structure for you.

For this tutorial, we will instead create an empty project and add the WSDL file and other configuration files later in this chapter. Make sure the **Use a template** checkbox is cleared.

7. Click **Finish**.
8. You are prompted to open the Artix perspective, which is associated with Artix Designer projects. Click **Yes**.

A **JavaHello** project, containing a number of empty folders, is added to the Navigator view in Eclipse.

Figure 3: *The Empty JavaHello Project in the Navigator View*



Now create a second empty, basic Web services project and name it CppHello.

Your Eclipse workspace now displays two Artix projects in the Navigator view:

- JavaHello
- CppHello

Creating a WSDL File

Overview

This section explains how to use Artix Designer to create a simple WSDL file that forms the basis of your Web services application. The same WSDL file is used to generate both Java and C++ versions of the application.

Creating an empty WSDL file

To create an empty WSDL file:

1. In the Eclipse workspace, select **File | New | Other**.
2. In the New dialog box, expand the **IONA Artix Designer** folder.
3. Select **WSDL File** and click **Next**.
4. In the **WSDL File** panel, select the **JavaHello** project folder. This specifies where the WSDL file is to be stored.
5. In the **File name** text box, type `HelloWorld`.
6. Click **Finish**.

The `HelloWorld.wsdl` file opens in the WSDL Editor.

Linking to the WSDL file from CppHello

To generate the C++ client and server code from the same `HelloWorld.wsdl` WSDL file, link to the file from the CppHello project:

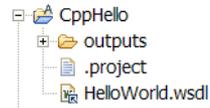
1. Follow steps [1](#) to [3](#) above.
2. In the **WSDL File** panel, select the **CppHello** project folder.
3. In the **File name** text box, type `HelloWorld`.
4. Click the **Advanced** button.
5. Select the **Link to file in the file system** checkbox and click **Browse**.
6. Browse to the `EclipseWorkspace\JavaHello` directory, select the **HelloWorld.wsdl** file, and click **Open**.

Note: You specified the location of your `EclipseWorkspace` directory when you first started Eclipse. The default location for Windows users is in your My Documents directory. The default location for UNIX/Linux users is `ArtixInstallDir/artix/3.0/eclipse/workspace`

7. Click **Finish**.

The `HelloWorld.wsd1` file now appears as a link in the CppHello project.

Figure 4: *The CppHello Project With a Link to the HelloWorld.wsd1 File*



Note: When you link to a file, the same file is used by both the CppHello and JavaHello projects.

It is also possible to import the WSDL file into the CppHello project by selecting **File | Import**. However, this would create a separate physical file, and any changes you made to one WSDL file would not be replicated in the other.

Defining the WSDL Elements

Overview

Next, add the elements to make the WSDL file a valid Artix contract.

Artix Designer provides a series of wizards that allow you to create each of these elements.

This section guides you through the task of creating the contract elements in the following topics:

- [“Defining Types” on page 46](#)
- [“Defining Messages” on page 50](#)
- [“Defining Port Types” on page 52](#)
- [“Defining Bindings” on page 55](#)
- [“Defining a Service” on page 58](#)

Defining Types

Overview

The `types` element of the WSDL file contains all the data types used between the client and server.

In this simple example, we will create two `element` types of type `xsd:string`:

- `InElement`, which maps to the *in* part of the request message that you will create later
 - `OutElement`, which maps to the *out* part of the response message.
-

Defining element types

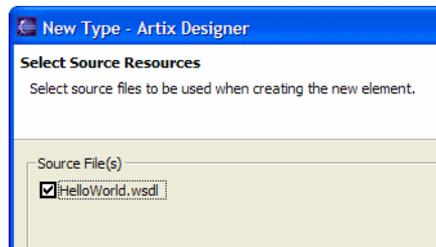
To define the `InElement` type:

1. Open the `HelloWorld.wsdl` file from either the `JavaHello` or the `CppHello` project.
2. Click the **Diagram** tab at bottom of the WSDL Editor view.
3. In the Diagram view, right-click the **Types** node.

Note: You can also add elements to a WSDL file from the **Artix Designer** menu, or by clicking the appropriate icon in the Artix toolbar. See [Table 4 on page 39](#) for more on the available icons.

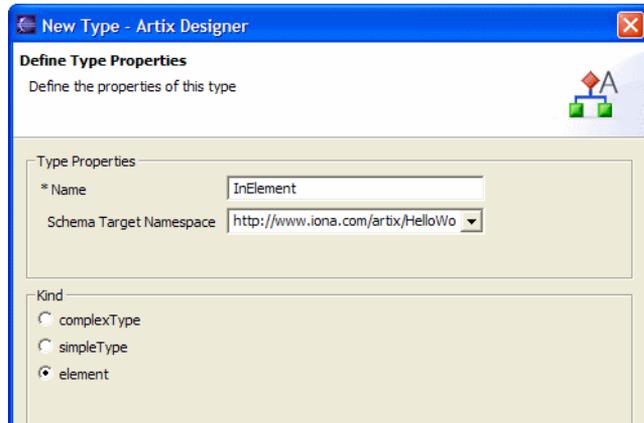
4. Select **New Type** from the pop-up menu. The New Type wizard opens.
5. In the Select Source Resources panel, make sure **HelloWorld.wsdl** is selected in the **Source File(s)** section.

Figure 5: *The Select Source Resources Panel*



6. Click **Next** to display the Define Type Properties panel.
7. Type **InElement** in the **Name** text box.
8. Accept the default target namespace provided. Select the **element** radio button.

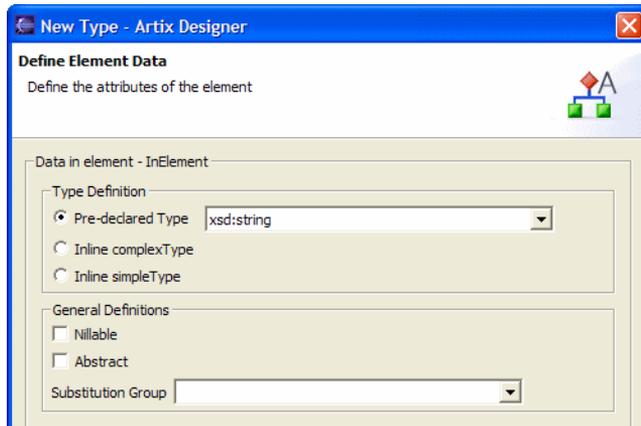
Figure 6: *The Define Type Properties Panel*



9. Click **Next** to display the Define Element Data panel.

10. Select the **Pre-declared Type** button, then select **xsd:string** from the drop-down list.

Figure 7: *The Define Element Data Panel*



11. Click **Next** to display the View Type Summary panel and click **Finish**.

To define the `OutElement` type:

1. Repeat steps 2 to 6 above.
2. In the Define Type Properties panel, enter `OutElement` in the **Name** field.
3. Select the **element** radio button, and click **Next**.
4. In the Define Element Data panel, select **Pre-declared Type** and select **xsd:string** from the drop-down list.
5. Click **Next**.
6. In the View Type Summary panel, click **Finish**.
7. Save the WSDL file by selecting **File|Save** from the menu bar or by right-clicking in the Source view and selecting **Save**.

Review

Click the **Source** tab at the bottom of the WSDL Editor view to look over the WSDL file created so far.

In the Outline view in the lower left of the Eclipse window, open the **Types** node. Click the name of a `types` element to jump to that element in the WSDL Editor view.

```
<types>
  <schema targetNamespace="http://www.iona.com/artix/HelloWorld"
    xmlns="http://www.w3.org/2001/XMLSchema">
    <element name="InElement" type="string"/>
    <element name="OutElement" type="string"/>
  </schema>
</types>
```

Defining Messages

Overview

Now that you have created the WSDL types, you can define the request and response messages for your Web service.

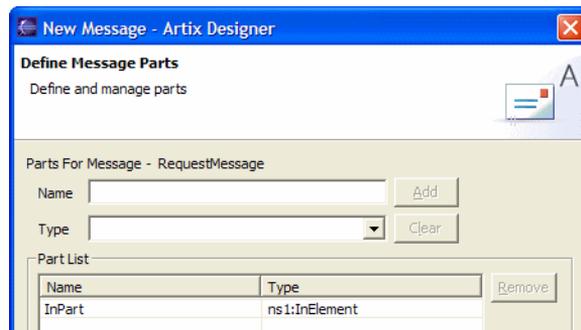
You will use your types as the message parts.

Defining messages

To define the request message:

1. With the `HelloWorld.wsdl` file open and the Diagram view displayed, right-click the **Messages** node.
2. Select **New Message** from the pop-up menu. The New Message wizard opens.
3. In the Select Source Resources panel, make sure **HelloWorld.wsdl** is selected in the **Source File(s)** section.
4. Click **Next** to display the Define Message Properties panel.
5. Type `RequestMessage` in the **Name** text box.
6. Click **Next** to display the Define Message Parts panel.
7. Type `InPart` in the **Name** text box and select `ns1:InElement` from the **Type** drop-down list.
8. Click **Add** to add the message part to the **Part List** section.

Figure 8: *The Define Message Parts Panel*



9. Click **Next** to display the View Message Summary panel.

10. Click **Finish**.

To define the response message:

1. Repeat steps 1 to 4 above.
 2. In the Define Message Properties panel, type **ResponseMessage** in the **Name** text box and click **Next**.
 3. In the Define Message Parts panel, type **OutPart** in the **Name** text box and select **ns1:OutElement** from the **Type** drop-down list.
 4. Click **Add** to add the message part to the **Part List** section.
 5. Click **Next** to display the View Message Summary panel.
 6. Click **Finish**.
 7. Save the WSDL file.
-

Review

You have now added request and response messages to your WSDL file.

The request message includes an in part that maps to the `InElement` type, and the response message includes an out part that maps to the `OutElement` type.

```
<message name="RequestMessage">
  <part element="tns:InElement" name="InPart"/>
</message>
<message name="ResponseMessage">
  <part element="tns:OutElement" name="OutPart"/>
</message>
```

For a more thorough explanation of creating messages, see [Designing Artix Solutions](#).

Defining Port Types

Overview

The `portType` element contains operations, which are composed of one or more messages:

- A one-way operation includes only an input message; the client application does not receive a response from the Web service.
- A request-response operation includes an input message, an output message, and zero or more fault messages¹.

In this example, you will define a port type that includes one request-response operation called `sayHi` which uses `RequestMessage` as its input and `ResponseMessage` as its output.

There is nothing significant about the names assigned to the messages or parts; name assignments are to assist the developer. Artix does not care what names are used.

Defining a port type

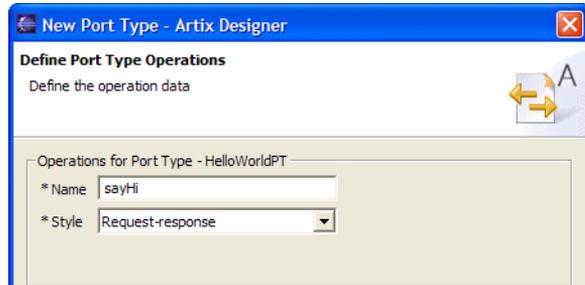
To define a port type:

1. With the `HelloWorld.wsdl` file open and the Diagram view displayed, right-click the **Port Types** node.
2. Select **New Port Type** from the pop-up menu. The New Port Type wizard opens.
3. In the Select Source Resources panel, make sure **HelloWorld.wsdl** is selected in the **Source File(s)** section.
4. Click **Next** to display the Define Port Type Properties panel.
5. Type `HelloWorldPT` in the **Name** text box.
6. Click **Next** to display the Define Port Type Operations panel.
7. Type `sayHi` in the **Name** text box.

1. Defining and coding fault messages is discussed in “Creating User-Defined Exceptions” in both [Developing Artix Applications in C++](#) and [Developing Artix Applications in Java](#).

8. Select **Request-response** from the **Style** drop-down list.

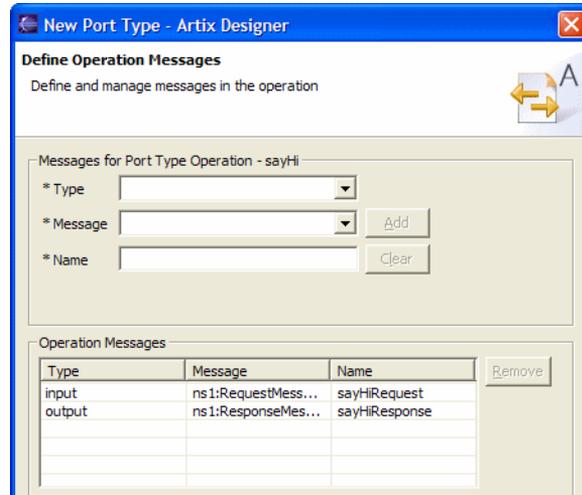
Figure 9: *The Define Port Type Operations Panel*



9. Click **Next** to display the Define Operation Messages panel.
10. In the **Type** drop-down list, select **input**.
11. In the **Message** drop-down list, select **ns1:RequestMessage**.
The name **sayHiRequest** appears in the **Name** text box. You can change this to something more meaningful for your application if you prefer. For this tutorial, leave the suggested name as is.
12. Click **Add** to add the operation to the **Operation Messages** section.
13. Expand the **Type** drop-down list again. Note that **input** no longer appears in the list, because an operation can have only one input message.
14. Select **output** from the **Type** list and **ns1:ResponseMessage** from the **Message** list.
The name **sayHiResponse** appears in the **Name** text box. Leave the suggested name as is.

- Click **Add** to add the operation to the **Operation Messages** section.

Figure 10: *The Define Operation Messages Panel*



Note: Expand the **Type** drop-down list again and notice that the **output** entry no longer appears in the list. This is because an operation can have only one output message.

Although this example does not include any fault messages, you can add one or more fault messages to each operation.

- Click **Next** to display the Port Type Summary panel.
- Click **Finish** to close the wizard.
- Save the WSDL file.

Review

You have now added the following `portType` element to your WSDL file.

```
<portType name="HelloWorldPT">
  <operation name="sayHi">
    <input message="tns:RequestMessage" name="sayHiRequest"/>
    <output message="tns:ResponseMessage" name="sayHiResponse"/>
  </operation>
</portType>
```

Defining Bindings

Overview

The `binding` element in a WSDL file defines the message format and protocol details for each port. Each binding is associated with a single `portType` element, although the same `portType` can be associated with multiple bindings.

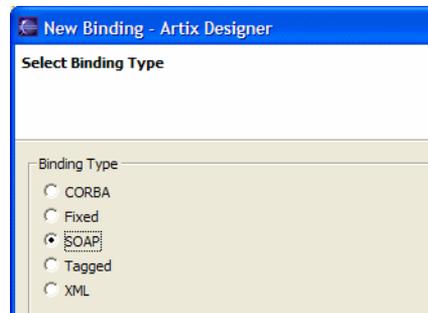
In this example, you will specify the document/literal binding style, which is required when message parts are element types.

Defining a binding

To define a binding:

1. With the `HelloWorld.wsdl` file open and the Diagram view displayed, right-click the **Bindings** node.
2. Select **New Binding** from the pop-up menu. The New Binding wizard opens.
3. In the Select Source Resources panel, make sure **HelloWorld.wsdl** is selected in the **Source File(s)** section.
4. Click **Next** to display the Select Binding Type panel.
5. Select **SOAP** from the list of binding types.

Figure 11: *The Select Binding Type Panel*



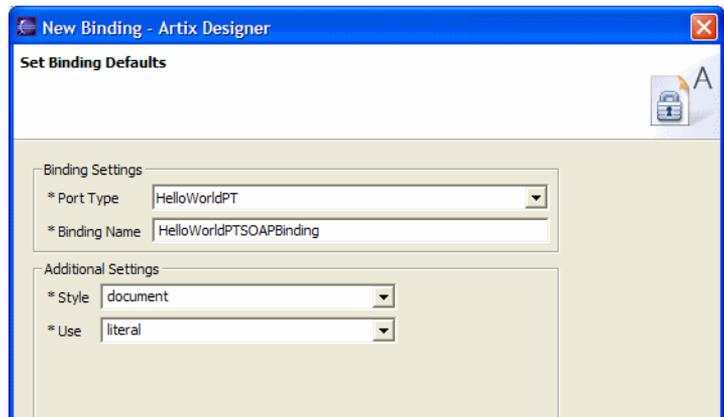
6. Click **Next** to display the Set Binding Defaults panel.
7. Select **HelloWorldPT** from the **Port Type** drop-down list.

In this case, your WSDL file contains only one `portType` element. If there were multiple port types, you would select one from the drop-down list.

Note: A name is already entered in the **Binding** text box. You can change this entry, but be sure to give each binding in the WSDL file a unique name.

8. In the **Additional Settings** section, select **document** from the **Style** drop-down list, and select **literal** from the **Use** drop-down list.

Figure 12: *The Set Binding Defaults Panel*



9. Click **Next** to display the Edit Binding panel.
10. In the Operations Editor on the left, expand the **Operations** node and then click each **sayHi** operation node to review its binding details.
11. Click **Next** to display the View Binding Summary panel.
12. Click **Finish** to close the wizard.
13. Save the WSDL file.

Review

You have now added the following `binding` element to your WSDL file.

```
<binding name="HelloWorldPTSOAPBinding" type="tns:HelloWorldPT">
  <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>
```

Defining a Service

Overview

The `service` element of a WSDL file provides transport-specific information. Each service element can include one or more `port` elements. Each `port` element must be uniquely identified by the value of its `name` attribute.

Each `port` element is associated with a single `binding` element, although the same `binding` element can be associated with one or more `port` elements. In addition, a WSDL file may contain multiple `service` elements.

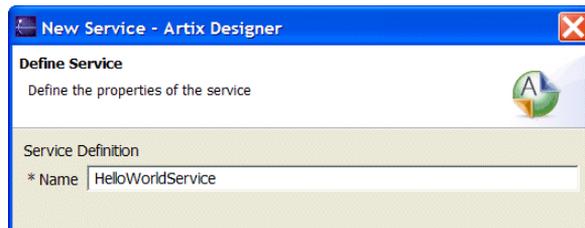
In this example, the WSDL file contains one `service` element, which contains a single `port` element.

Defining a service

To define a service:

1. With the `HelloWorld.wsdl` file open and the Diagram view displayed, right-click the **Services** node.
2. Select **New Service** from the pop-up menu. The New Service wizard opens.
3. In the Select Source Resources panel, make sure **HelloWorld.wsdl** is selected in the **Source File(s)** section.
4. Click **Next** to display the Define Service panel.
5. Type `HelloWorldService` in the **Name** text box.

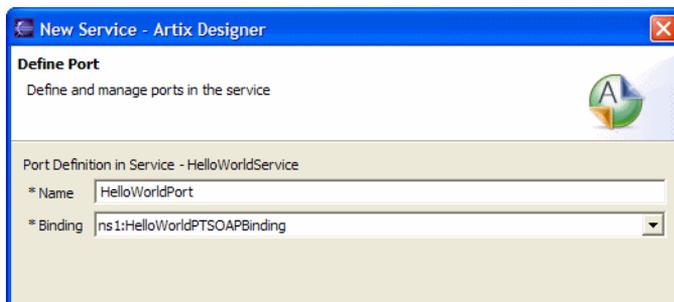
Figure 13: *The Define Service Panel*



6. Click **Next** to display the Define Port panel.
7. Type `HelloWorldPort` in the **Name** text box.

8. Select **ns1:HelloWorldPTSOAPBinding** from the **Binding** drop-down list.

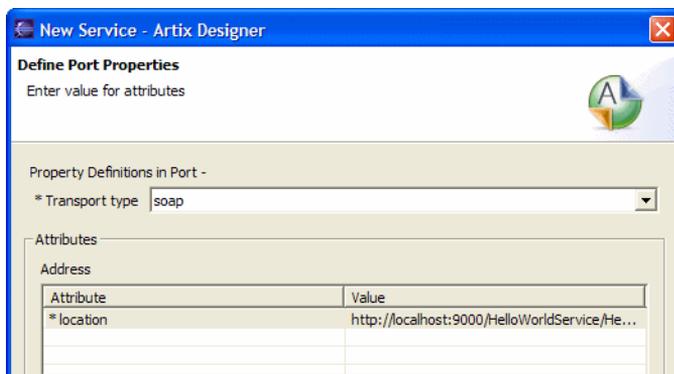
Figure 14: *The Define Port Panel*



9. Click Next to display the Define Port Properties panel.
10. From the **Transport Type** drop-down list, select **soap**.
11. In the **Address** section, click below the **Value** header and type the following as the value for the location attribute:

`http://localhost:9000/HelloWorldService/HelloWorldPort`

Figure 15: *The Define Port Properties panel*



12. Click **Next** to display the View Service and Port Summary panel.

13. Click **Finish** to close the wizard.
14. Save the WSDL file.

Review

You have now completed your WSDL file and are ready to use it to develop an application.

Click the **Source** tab in the WSDL Editor to review the WSDL that you have created. It should look like this:

Example 10: *The completed HelloWorld.wsdl file*

```
<?xml version="1.0" encoding="UTF-8"?>
<!--WSDL file template-->
<!--Created by IONA Artix Designer-->
<definitions name="HelloWorld.wsdl"
  targetNamespace="http://www.iona.com/artix/HelloWorld"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.iona.com/artix/HelloWorld"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <types>
    <schema
      targetNamespace="http://www.iona.com/artix/HelloWorld"
      xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="InElement" type="string"/>
      <element name="OutElement" type="string"/>
    </schema>
  </types>
  <message name="RequestMessage">
    <part element="tns:InElement" name="InPart"/>
  </message>
  <message name="ResponseMessage">
    <part element="tns:OutElement" name="OutPart"/>
  </message>
  <portType name="HelloWorldPT">
    <operation name="sayHi">
      <input message="tns:RequestMessage" name="sayHiRequest"/>
      <output message="tns:ResponseMessage"
        name="sayHiResponse"/>
    </operation>
  </portType>
</definitions>
```

Example 10: *The completed HelloWorld.wsdl file (Continued)*

```
<binding name="HelloWorldPTSOAPBinding"
  type="tns:HelloWorldPT">
  <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="HelloWorldService">
  <port binding="tns:HelloWorldPTSOAPBinding"
    name="HelloWorldPort">
    <soap:address location=
      "http://localhost:9000/HelloWorldService/HelloWorldPort"/>
  </port>
</service>
</definitions>
```

Developing the Applications

Overview

In this section, you will develop client and server applications in both Java and C++ based on the `HelloWorld.wsdl` file.

To do this, you will use the Artix Generator tool to generate the code and any other necessary configuration files.

Compiling the code automatically

Because Artix Designer is integrated with the Eclipse JDT and CDT, you can ensure that your code is compiled automatically as soon as it is generated. In addition, any changes you make to a Java or C++ file will be recompiled as soon as you save the file.

To make sure your code is compiled automatically, select **Build Automatically** from the **Project** menu in Eclipse.

Creating code generation configurations

Artix Generator

The Artix Generator allows you to create and manage code generation configurations, and to generate code.

In this example, we will create separate code generation configurations for a

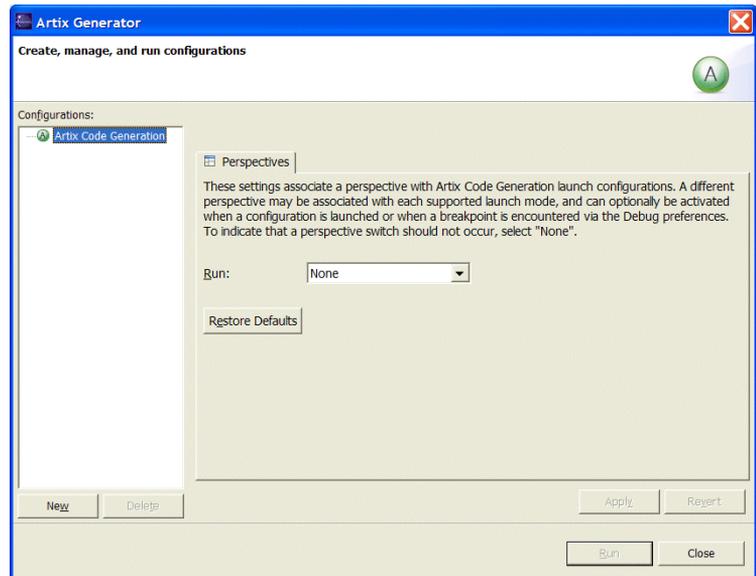
- Java client and server
- C++ client and server

Creating the Java client and server

To create the Java client and server configuration:

1. From the **Artix Designer** menu, select **Artix Generator**.
2. In the Artix Generator window, click **New** to create a configuration.

Figure 16: *The Artix Generator Window*



3. In the **Name** text box, replace the default name with something more meaningful, such as **JavaHello**.

4. In the General Details tabbed page:
 - i. Type a description for the configuration in the **Description** text box. For example, enter **Configuration for HelloWorld Java client and server**.
 - ii. From the **Targeted Project** drop-down list, select **JavaHello**.
 - iii. From the **WSDL (XSD) File** drop-down list, select **HelloWorld.wsdl**.
 - iv. Make sure the **Installation Directory** text box displays the path to your Artix installation directory; for example `c:\iona`.
 - v. Make sure the **Environment File** text box displays the correct path to your `artix_env` script.

Figure 17: *The General Tab*

The screenshot shows the 'General' tab of the Artix Designer configuration window. It is divided into two sections: 'General Details' and 'Artix Details'. In the 'General Details' section, the 'Description' field contains 'HelloWorld Java client and server', the 'Targeted Project' dropdown is set to 'JavaHello', and the 'WSDL (XSD) File' dropdown is set to 'HelloWorld.wsdl'. In the 'Artix Details' section, the 'Installation Directory' field contains 'C:\IONA' with a 'Browse' button to its right, the 'Version Number' field contains '3.0', and the 'Environment File' field contains 'C:\IONA\artix\3.0\bin\artix_env.bat'.

5. Click the **Generation** tab:
 - i. In the Generation Type section, select **Application**.
 - ii. In the Application Type section, select **Client and server**.

- iii. In the Development Language section, select **Java**.

Figure 18: *The Generation Tab*



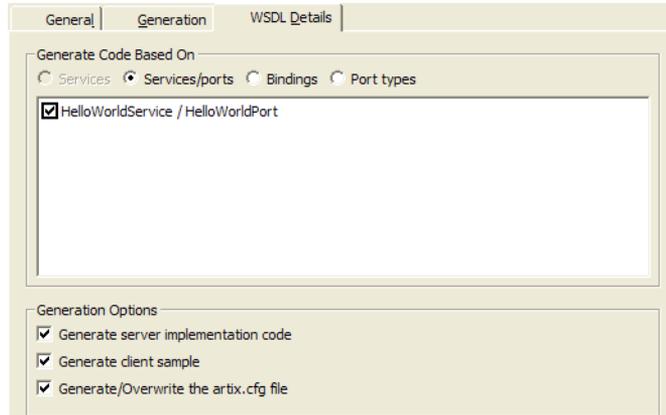
The screenshot shows a software interface with three tabs: "General", "Generation", and "WSDL Details". The "Generation" tab is active. It contains several sections:

- Generation Type:** Radio buttons for "Container", "Service plug-in", "Application" (selected), "CORBA IDL", "Switch", and "Types".
- Application Type:** Radio buttons for "Client", "Server", and "Client and server" (selected).
- Development Language:** Radio buttons for "C++" and "Java" (selected).
- Optional Services:** Checkboxes for "Locator", "Management", and "Session Manager", all of which are unchecked.
- Java Code Generation Options:**
 - Checked checkboxes for "Override namespace as package name" and "Use default package name".
 - A text field for "Java Package:" containing the value "com.iona.artix.JavaHello".
 - Unchecked checkbox for "Generate an Ant script".

6. Click the **WSDL Details** tab. Then:
 - i. Select the **Services/ports** radio button.
 - ii. Make sure the **HelloWorldService/HelloWorldPort** checkbox is checked.

- iii. In the Generation Options section, select all three check boxes.

Figure 19: *The WSDL Details Tab*



7. Click **Run**.

Artix Generator creates all the Java classes and configuration files for your client application. The Eclipse JDT compiles the code automatically.

The source code is stored in the following location:

```
EclipseWorkspace\JavaHello\outputs\applications\JavaHello\src\com
\iona\artix\JavaHello
```

And the compiled bytecode is written to the following location:

```
EclipseWorkspace\JavaHello\bin
```

Creating the C++ client and server

Since the Java and C++ applications are so similar, you can create the C++ configuration quickly by creating a duplicate of the Java configuration and editing it.

Note: Make sure you have sourced the correct version of Visual C++ in your `start_eclipse` script before creating the C++ configuration. See the [Installation Guide](#) for details.

To create the C++ client and server configuration:

1. In the Eclipse menu, select the **Artix Designer | Artix Generator** menu.
2. In the Artix Generator window, in the Configurations tree on the left, right-click the **JavaHello** configuration and select **Duplicate** from the context menu.
3. In the **Name** text box, change the name to **CppHello**.
4. In the General tabbed page:
 - i. Edit the description.
 - ii. From the **Targeted Project** drop-down list, select **CppHello**.
 - iii. From the **WSDL (XSD) File** drop-down list, select **HelloWorld.wsdl**.
5. Click the **Generation** tab. Then:
 - i. In the Generation Type section, select **Application**.
 - ii. In the Application Type section, select **Client and server**.
 - iii. In the Development Language section, select **C++**.
 - iv. In the C++ Code Generation section, select the **Generate a make file** checkbox.
6. Click the **WSDL Details** tab. Then:
 - i. Select the **Services/ports** radio button.
 - ii. Make sure the **HelloWorldService/HelloWorldPort** checkbox is checked.
 - iii. In the Generation Options section, check all three options.
7. Click **Run**.

Artix Generator creates all the C++ source and header files for your client and server applications in the following location:

```
EclipseWorkspace\CppHello\outputs\applications\CppHello\src
```

Adding Logic to the Code

Overview

Through the code generation process, you have created client and server applications in both C++ and Java.

All of these applications compile and run. However, because there is no business logic in the server implementation, and because the C++ client code does not actually make a request against the Web service, running the applications does not produce output.

You need to complete the coding in the files representing the C++ and Java implementation objects and in the C++ client mainline file. The Java client mainline file is a complete but very basic application, and thereby needs no modification.

In this section

This section contains the following topics:

- [“The C++ Code” on page 69](#)
- [“The Java Code” on page 72](#)

The C++ Code

Overview

The code generation wizard produces several files. This subsection explains the purpose of each of these files. The files are:

- [“HelloWorldPT.h”](#)
- [“HelloWorldPTClient.h/.cxx”](#)
- [“HelloWorld_wsdlTypesFactory.h/.cxx” on page 69](#)
- [“HelloWorldPTClientSample.cxx” on page 70](#)
- [“HelloWorldPTServer.h/.cxx” on page 70](#)
- [“HelloWorldPTServerSample.cxx” on page 70](#)

HelloWorldPT.h

This header file is common to both the client and server applications. It contains the signatures for each of the Web service operations. Open this file and review the signature for the `sayHi` method.

```
virtual void
sayHi (
    const IT_Bus::String &InPart,
    IT_Bus::String &OutPart
) IT_THROW_DECL((IT_Bus::Exception)) = 0;
```

HelloWorldPTClient.h/.cxx

These files represent the client proxy class. Your client mainline code must instantiate an instance of this class to invoke on the Web service. The proxy class includes multiple constructors, a destructor, and a method for each of the Web service's operations.

In this simple application, your client code uses the no argument constructor. Alternative constructors allow you to change the WSDL file, service name, or port name initialization values. One constructor allows initialization from an Artix reference.

HelloWorld_wsdlTypesFactory.h/.cxx

These files are common to both the client and server applications and include definitions and implementations for the factory methods required if your application-specific types includes the `anyType`.

For this tutorial, you do not need to be concerned with the contents of these files.

HelloWorldPTClientSample.cxx

For the client application, you only need to work with the `HelloWorldPTClientSample.cxx` file.

The code generation process produced a simple invocation of the `sayHi` method, but the code is commented out, there is no value assigned to the `InPart` string, and there is no output statement.

To add these features, complete the following steps:

1. Open the `HelloWorldPTClientSample.cxx` file and add the following code:

```
IT_Bus::String InPart="Artix User";
IT_Bus::String OutPart;
client.sayHi(InPart, OutPart);
cout << OutPart << endl;
```

2. Save and exit the file.

The Eclipse CDT automatically recompiles the code with your changes.

HelloWorldPTServer.h/.cxx

These files represent the server stub class. Your code does not directly use this class. Rather, the implementation class is a subclass of the `HelloWorldPTServer` class.

For this tutorial, you do not need to be concerned with the contents of these files.

HelloWorldPTServerSample.cxx

This file represents the server mainline application. For this tutorial, you do not need to edit the contents of this file. The server mainline instantiates an instance of the implementation class and registers it with the Artix runtime. The process then enters an event loop to process incoming requests.

HelloWorldPTImpl.h/.cxx

These files represent your Web service's implementation class. The `HelloWorldPTImpl.cxx` file contains compilable code, but there is no processing logic in the method bodies.

For the server application, you must add processing logic to the `sayHi` method in the implementation class. To do this, complete the following steps:

1. Open the `HelloWorldPTImpl.cxx` file and note the signature for the `sayHi` method.

The method includes two parameters, the first representing the part within the input message and the second representing the part within the output message. The return type is `void`.

2. Add the following code to the `sayHi` method body:

```
OutPart="Hello " + InPart;  
return;
```

3. Save and exit the file.

The Java Code

Overview

The code generation produced several files. This subsection explains the purpose of each of these files. The files are:

- “HelloWorldPT.java”
 - “HelloWorldPTTypeFactory.java”
 - “HelloWorldPTDemo.java”
 - “HelloWorldPTServer.java”
 - “HelloWorldPTImpl.java”
-

HelloWorldPT.java

This file represents the interface definition common to both the client and server applications. This interface defines the operation offered by the Web service.

```
public String sayHi(String inPart) throws RemoteException
```

HelloWorldPTTypeFactory.java

Definition of the classes that create and manage any types defined in your WSDL file.

HelloWorldPTDemo.java

This file represents the client mainline application. For this simple example, the generated code in this file represents a fully functioning application. With a more involved application, you would use this code as a template for writing a more complex client application.

HelloWorldPTServer.java

This file contains starting point code for a server mainline application. For this simple example, the generated code in this file represents a fully functional application. With a more involved application, you might extend the generated code.

You may want to add the following line to the `main()` method, before the `bus.run();` statement, so that something appears in the Eclipse Console view when the server runs:

```
System.out.println("Starting server ...");
```

The Eclipse JDT automatically recompiles the bytecode with your changes.

HelloWorldPTImpl.java

This file contains the starting point code for your Web service's implementation class. For this example, you need to modify the `sayHi` method.

To complete the `sayHi` method:

1. Open the file `HelloWorldPTImpl.java`.
2. Add the following code to the `sayHi` method body:

```
return "Hello " + inPart;
```

3. Save the file.

Running the Applications

Overview

You are now ready to run the client and server applications in both C++ and Java.

You can launch both sets of applications, Java and C++, from within the Eclipse environment, although the procedures for each is slightly different.

Running the C++ applications

To run the C++ client against the C++ server:

1. Open a command prompt and change directory to the following:

```
EclipseWorkspace\CppHello\outputs\applications\CppHello\bin
```

2. Run the `start_service>HelloWorldPTServer` script.
The server application launches in a new command window.
3. Run the `start_client>HelloWorldPTClient` script.
The client application launches and displays the words “Hello Artix User”.

Press **Ctrl+C** to close the command windows.

Running the Java applications

To run the Java server:

1. Right-click the **JavaHello** project folder and select **Run|Run** from the context menu.
2. Select **JavaHello>HelloWorldPTServer_server** from the **Configurations** tree on the left.
3. Click **Run**.

The server process starts running in the Eclipse Console view.

To run the Java client:

1. Right-click the **JavaHello** project folder and select **Run|Run** from the context menu.
2. Select **JavaHello>HelloWorldPTDemo_client** from the **Configurations** tree on the left.
3. Click the **Arguments** tab.

4. In the **Program Arguments** text box, add the following to the end of the argument:

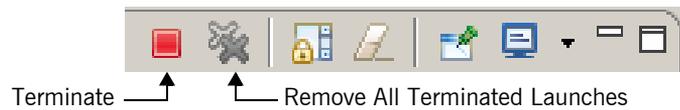
```
sayHi "Artix User"
```

5. Click **Apply**.
6. Click **Run**.

The words "Hello Artix User" appear in the Eclipse Console view.

To clear the client output, click the **Remove All Terminated Launches** button on the top right of the Console view.

Figure 20: *The Terminate and Remove All Terminated Launches Buttons*



To stop the server process, click the **Terminate** button.

Click the **Remove All Terminated Launches** button again to clear the server output.

Glossary

A

ART

IONA's Adaptive Runtime Technology (ART) is a modular framework for constructing distributed systems, based on a lightweight core and an open-ended set of plug-ins.

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

CDT

C/C++ Development Tools, a subsystem of the Eclipse development environment that automates the writing and testing of applications in C and 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 elements.

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 elements.

CORBA

CORBA (Common Object Request Broker Architecture) defines standards for interoperability and portability among distributed objects, independently of the language in which those objects are written. It is a robust, industry-accepted standard from the OMG (Object Management Group), deployed in thousands of mission-critical systems.

CORBA also specifies an extensive set of services for creating and managing distributed objects, accessing them by name, storing them in persistent stores, externalizing their state, and defining ad hoc relationships between them. An ORB is the core element of the wider OMG framework for developing and deploying distributed components.

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**EAI**

Enterprise Application Integration, the use of software and architectural principles to integrate disparate enterprise applications.

Eclipse

An open source application development framework developed by the Eclipse Foundation. See eclipse.org.

Embedded Mode

Operational 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 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 endpoint can be compiled into an application). Contrast with Service.

Enterprise Service Bus

See [Service bus](#).

H

Host

The network node on which a particular service resides.

J

JDT

Java Development Tools, a subsystem of the Eclipse development environment that automates the writing and testing of applications in Java.

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 port type, which allows for a logical contract to have multiple bindings and thus multiple wire message formats for the same contract.

Message

A WSDL message is an abstract definition of the data being communicated. Each part of a message is associated with defined types. A WSDL message is analogous to a parameter in object-oriented programming.

O

Operation

A WSDL operation is an abstract definition of the action supported by the service. It is defined in terms of input and output messages. An operation is loosely analogous to a function or method in object-oriented programming, or a message queue or business process.

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 file using the binding definition.

Port Type

A WSDL port type is a collection of abstract operations, supported by one or more endpoints. A port type is loosely analogous to a class in object-oriented programming. A port type can be mapped to multiple transports using multiple bindings.

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 WSDL contract and apply to both endpoints 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 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.

SOAP

SOAP is an XML-based messaging framework specifically designed for exchanging formatted data across the Internet. It can be used for sending request and reply messages or for sending entire XML documents. As a protocol, SOAP is simple, easy to use, and completely neutral with respect to operating system, programming language, or distributed computing platform.

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 interoperability 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` element of an Artix contract.

Type

A WSDL data type is a container for data type definitions that is used to describe messages (for example, an XML schema).

W

Web Services Description Language

See [WSDL](#).

WSDL

WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information.

A WSDL document defines services as collections of network endpoints, or ports. 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 XSD).
- **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.

Source: Web Services Description Language (WSDL) 1.1. W3C Note 15 March 2001. (<http://www.w3.org/TR/wsdl>)

X

XML

XML is a simpler but restricted form of SGML (Standard Generalized Markup Language). The markup describes the meaning of the text. XML enables the separation of content from data. XML was created so that richly structured documents could be used over the web.

XSD

XML Schema Definition (XSD) is the language used to define an XML Schema. The XML Schema defines the structure of an XML document.

In Artix, a schema can be a standalone resource within a collection, or it can be used as an import to define the types within a WSDL contract.

XSLT

XSLT stands for XSL Transformations. XSL is a stylesheet language for XML. XSLT is a set of extensions to XSL that allows one XML document to be transformed into another. For more information see the [XSLT specification](#).

Index

A

Adaptive Runtime Technology, see ART
applications
 developing 62
 running 74
ART 2, 4, 14
 glossary entry 77
Artix
 bus 15
 contracts 17, 18, 45
 documentation 9
 features 5
 locator 19
 session manager 19
 transformer 19
Artix Designer
 projects 38, 41
 using 35–75
Artix Generator 62

B

BEA Tuxedo 4
binding
 glossary entry 77
bindings 17, 34, 55
bridge
 glossary entry 77
Bus
 glossary entry 77
bus 15

C

C/C++ Development Tools, see CDT
CDR 6
CDT 37, 41, 62, 70
 glossary entry 77
COBOL 36
code
 adding logic to 68
 generating 62
Common Data Representation, see CDR

connection
 glossary entry 77
contract
 glossary entry 77
contracts 17, 18, 45
CORBA 6, 38
 glossary entry 78
CORBA IDL 7, 36

D

data type, see type in glossary
deployment mode
 glossary entry 78
deployment phase 8
design phase 7
development phase 8

E

EAI 3
 glossary entry 78
Eclipse 8, 36, 37, 38, 39, 40, 41, 42, 43, 49, 62,
 66, 67, 70, 72, 74
 console view 72, 74, 75
 glossary entry 78
 help system 40
embedded mode
 glossary entry 78
endpoint
 glossary entry 78
endpoints 14
enterprise application integration, see EAI
enterprise service bus, See ESB
ESB 2

F

Field Manipulation Language, see FML
Fixed 6
fixed record length, see FRL
FML 6
FRL 6

G

G2 6
 generating code 62

H

host
 glossary entry 79
 HTTP 6

I

IDL 7
 IIOP 6

J

Java Development Tools, see JDT
 Java Messaging Service 6
 JDT 37, 41, 62, 66, 72
 glossary entry 79

L

locator 19

M

marshalling format
 glossary entry 79
 message
 glossary entry 79
 messages 17, 50
 MQSeries 6

O

operation
 glossary entry 79
 operations 17, 31

P

payload format
 glossary entry 79
 payload formats 6
 plug-ins 14
 ports 17
 port type
 glossary entry 80
 portTypes 17, 22, 31, 52
 protocol
 glossary entry 80

protocols 6

R

router
 glossary entry 80
 routing
 glossary entry 80

S

service 58
 glossary entry 80
 service bus
 glossary entry 80
 service-oriented architecture, see SOA
 services 17, 34
 session manager 19
 SOA 2
 SOAP 2, 6
 glossary entry 81
 standalone mode
 glossary entry 81
 switch
 glossary entry 81
 system
 glossary entry 81

T

TIBCO 6
 TibrvMsg 6
 transformer 19
 transport
 glossary entry 81
 transport plug-in
 glossary entry 81
 transports 6
 Tuxedo 6
 type
 glossary entry 81
 types 17, 46

V

VRL 6

W

W3C 22
 Web Services Description Language, see WSDL
 WebSphere MQ 4

World Wide Web Consortium, see W3C

WSDL 9, 17, 21–34

defined 22

glossary entry 82

WSDL files

adding elements to 45

creating 43

X

XML 6

glossary entry 82

XML Schema Definition, see XSD

XSD 25, 36

glossary entry 83

