



Artix™

Artix for J2EE

Version 4.0, March 2006

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.

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 © 1999-2006 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: 03-Apr-2006

Contents

List of Figures **vii**

Preface **ix**

Part I Introduction

Chapter 1 Introduction	1
J2EE Connector Architecture Overview	2
System-Level Contracts	4
Common Client Interface	5
Artix J2EE Connector Overview	6
Artix Servlet Container Support	9
Artix Concepts	10

Part II Using Artix in a J2EE Application Server

Chapter 2 Getting Started with Artix J2EE Connector	13
Introduction	14
Running the Hello World Demo on JBoss	15
Running the Hello World Demo on WebLogic	19
Running the Hello World Demo on WebSphere	23
Chapter 3 Exposing a Web Service to a J2EE Application	27
Introduction	28
Mapping the WSDL to Java	30
Writing your J2EE Application	31
Connection Management API Definition	32
Using the Connection Management API	33
Packaging your Application	37

Chapter 4 Exposing a J2EE Application as a Web Service	41
Introduction	42
Mapping the WSDL to Java	44
Implementing a Stateless Session Bean	45
Configuring Inbound Connections	47
Chapter 5 Deploying Artix J2EE Connector	51
Setting the Artix Environment	52
Deploying to JBoss	54
Deploying to WebLogic	57
Deploying to WebSphere	60
Chapter 6 Transactions	63
Transactions Overview	64
Local Transactions	67
Global Transactions	72
Outbound Global Transactions	73
Inbound Global Transactions	76
Chapter 7 Security	79
Outbound Security	80
Configuring Outbound Security	83
Credentials Mapping	84
Configuring Credentials Mapping in JBoss	86
Inbound Security	89
Configuring Inbound Security	92
Securing the Target EJB	93
Configuring JAAS Login Module	95
Configuring EJB Create Username and Password	97
Configuring a Secure Transport	99
Part III Using Artix in a Servlet Container	
Chapter 8 Exposing Artix Web Services from a Servlet Container	103
Introduction	104
Configuring Servlet Container to Run an Artix Application	107

Building an Artix Application	112
Mapping the WSDL to Java	113
Writing the Implementation Class	115
Developing an Artix Java Plug-in	116
Configuring Artix to Use Your Plug-in	120
Building and Deploying your Web Application	123
Part IV Reference Information	
Chapter 9 Artix J2EE Connector Configuration Properties	129
Configuration Properties	130
ArtixInstallDir	131
ArtixLicenseFile	132
LogLevel	133
ConfigurationDomain	134
ConfigurationScope	135
EJBServicePropertiesURL	136
EJBServicePropertiesPollInterval	137
MonitorEJBServiceProperties	138
JAASLoginConfigName	139
JAASLoginUserName	140
JAASLoginPassword	141
Setting Configuration Property Values	142
Setting Configuration Property Values in JBoss	143
Setting Configuration Property Values in WebLogic	144
Setting Configuration Property Values in WebSphere	145
Index	147

CONTENTS

List of Figures

Figure 1: J2EE Connector Architecture Component Structure	3
Figure 2: Connecting J2EE Applications to Web services using Artix J2EE Connector	7
Figure 3: Hello World Demo Running	18
Figure 4: Artix J2EE Connector Participating in Local Transactions	68
Figure 5: Artix J2EE Connector in an Outbound Global Transaction	73
Figure 6: Artix J2EE Connector in an Inbound Global Transaction	76
Figure 7: Artix J2EE Connector Propagating Credentials with Outbound Connections	80
Figure 8: Artix J2EE Connector Propagating Credentials with Inbound Connections	90
Figure 9: Exposing Artix Web Service from a Servlet Container	105
Figure 10: Classloader Configuration	110

LIST OF FIGURES

Preface

What is Covered in this Guide

This book describes how to use Artix in a J2EE application server environment and how to use Artix in a servlet container environment.

Who Should Read this Guide

This guide is aimed at J2EE application programmers who want to use Artix to develop and deploy distributed J2EE applications that are Web service enabled.

To use the Artix for J2EE guide, although you do not need an in depth knowledge of Artix concepts, WSDL and Web services, you do need to be familiar with these topics. The following guides are a good place to start if you are not already familiar with Artix concepts, WSDL and Web Services:

- Getting Started with Artix
- Designing Artix Solutions

In addition, the following may provide useful background information:

- *Understanding Web Services: XML, WSDL, SOAP, and UDDI*, written by Eric Newcomer, published by Addison Wesley, ISBN 0-201-75081-3.
- *Understanding SOA with Web Services*, written by Eric Newcomer and Greg Lomow, published by Addison Wesley, ISBN 0-321-18086-0.
- The W3C XML Schema page at: www.w3.org/XML/Schema.
- The W3C WSDL specification at: www.w3.org/TR/wsdl.

Organization of this Guide

This guide is divided into the following parts:

- **Part I, Introduction**, which gives an overview of the J2EE Connector Architecture, the Artix J2EE Connector, and the Artix servlet container support.
- **Part II, Using Artix in a J2EE Application Server**, which describes:
 - i. Getting started with the Artix J2EE Connector by running a simple demo.
 - ii. Exposing a Web service to a J2EE application
 - iii. Exposing a J2EE application as a Web service
 - iv. Deploying Artix J2EE Connector
 - v. Using transactions with the Artix J2EE Connector
 - vi. Artix J2EE Connector security
- **Part III, Using Artix in a Servlet Container**, which describes how to expose Artix Web services from a servlet container environment.
- **Part IV, Reference Information**, which provides details of the configuration properties supported by the Artix J2EE Connector.
- **Glossary of Terms**, which explains the terminology used in this book.
- **Index**

The Artix Library

The Artix documentation library is organized in the following sections:

- [Getting Started](#)
- [Designing and Developing Artix Solutions](#)
- [Configuring and Deploying Artix Solutions](#)
- [Using Artix Services](#)
- [Integrating Artix Solutions](#)
- [Integrating with Enterprise Management Systems](#)
- [Reference Documentation](#)

Getting Started

The books in this section provide you with a background for working with Artix. They describe many of the concepts and technologies used by Artix. They include:

- [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.
- [Using Artix Designer](#) describes how to use Artix Designer to build Artix solutions.
- [Artix Technical Use Cases](#) provides a number of step-by-step examples of building common Artix solutions.

Designing and Developing Artix Solutions

The books in this section go into greater depth about using Artix to solve real-world problems. They describe how Artix uses WSDL to define services, and how to use the Artix APIs to build new services. They include:

- [Building Service-Oriented Architectures with Artix](#) provides an overview of service-oriented architectures and describes how they can be implemented using Artix.
- [Understanding Artix Contracts](#) describes the components of an Artix contract. Special attention is paid to the WSDL extensions used to define Artix-specific payload formats and transports.
- [Developing Artix Applications in C++](#) discusses the technical aspects of programming applications using the C++ API.
- [Developing Advanced Artix Plug-ins in C++](#) discusses the technical aspects of implementing advanced plug-ins (for example, interceptors) using the C++ API.
- [Developing Artix Applications in Java](#) discusses the technical aspects of programming applications using the Java API.

Configuring and Deploying Artix Solutions

This section includes:

- [Configuring and Deploying Artix Solutions](#) discusses how to configure and deploy Artix-enabled systems, and provides examples of typical use cases.

Using Artix Services

The books in this section describe how to use the services provided with Artix:

- [Artix Locator Guide](#) discusses how to use the Artix locator.

- [Artix Session Manager Guide](#) discusses how to use the Artix session manager.
- [Artix Transactions Guide, C++](#) explains how to enable Artix C++ applications to participate in transacted operations.
- [Artix Transactions Guide, Java](#) explains how to enable Artix Java applications to participate in transacted operations.
- [Artix Security Guide](#) explains how to use the security features of Artix.

Integrating Artix Solutions

The books in this section describe how to integrate Artix solutions with other middleware technologies:

- [Artix for CORBA](#) provides information on using Artix in a CORBA environment.
- [Artix for J2EE](#) provides information on using Artix to integrate with J2EE applications.

For details on integrating with Microsoft's .NET technology, see the documentation for Artix Connect.

Integrating with Enterprise Management Systems

The books in this section describe how to integrate Artix solutions with a range of enterprise management systems. They include:

- [IBM Tivoli Integration Guide](#) explains how to integrate Artix with IBM Tivoli.
- [BMC Patrol Integration Guide](#) explains how to integrate Artix with BMC Patrol.
- [CA WSDM Integration Guide](#) explains how to integrate Artix with CA WSDM.

Reference Documentation

These books provide detailed reference information about specific Artix APIs, WSDL extensions, configuration variables, command-line tools, and terminology. The reference documentation includes:

- [Artix Command Line Reference](#)
- [Artix Configuration Reference](#)
- [Artix WSDL Extension Reference](#)
- [Artix Java API Reference](#)
- [Artix C++ API Reference](#)

- [Artix .NET API Reference](#)
- [Artix Glossary](#)

Getting 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/4.0/index.xml>

You can also search within a particular book. To search within a 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.

Artix Online Help

Artix Designer and the Artix Management Console include comprehensive online help, providing:

- Step-by-step instructions on how to perform important tasks
- A full search feature
- Context-sensitive help for each screen

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

- Select **Help | Help Contents** from the menu bar. Sections on Artix Designer and the Artix Management Console appear in the contents panel of the Eclipse help browser.
- Press **F1** for context-sensitive help.

In addition, there are a number of cheat sheets that guide you through the most important functionality in Artix Designer. To access these, select **Help | Cheat Sheets**.

Artix Glossary

The [Artix Glossary](#) provides a comprehensive reference of Artix terminology. It provides quick definitions of the main Artix components and concepts. All terms are defined in the context of the development and deployment of Web services using Artix.

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:

<i>Fixed width</i>	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 <code>IT_Bus::AnyType</code> class.
	Constant width paragraphs represent code examples or information a system displays on the screen. For example:
	<pre>#include <stdio.h></pre>
<i>Fixed width italic</i>	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:
	<pre>% cd /users/<i>YourUserName</i></pre>
<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

Part I

Introduction

In this part

This part contains the following chapter:

Introduction	page 1
------------------------------	------------------------

Introduction

Artix can be used in a J2EE application server environment and a servlet container environment. Using the Artix J2EE Connector, developers can easily connect their J2EE applications to Artix Web services and expose their J2EE applications as Artix Web services from within their chosen J2EE application server. In addition, Artix Web services can be exposed from servlet container. This chapter introduces the Artix J2EE Connector and the J2EE Connector Architecture on which it is implemented. This chapter also introduces Artix servlet container support and points you to resources that explain Artix concepts, WSDL and Web services.

In this chapter

This chapter discusses the following topics:

J2EE Connector Architecture Overview	page 2
Artix J2EE Connector Overview	page 6
Artix Servlet Container Support	page 9
Artix Concepts	page 10

J2EE Connector Architecture Overview

Overview

The J2EE Connector Architecture is part of the Java 2 Platform, Enterprise Editions (J2EE) 1.3 specification. It outlines a standard architecture for enabling J2EE applications to access resources in diverse Enterprise Information Systems (EISs). The goal is to standardize access to non-relational resources in the same way the JDBC API standardizes access to relational data.

The J2EE Connector Architecture is implemented in a J2EE application server and an EIS-specific resource adapter. The EIS resource adapter plugs into the J2EE application server and provides a system library specific to, and connectivity to, that EIS.

In this section

This section introduces the J2EE Connector Architecture. The following topics are covered:

- [Graphical representation](#)
- [System-Level Contracts](#)
- [Common Client Interface](#)

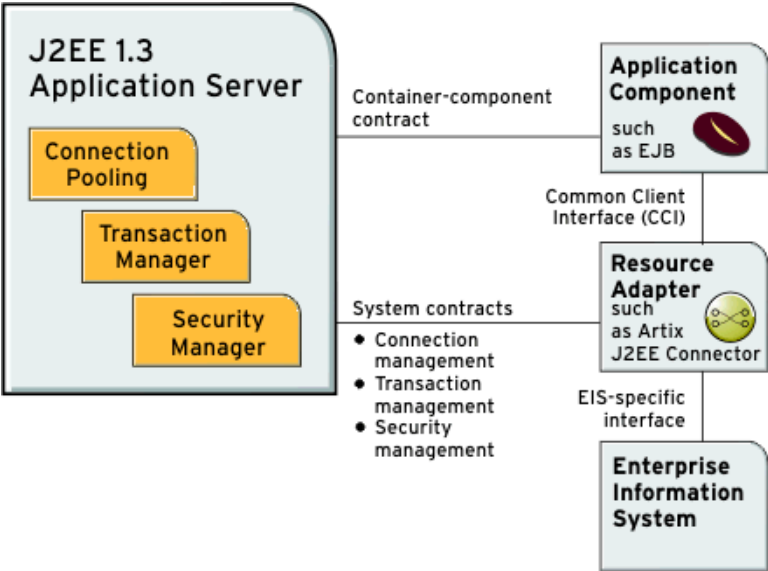
More information

For more information on the J2EE Connector Architecture and to view the specification itself, visit [Sun Microsystems' website](http://java.sun.com) (<http://java.sun.com>).

Graphical representation

Figure 1 shows the components defined by the J2EE Connector Architecture.

Figure 1: J2EE Connector Architecture Component Structure



System-Level Contracts

Overview

The J2EE Connector Architecture defines system-level contracts that are implemented by the J2EE application server and the EIS resource adapter. The following system-level contracts are specified in version 1.0 of the J2EE Connector Architecture:

- [Connection management](#)
 - [Transaction management](#)
 - [Security management](#)
-

Connection management

The connection management contract provides a consistent application programming model for enabling a J2EE application to connect to an EIS, and for allowing a J2EE application server to pool such connections. It facilitates a scalable and efficient environment that can support a large number of components requiring access to an EIS.

Transaction management

The transaction management contract defines the scope of transactional integration between a J2EE application server and an EIS that supports transactional access. It defines three levels of transaction support—no transactions, local transactions, and global or XA transactions.

Security management

The security management contract allows a J2EE application to access an EIS in a secure environment. This reduces security threats to the EIS and protects valuable information resources managed by the EIS. Mechanisms that can be used to protect an EIS against security threats include:

- Identification and authentication of principals (human users) to verify that they are who they say they are.
- Authorization and access control to determine whether a principal is allowed to access the EIS.
- Transport-level security to protect communications between the J2EE application server and the EIS.

Common Client Interface

Overview

The Common Client Interface (CCI) defines a common application programming model to allow application components and tools to interact with resource adapters. It is independent of any specific EIS. It is a low-level API and is similar to other J2EE interfaces such as the Java Database Connectivity (JDBC) interface.

Artix J2EE Connector Overview

Overview

The Artix J2EE Connector is a J2EE Connector Architecture resource adapter. It enables you to expose Artix Web services to your J2EE applications and allows you to expose your J2EE applications as Artix Web services.

The term Web services is used here to include SOAP over HTTP based services and any service that has been exposed as a Web service by Artix. Artix uses Web Services Definition Language (WSDL) contracts to expose services. The Artix J2EE Connector can use the Artix WSDL files to transparently connect your J2EE applications over multiple transports to any Artix-enabled back-end service. This includes HTTP, CORBA, IIOP, IBM WebSphere MQ, Java Messaging Service (JMS), BEA Tuxedo, and TIBCO Rendezvous.

To use the Artix J2EE Connector you do not need an in depth knowledge of Artix, WSDL or Web services. However, it would help if you were familiar with Artix and its approach to Web services. The guides listed under the [Getting Started](#) subsection of the Preface are a good place to start.

In this section

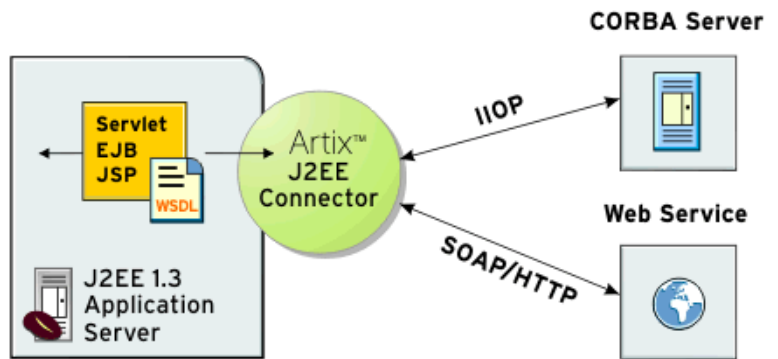
This section provides a high-level overview of Artix J2EE Connector's components and how it can be used to manage both outbound and inbound Web service connections, security and transactions for your J2EE applications. The following topics are covered:

- [Graphical representation](#)
- [Artix J2EE Connector RAR file](#)
- [Artix J2EE Connector deployment descriptor file](#)
- [Connection management](#)
- [Security management](#)

Graphical representation

Figure 2 illustrates at a high-level how the Artix J2EE Connector can be used to expose a Web service to a J2EE application. It acts as a bridge between J2EE and SOAP over HTTP Web services. This is the simplest example. It also illustrates that the Artix J2EE Connector can be used as a bridge between J2EE and a CORBA server that has been exposed as a Web service by Artix.

Figure 2: *Connecting J2EE Applications to Web services using Artix J2EE Connector*



Artix J2EE Connector RAR file

The Artix J2EE Connector resource adapter is packaged as a standard J2EE Connector Architecture resource adapter archive (RAR) file, `artix.rar`. The `artix.rar` file contains all the classes that Artix J2EE Connector needs to manage the connections between J2EE applications and Artix Web services. The Artix J2EE Connector uses the Java Native Interface (JNI) to access core Artix functionality. The relevant native code libraries are accessed from the Artix installation as needed at runtime.

Artix J2EE Connector deployment descriptor file

The Artix J2EE Connector's deployment descriptor file, `ra.xml`, contains information about Artix J2EE Connector's resource implementation, configuration properties, transaction and security support. It describes the capabilities of the resource adapter and provides a deployer with enough information to properly configure the resource adapter in an application server environment. An application server relies on the information in the

deployment descriptor to know how to interact properly with the resource adapter. The deployment descriptor is contained in the Artix J2EE Connector RAR file, `artix.rar`.

You should not change the settings in the Artix J2EE Connector deployment descriptor file. When deploying the Artix J2EE Connector, you can set the configuration properties to suit your environment using your J2EE application server's deployment tools. The configuration property values for your environment are not stored in the read-only deployment descriptor, `ra.xml`. Instead, your application server stores them separately in its own copy or representation of the deployment descriptor. The application server configured deployment descriptor properties override the entries in the `ra.xml` file.

Connection management

The Artix J2EE Connector manages both outbound and inbound Artix Web service connections. To run a simple demo, see [“Getting Started with Artix J2EE Connector” on page 13](#).

For more information on how to use the Artix J2EE Connector to manage outbound connections, see [“Exposing a Web Service to a J2EE Application” on page 27](#).

For more information on how to use the Artix J2EE Connector to manage inbound connections, see [“Exposing a J2EE Application as a Web Service” on page 41](#).

Security management

The Artix J2EE Connector supports credentials propagation. It propagates username and password credentials with outbound and inbound Artix Web service requests.

For more information on using security with the Artix J2EE Connector, see [“Security” on page 79](#).

Transaction management

The Artix J2EE Connector supports local and global (XA) transactions, as specified by the J2EE Connector Architecture.

For more information on using transactions with the Artix J2EE Connector, see [“Transactions” on page 63](#).

Artix Servlet Container Support

Overview

You can expose Artix Web services from a servlet container. You can expose Artix Web services from a servlet container. Artix provides the servlet component of the Web service. It provides a basic servlet, the `ArtixServlet.class`, and a servlet transport plug-in, which you can use to route HTTP requests to the servlet to Artix. You must write the Web service implementation class and generate an Artix Java plug-in. The Artix Java plug-in is required to create an instance of your Web service implementation and register it with the Artix bus.

Client applications use the information in the WSDL file to initialize a proxy to the Web service. Client applications can invoke on the Web services through the HTTP port assigned to the servlet container or using any of the transports supported by Artix.

More information

For more information on how to expose Artix Web services from a servlet container, see [“Exposing Artix Web Services from a Servlet Container”](#) on [page 103](#).

Artix Concepts

Overview

To use Artix in a J2EE application server or a servlet container environment, you do not need an in depth knowledge of Artix concepts, WSDL and Web services. In fact, for a simple application, everything that you need to get up and running is provided in this guide. However, if you are developing a complex application, you may need to be more familiar with Artix concepts, WSDL, Web services, and Service Oriented Architectures (SOA). The following will help provide you with the background information that you need:

- [Other Artix guides](#)
 - [Artix glossary](#)
 - [Other resources](#)
-

Other Artix guides

The Artix guides listed in Preface under “[Getting Started](#)” on page x and “[Designing and Developing Artix Solutions](#)” on page xi contain useful introductory material on Artix technology, WSDL, Web services and SOA.

Artix glossary

The Artix library also includes a comprehensive [glossary](#) that explains the terminology used in this guide and in the rest of the Artix guides.

Other resources

The following also provide useful background information:

- *Understanding Web Services: XML, WSDL, SOAP, and UDDI*, written by Eric Newcomer, published by Addison Wesley, ISBN 0-201-75081-3.
- *Understanding SOA with Web Services*, written by Eric Newcomer and Greg Lomow, published by Addison Wesley, ISBN 0-321-18086-0.
- The W3C XML Schema page at: www.w3.org/XML/Schema.
- The W3C WSDL specification at: www.w3.org/TR/wsdl.

Part II

Using Artix in a J2EE Application Server

In this part

This part contains the following chapters:

Getting Started with Artix J2EE Connector	page 13
Exposing a Web Service to a J2EE Application	page 27
Exposing a J2EE Application as a Web Service	page 41
Deploying Artix J2EE Connector	page 51
Transactions	page 63
Security	page 79

Getting Started with Artix J2EE Connector

This chapter focuses on getting started with the Artix J2EE Connector. It walks you through a simple Hello World demo that shows you how to use the Artix J2EE Connector to connect a servlet, which is deployed in a J2EE application server, to a SOAP over HTTP Web service. JBoss, WebLogic, and WebSphere are used as example J2EE application servers.

In this chapter

This chapter contains the following sections:

Introduction	page 14
Running the Hello World Demo on JBoss	page 15
Running the Hello World Demo on WebLogic	page 19
Running the Hello World Demo on WebSphere	page 23

Introduction

Overview

This chapter is based on running the Artix J2EE `Hello World` demo. It shows how you use the Artix J2EE Connector to connect a servlet deployed in a J2EE application server to a SOAP over HTTP Artix Web service.

Demo location

The demo can be found in:

```
ArtixInstallDir/artix/Version/demos/j2ee/hello_world_soap_http
```

WSDL file location

The Artix Web service WSDL file, `hello_world.wsdl`, used to build both the client J2EE application and the Artix server for this demo can be found in:

```
ArtixInstallDir/artix/Version/demos/basic/hello_world_soap_http/  
etc
```

Running the Hello World Demo on JBoss

Overview

To run the `Hello World` demo on JBoss, complete the following steps:

Step	Action
1	Set Artix environment
2	Start the JBoss server
3	Deploy the Artix J2EE Connector to JBoss
4	Configure the connection factory
5	Build the demo
6	Deploy the Hello World application to JBoss
7	Start the back-end Artix server
8	Run the Hello World demo

Set Artix environment

You must set the Artix environment before running JBoss or building the demo. See [“Setting the Artix Environment” on page 52](#) for more detail.

Start the JBoss server

Start the JBoss server by running the following command from your `JBossHome/bin` directory:

(Windows) `run.bat`

(UNIX) `run.sh`

Deploy the Artix J2EE Connector to JBoss

To deploy the Artix J2EE Connector to JBoss, copy the Artix J2EE Connector RAR file, `artix.rar`, from your

```
ArtixInstallDir/lib/artix/j2ee/4.0
```

directory, to your JBoss deployment directory:

```
JBossHome/server/default/deploy
```

Configure the connection factory

Connection factory configuration details are contained in the JBoss-specific Artix J2EE Connector deployment descriptor file, *CFactoryName-ds.xml* file. This demo provides a deployment descriptor for use with JBoss 4. To configure the connection factory, copy the *artixj2ee_1_5-ds.xml* file from your

```
ArtixInstallDir/artix/Version/demos/j2ee  
/hello_world_soap_http/etc
```

directory, to your JBoss deployment directory:

```
JBossHome/server/default/deploy
```

Build the demo

Build the `Hello World` demo from the *ArtixInstallDir/artix/Version/demos/j2ee/hello_world_soap_http* directory by running the following command:

```
(Windows) > ant
```

```
(Unix) % ant
```

The `ant` utility is a Java-based build tool. It is bundled with Artix. The `build.xml` file located in the demo directory contains the instructions for building the `Hello World` application, in an XML format that is understood by the `ant` utility. For more information about `ant`, see <http://ant.apache.org/>.

Deploy the Hello World application to JBoss

To deploy the Hello World application to JBoss, copy the Hello World application WAR file, `helloworld.war`, from your

```
ArtixInstallDir/artix/Version/demos/j2ee  
/hello_world_soap_http/j2ee_archives
```

directory, to your JBoss deployment directory:

```
JBossHome/server/default/deploy
```

Start the back-end Artix server

You can use either the Artix Java server or the Artix C++ server from the Artix basic Hello World demo as the back-end server in this example. It is located in:

```
ArtixInstallDir/artix/Version/demos/basic  
/hello_world_soap_http
```

In either case, you must compile the server before you can start it. For more information on how to compile and start the back-end Artix server, see the `README.txt` file located in the `basic/hello_world_soap_http` demo directory.

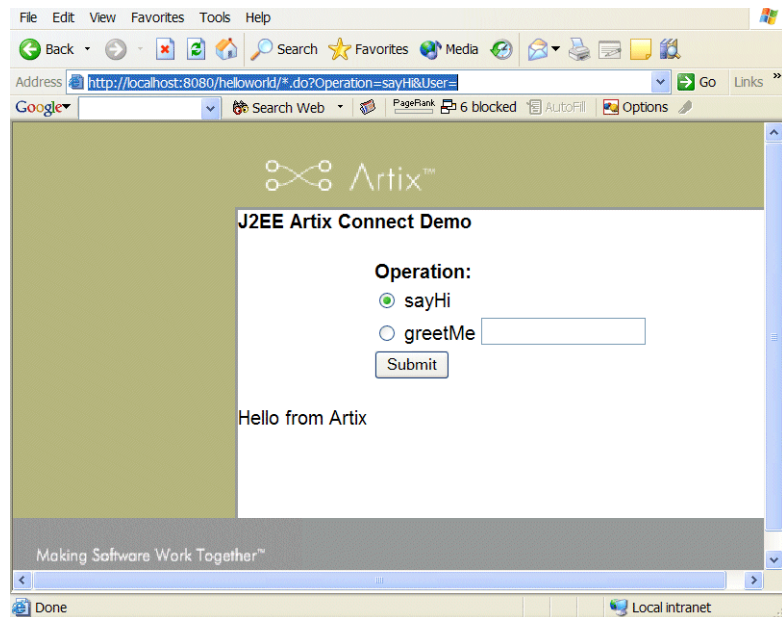
Run the Hello World demo

The Hello World demo presents a servlet view of the Hello World Web service. If JBoss is running under its default URI, and assuming that the application server is running on the same machine as the web browser, the servlet is available on JBoss at the following URI:

<http://localhost:8080/helloworld/rundemo.do>

The Hello World demo is displayed as shown in [Figure 3](#):

Figure 3: *Hello World Demo Running*



Running the Hello World Demo on WebLogic

Overview

To run the `Hello World` demo on WebLogic, complete the following steps:

Step	Action
1	Add Artix J2EE Connector API JAR to WebLogic's classpath
2	Set Artix environment
3	Start the WebLogic server
4	Configure the connection factory
5	Deploy Artix J2EE Connector to WebLogic
6	Build the demo
7	Deploy the Hello World application to WebLogic
8	Start the back-end Artix server
9	Run the Hello World demo

Add Artix J2EE Connector API JAR to WebLogic's classpath

WebLogic uses independent classloaders for each connection factory. The Artix J2EE Connector's API classes must be available to the application's classloader and to the resource adapter's classloader. This can lead to the problem of sharing classes across classloaders.

To prevent such class sharing problems, place the shared API classes on WebLogic's `CLASSPATH`. You can do this by appending the Artix J2EE Connector API JAR file, `artixj2ee.jar`, to WebLogic's `CLASSPATH` or to your global `CLASSPATH` environment variable. The `artixj2ee.jar` file is located in:

```
ArtixInstallDir/lib/artix/j2ee/Version
```

Alternatively, you can update WebLogic's start scripts. See the WebLogic documentation for details.

Set Artix environment

You must set the Artix environment before running WebLogic or building the demo. See “[Setting the Artix Environment](#)” on page 52 for more detail.

Start the WebLogic server

Start the WebLogic server by running the following command from your *BEA_Home/user_projects/domains/mydomain* directory:

```
startWebLogic.cmd
```

Configure the connection factory

Connection factory details are contained in the WebLogic-specific Artix J2EE Connector deployment descriptor, *weblogic-ra.xml*. WebLogic expects to find this file in the Artix J2EE Connector’s RAR file, *artix.rar*. To configure the connection factory, you must add the *weblogic-ra.xml* file to the *artix.rar* file prior to deploying the RAR file. The `Hello World` demo build file, *build.xml*, references an `ant` target that does this for you. Run the `ant` target as follows from the

ArtixInstallDir/artix/Version/demos/j2ee/hello_world_soap_http directory:

```
(Windows) > ant prepare.rar.to.deploy
```

```
(UNIX) % ant prepare.rar.to.deploy
```

The `ant` utility is a Java-based build tool. It is bundled with Artix. The `prepare.rar.to.deploy` target makes a copy of the *artix.rar* file, extracts the contents, adds the *weblogic-ra.xml* file and rebuilds the RAR file. The rebuilt *artix.rar* file is placed in the *j2ee-archives* directory of the demo.

The `ant` target is defined in the *common.xml* file, which is located in the *ArtixInstallDir/artix/Version/demos* directory.

For more details about `ant`, see <http://ant.apache.org/>.

Deploy Artix J2EE Connector to WebLogic

To deploy the Artix J2EE Connector to WebLogic, copy the Artix J2EE Connector RAR file, *artix.rar*, from your

```
ArtixInstallDir/artix/Version/demos/j2ee/hello_world_soap_http/  
j2ee-archives
```

directory, to your WebLogic auto-deployment directory:

WebLogic 8

```
BEA_Home/user_projects/domains/mydomain/applications
```

WebLogic 9*BEA_Home/user_projects/domains/mydomain/autodeploy*

Note: If you are running WebLogic in production mode (with auto-deployment disabled), refer to the WebLogic documentation for instructions on deploying a J2EE Connector Architecture resource adapter or connector from the administration web console.

Build the demo

Build the `Hello World` demo from the `ArtixInstallDir/artix/Version/demos/j2ee/hello_world_soap_http` directory by running the following command:

(Windows) > ant

(Unix) % ant

Deploy the Hello World application to WebLogic

To deploy the `Hello World` application to WebLogic, copy the `Hello World` application WAR file, `helloworld.war`, from your

```
ArtixInstallDir/artix/Version/demos/j2ee
/hello_world_soap_http/j2ee_archives
```

directory, to your WebLogic auto-deployment directory:

WebLogic 8*BEA_Home/user_projects/domains/mydomain/applications***WebLogic 9***BEA_Home/user_projects/domains/mydomain/autodeploy***Start the back-end Artix server**

You can use either the Artix Java server or the Artix C++ server from the Artix basic `Hello World` demo as the back-end server in this example. It is located in:

```
ArtixInstallDir/artix/Version/demos/basic
/hello_world_soap_http
```

In either case, you must compile the server before you can start it. For more information on how to compile and start the back-end Artix server, see the `README.txt` file located in the `basic/hello_world_soap_http` demo directory.

Run the Hello World demo

The `HelloWorld` demo presents a servlet view of the `HelloWorld` Web service. The servlet is available on WebLogic's host in the `/helloworld` context. If WebLogic is running under its default URI, and assuming that the application server is running on the same machine as the web browser, the servlet is available on WebLogic at the following URI:

<http://localhost:7001/helloworld/rundemo.do>

The `HelloWorld` demo is displayed as shown in [Figure 3 on page 18](#).

Running the Hello World Demo on WebSphere

Overview

To run the `Hello World` demo on WebSphere, complete the following steps:

Step	Action
1	Set Artix environment
2	Start the WebSphere server
3	Deploy the Artix J2EE Connector to WebSphere
4	Build the demo
5	Deploy the Hello World application to WebSphere
6	Start the back-end Artix server
7	Run the Hello World demo

Set Artix environment

You must set the Artix environment before running WebSphere. See [“Setting the Artix Environment” on page 52](#) for more detail.

Start the WebSphere server

Start the WebSphere server by running the following command from your `WebSphereHome/bin` directory:

```
(Windows) startServer.bat server1
(UNIX) startServer.sh server1
```

Deploy the Artix J2EE Connector to WebSphere

To deploy the Artix J2EE Connector run the following Jython script, which deploys the Artix J2EE Connector and creates a connection factory:

On Windows:

```
WebSphereHome\bin\wsadmin.bat -lang jython -f
ArtixInstallDir\artix\Version\demos\j2ee\hello_world_soap_http\
etc\rardeploy.py
nodeName ArtixInstallDir\lib\artix\j2ee\Version\artix.rar
```

On UNIX:

```
WebSphereHome/bin/wsadmin.sh -lang jython -f
ArtixInstallDir/artix/Version/demos/j2ee/hello_world_soap_http/
etc/rardeploy.py
<nodeName> ArtixInstallDir/lib/artix/j2ee/Version/artix.rar
```

For more information on Jython, see www.jython.org.

Alternatively, you can use the WebSphere Administrative Console to deploy the Artix J2EE Connector. Please refer to the WebSphere documentation for details on how to deploy a J2EE Connector Architecture resource adapter.

Build the demo

Build the `Hello World` demo from the

`ArtixInstallDir/artix/Version/demos/j2ee/hello_world_soap_http` directory by running the following command:

```
(Windows) > ant
```

```
(Unix) % ant
```

The `ant` utility is a Java-based build tool. It is bundled with Artix. The `build.xml` file located in the demo directory contains the instructions for building the `Hello World` application, in an XML format that is understood by the `ant` utility. For more information about `ant`, see <http://ant.apache.org/>.

Deploy the Hello World application to WebSphere

To deploy the `Hello World` application, run the following Jython script:

On Windows:

```
WebSphereHome\bin\wsadmin.bat -lang jython -f
ArtixInstallDir\artix\Version\demos\j2ee\hello_world_soap_http\
etc\appdeploy.py
nodeName serverName ArtixInstallDir\artix\Version\demos\j2ee\
hello_world_soap_http\j2ee-archives\helloworld.war
```

On UNIX:

```
WebSphereHome/bin/wsadmin.sh -lang jython -f
ArtixInstallDir/artix/Version/demos/j2ee/hello_world_soap_http/
etc/appdeploy.py
nodeName serverName ArtixInstallDir/artix/Version/demos/j2ee/
hello_world_soap_http/j2ee-archives/helloworld.war
```

Alternatively, you can use the WebSphere Administrative Console to deploy the Hello World application. Please refer to the WebSphere documentation for details on how to deploy applications.

Start the back-end Artix server

You can use either the Artix Java server or the Artix C++ server from the Artix basic Hello World demo as the back-end server in this example. It is located in:

```
ArtixInstallDir/artix/Version/demos/basic  
/hello_world_soap_http
```

In either case, you must compile the server before you can start it. For more information on how to compile and start the back-end Artix server, see the README.txt file located in the basic/hello_world_soap_http demo directory.

Run the Hello World demo

The Hello World demo presents a servlet view of the Hello World Web service. The servlet is available on your WebSphere host in the helloworld context. If WebSphere is running under its default URI, and assuming that the application server is running on the same machine as the web browser, the servlet is available on WebSphere at the following URI:

<http://localhost:9080/helloworld/rundemo.do>

The Hello World demo is displayed as shown in [Figure 3 on page 18](#).

Exposing a Web Service to a J2EE Application

You can use the Artix J2EE Connector to connect your J2EE applications to Web services. This chapter walks you through the steps involved.

In this chapter

This chapter discusses the following topics:

Introduction	page 28
Mapping the WSDL to Java	page 30
Writing your J2EE Application	page 31
Packaging your Application	page 37

Introduction

Overview

This section outlines how you expose a Web service to your J2EE application using the Artix J2EE Connector. The following topics are covered:

- [Implementation steps](#)
- [How it works](#)
- [Demo](#)

Implementation steps

The following is a high-level view of the steps that you need to complete to connect your J2EE application to a Web service using the Artix J2EE Connector. It assumes that the Web service WSDL file already exists. If, however, you need to develop a WSDL file, please refer to the [Understanding Artix Contracts](#) guide.

Step	Action
1	Obtain a copy of, or details of the location of, the WSDL file for the Web service to which you want to connect.
2	Map the WSDL file to Java to obtain the Java interfaces that you will use when writing your application. Artix provides a <code>wsdltojava</code> command-line utility that does this for you. The Artix WSDL-to-Java mapping is based on the JAX-RPC standard.
3	Write your application.
4	Package your application.
5	Deploy the your application and the Artix J2EE Connector to your J2EE application server.

The rest of this chapter describes steps 1 to 4 in detail. For deployment details, see [“Deploying Artix J2EE Connector” on page 51](#).

How it works

The Artix J2EE Connector is provided with a Java JAX-RPC style interface that represents the Web service and the location of a WSDL file that describes the Web service. The `getConnection()` operation on the Artix J2EE Connector connection factory, returns a proxy that implements the Java JAX-RPC interface. When the application invokes an operation on the returned proxy, the Artix J2EE Connector uses the information in the corresponding WSDL file to determine the appropriate binding information for the Web service. The binding information describes the low-level details around access to the Web service, the protocol address and wire format. Typically this is SOAP over HTTP, but it can be fixed format over JMS, CDR over IIOP, or any one of the many transports that Artix supports. The Artix J2EE Connector uses Artix to invoke on the Web service using the appropriate binding.

In addition, the proxy supports a `close()` operation. This is used when the application is finished with the Web service. The `close()` operation returns the proxy to the application server's connection pool so it can be reused by other components.

Demo

The examples used in this chapter are taken from the J2EE Hello World demo, which can be found in:

```
ArtixInstallDir/artix/Version/demos/j2ee/hello_world_soap_http
```

If you want to run this demo, see [“Getting Started with Artix J2EE Connector” on page 13](#) or the `README.txt` file in the demo directory.

Mapping the WSDL to Java

Overview

The Artix development tools include a `wSDLtojava` command-line utility that you can use to generate Java interfaces from the WSDL file. Artix maps WSDL types to Java using the mapping described in the JAX-RPC specification.

Syntax of `wSDLtojava` command

To generate Java interfaces from a WSDL file, run the following command:

```
wSDLtojava -d [output_dir] -interface -p package wSDL_contract
```

The parameters shown above are defined as follows:

- `-d [output_dir]` Specifies the directory to which the generated code is written. The default is the current working directory.
- `-interface` Generates the Java interface for the service.
- `-p <[wSDL namespace =]
Package
Name>` Specifies the name of the Java package to use for the generated code. You can optionally map a WSDL namespace to a particular package name if your contract has more than one namespace.
- `wSDL_contract` Specifies the WSDL file from which the Java code is being generated.

Example

For example, the following `wSDLtojava` command was used to generate the `Greeter.java` interface class that is provided in the J2EE Hello World demo:

```
wSDLtojava -d src -interface -p demo.ejb hello_world.wSDL
```

The `hello_world.wSDL` file can be found in:

```
ArtixInstallDir/artix/Version/demos/basic/hello_world_soap_http/  
etc
```

More information

For more information on the `wSDLtojava` command-line utility, see the *Developing Artix Applications in Java* manual.

Writing your J2EE Application

Overview

The Artix J2EE Connector connection management API allows you to get a connection from your J2EE application to a Web service. The Artix J2EE Connector API usage pattern is consistent with general connection management in J2EE. This section provides an overview of the Artix J2EE Connector connection management interfaces and outlines typical usage scenarios.

In this section

This section covers the following topics:

Connection Management API Definition	page 32
Using the Connection Management API	page 33

Connection Management API Definition

Overview

The Artix J2EE Connector connection management API is packaged in `com.iona.connector` and consists of two interfaces—`ArtixConnectionFactory` and `Connection`. This subsection gives a brief description of each and points you to the Javadoc for more information. The following topics are covered:

- [ArtixConnectionFactory](#)
- [Connection](#)
- [Javadoc](#)

ArtixConnectionFactory

The `ArtixConnectionFactory` interface provides the methods to create a `Connection` that represents a Web service defined by the supplied parameters. The `ArtixConnectionFactory` interface is the type returned from an environment naming context lookup of the Artix J2EE Connector by a J2EE component.

Connection

The `Connection` interface provides a handle to a connection managed by the J2EE application server. It is the super interface of the Web service proxy returned by `ArtixConnectionFactory`. It allows the caller to return the proxy to the application server's pool when it is no longer needed. The returned proxy also implements the interface supplied as an argument to `getConnection()`.

Javadoc

For more detail on the Artix J2EE Connector API, see the [Artix Javadoc](#).

Using the Connection Management API

Overview

The Artix J2EE Connector `ArtixConnectionFactory` interface has several method signatures that you can use. This allows you to use the `ArtixConnectionFactory` interface in a way that best suits your environment. This subsection outlines the possible usage scenarios. The following topic are covered:

- [Hardcoding WSDL location details in your application](#)
- [Providing WSDL location details at runtime](#)
- [Omitting the port name parameter](#)
- [Configuring Artix to locate the WSDL at runtime](#)
- [Accessing the Artix bus directly](#)
- [More detail on Artix J2EE Connector API](#)

Hardcoding WSDL location details in your application

The following example code is taken from the `Hello World` demo used in “[Getting Started with Artix J2EE Connector](#)” on page 13. It had been simplified to make it easier to read. It demonstrates how the WSDL location details can be hardcoded in your application:

Example 1: *Hello World* servlet

```
Context ctx = new InitialContext();  
  
1 ArtixConnectionFactory factory =  
  (ArtixConnectionFactory)ctx.lookup("java:comp/env/eis/  
  ArtixConnector");  
  
2 URL wsdlLocation = getClass().getResource("/hello_world.wsdl");  
3 QName serviceName = new  
  QName("http://www.iona.com/hello_world_soap_http",  
  "SOAPService");  
4 QName portName = new QName("", "SoapPort");  
  
5 Greeter greeter = (Greeter)factory.getConnection(Greeter.class,  
  wsdlLocation, serviceName, portName);  
  
6 greeter.sayHi();  
  
7 ((Connection)greeter).close();
```

The code in [Example 1](#) can be explained as follows:

1. Retrieve the connection factory from JNDI.
2. Determine the WSDL location URL from the classpath using the JVM runtime. The WSDL file must be available on the classpath for this to work.
3. Create a `QName` that identifies which service in the WSDL file to use.
4. Create a `QName` that identifies which port in the WSDL file to use.
5. Invoke on the connection factory to create a connection to the Web service and return a proxy.
6. Invoke on the service.
7. Close the connection to the service and return to the application server connection pool.

Providing WSDL location details at runtime

The following example code shows the same code, but in this case the WSDL file is located by the runtime using Artix:

Example 2: *Hello World servlet*

```
Context ctx = new InitialContext();

1 ArtixConnectionFactory factory =
  (ArtixConnectionFactory)ctx.lookup("java:comp/env/eis/
  ArtixConnector");

2 QName serviceName = new
  QName("http://www.iona.com/hello_world_soap_http",
  "SOAPService");

3 QName portName = new QName("", "SoapPort");

4 Greeter greeter = (Greeter)factory.getConnection(Greeter.class,
  serviceName, portName);

5 greeter.sayHi();

6 ((Connection)greeter).close();
```

The code in [Example 2](#) can be explained as follows:

1. Retrieve the connection factory from JNDI.
2. Create a `QName` that identifies which service in the WSDL contract to use. This is used by the Artix runtime to locate the WSDL contract. See [Configuring Artix to locate the WSDL at runtime](#) for more detail.
3. Create a `QName` that identifies which port in the WSDL contract to use.
4. Invoke on the connection factory to create a connection to the Web service and return a proxy.
5. Invoke on the service.
6. Close the connection to the service.

Omitting the port name parameter

The `ArtixConnectionFactory` API also allows you to omit the port name parameter. You can drop the port name parameter if the WSDL file only defines one port or the first port defined in a WSDL file that has a number of port definitions is the port that you want to use.

Configuring Artix to locate the WSDL at runtime

There are several ways in which Artix can find WSDL files and endpoint references at runtime. For more detail, see the *Accessing Contracts and References* chapter in the [Configuring and Deploying Artix Solutions](#) guide.

Note: One way that Artix uses find WSDL files and endpoint references is the command line. This approach is not appropriate for the Artix J2EE Connector.

Accessing the Artix bus directly

If you need to access the Artix bus directly, you must use the `com.iona.connector.ArtixConnectionFactory.getBus()` method. For example, you might need to access the bus context registry or create a reference. [Example 3](#) shows how to use the `ArtixConnectionFactory.getBus()` method.

Example 3: Using `ArtixConnectionFactory.getBus()`

```

Context ctx = new InitialContext();
1 ArtixConnectionFactory factory =
  (ArtixConnectionFactory)ctx.lookup(EIS_JNDI_NAME);
2 Bus bus = (Bus)factory.getBus();
3 ContextRegistry registry = bus.getContextRegistry();

```

The code shown in [Example 3](#) can be explained as follows:

1. Retrieve the connection factory from JNDI.
2. Cast the connection factory to `com.ionajbus.Bus`.
3. Call `getContextRegistry()` on the returned bus to get a reference to the context registry. The `com.ionajbus.ContextRegistry` object manages all of the context objects for the application.

For more information on message contexts, see the *Using Message Contexts* chapter in the [Developing Artix Applications in Java](#) guide.

Note: If you are using WebLogic, you must ensure that the bus, and any dependencies that it might have, are available to the classloader that loads the application. The easiest way to do this is add the Artix Java runtime JAR, `ArtixInstallDir/lib/artix/java_runtime/4.0/java_runtime-rt.jar`, to WebLogic's system classpath.

More detail on Artix J2EE Connector API

For more detail on the Artix J2EE Connector API, see the [Artix Javadoc](#).

Packaging your Application

Overview

When packaging and deploying your J2EE application you must declare the resource reference used in your code in your application deployment descriptor and map that resource reference to a resource. In addition, you need to package the Web service interface classes with your application.

In this section

This section describes the following:

- [Declaring the resource reference](#)
- [Mapping the resource reference](#)
- [Packaging Web service interface classes](#)

Note: The example deployment descriptors shown here are taken from the `Hello World` demo, which is used in [“Getting Started with Artix J2EE Connector”](#) on page 13.

Declaring the resource reference

You must declare the resource reference used in your code in your application deployment descriptor, `ejb-jar.xml` or `web.xml`, by adding a `resource-ref` tag. For example, in the `Hello World` demo, the `helloworld.war` file contains a `web.xml` file that includes the following:

```
<resource-ref>
  <res-ref-name>eis/ArtixConnector</res-ref-name>
  <res-type>com.iona.connector.ArtixConnectionFactory
</res-type>
  </res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

Mapping the resource reference

You must map the resource reference used in your code to the resource. How you do this is dependent on the application server that you are using. For example, if you are using JBoss, you must add a `resource-ref` tag to

the application server deployment descriptor file, `jboss.xml`. For example, in the Hello World demo, the `helloworld.war` file contains a `jboss-web.xml` file that includes the following:

```
<jboss-web>
  <resource-ref>
    <res-ref-name>eis/ArtixConnector</res-ref-name>
    <res-type>com.iona.connector.ArtixConnectionFactory</res-type>
    <jndi-name>java:/ArtixConnector</jndi-name>
  </resource-ref>
</jboss-web>
```

The `jndi-name` of the `resource-ref` element binds the resource reference to the connection factory that has been previously declared.

Similarly, if you are using WebLogic, you need to add `reference-descriptor` tag to the application server deployment file, `weblogic.xml`. For example, in the Hello World demo, the `helloworld.war` file contains a `weblogic.xml` file that includes the following:

```
<weblogic-web-app>
  <reference-descriptor>
    <resource-description>
      <res-ref-name>eis/ArtixConnector</res-ref-name>
      <jndi-name>ArtixConnector</jndi-name>
    </resource-description>
  </reference-descriptor>
</weblogic-web-app>
```

If you are using WebSphere, you can use the WebSphere Administrative Console to map the resource reference to the resource while deploying the Artix J2EE Connector. Please refer to the WebSphere documentation for details.

Packaging Web service interface classes

You must package the interface classes that you generated from the Web service WSDL file with your J2EE application module when you are packaging and deploying it. If the WSDL file contains complex types, the `wsdltojava` utility will also produce helper classes. These also need to be packaged with your J2EE application module.

It is important to package these files in the appropriate location in your J2EE application module. For example, the `helloworld.war` file deployed in the Hello World demo described in [“Getting Started with Artix J2EE Connector” on page 13](#), the interface classes are packaged in the `WEB-INF/classes` directory.

More information

Please refer to the J2EE specification and your J2EE vendor documentation for more information on application packaging and deployment.

Exposing a J2EE Application as a Web Service

You can expose your J2EE application as a Web service using the Artix J2EE Connector.

In this chapter

This chapter discusses the following topics:

Introduction	page 42
Mapping the WSDL to Java	page 44
Implementing a Stateless Session Bean	page 45
Configuring Inbound Connections	page 47

Introduction

Overview

This section outlines how you expose a J2EE application as a Web service using the Artix J2EE Connector. The following topics are covered:

- [Implementation steps](#)
- [How it works](#)
- [Demo](#)

Implementation steps

The following is a high-level view of the steps that you need to complete to expose your J2EE application as a Web service using the Artix J2EE Connector. It assumes that the Web service WSDL file already exists. If, however, you need to develop a WSDL file, please refer to the [Understanding Artix Contracts](#) guide.

Step	Action
1	Obtain a copy of, or details of the location of, the WSDL file that defines the Web service that your application will implement.
2	Map the WSDL file to Java to obtain the Java interfaces that you will use when writing your application. Artix provides a <code>wsdltojava</code> command-line utility that does this for you. The Artix WSDL-to-Java mapping is based on the JAX-RPC standard.
3	Implement a stateless session bean (SLSB) whose remote interface extends the JAX-RPC interface generated by the <code>wsdltojava</code> utility.
4	Configure the Artix J2EE Connector for inbound connections by using an <code>ejb_servants.properties</code> file.
5	Deploy the Artix J2EE Connector and your application to your J2EE application server.

The rest of this chapter describes steps 1 to 4 in more detail. For deployment information, see [“Deploying Artix J2EE Connector” on page 51](#).

How it works

Your J2EE application must provide an end point to which the Artix J2EE Connector can dispatch incoming requests. This endpoint is a stateless session bean (SLSB). The SLSB implements a method for each service/port operation defined in the WSDL contract. The signature for each method is as defined by the JAX-RPC mapping.

For each port, the Artix J2EE Connector creates a servant and registers it with the Artix bus. A servant is an object that implements the service/port operations specified in the WSDL file. The port is mapped to the SLSB by its JNDI name. Each servant is given a JNDI name for the SLSB home on which to receive the request. The port-to-JNDI mapping is specified in an external properties file, `ejb_servants.properties`.

On receiving a request, the servant resolves the SLSB home object from JNDI and creates an instance of the bean. The servant forwards the request to the SLSB and passes return types or exceptions to the Artix runtime and from there to the client.

Demo

The examples used in this chapter are taken from the `Inbound Connection` demo, which can be found in:

```
ArtixInstallDir/artix/Version/demos/j2ee/inbound_connection
```

If you want to run this demo, see the `README.txt` file in the demo directory.

Mapping the WSDL to Java

Overview

The Artix development tools include a `wSDLtojava` command-line utility that you can use to generate Java interfaces from the WSDL file. Artix maps WSDL types to Java using the mapping described in the JAX-RPC specification.

Syntax of `wSDLtojava` command

To generate Java interfaces from a WSDL file, run the following command:

```
wSDLtojava -d [output_dir] -interface -p package wSDL_contract
```

The parameters shown above are defined as follows:

- `-d [output_dir]` Specifies the directory to which the generated code is written. The default is the current working directory.
- `-interface` Generates the Java interface for the service.
- `-p <[wSDL namespace =]
Package
Name>` Specifies the name of the Java package to use for the generated code. You can optionally map a WSDL namespace to a particular package name if your contract has more than one namespace.
- `wSDL_contract` Specifies the WSDL file from which the Java code is being generated.

Example

For example, the following `wSDLtojava` command was used to generate the `Greeter.java` interface class that is provided in the `Inbound Connection demo`:

```
wSDLtojava -d src -interface -p demo.greeter hello_world.wSDL
```

The `hello_world.wSDL` file can be found in:

```
ArtixInstallDir/artix/Version/demos/basic/hello_world_soap_http/  
etc
```

More information

For more information on the `wSDLtojava` command-line utility, see the [Developing Artix Applications in Java](#) guide.

Implementing a Stateless Session Bean

Overview

You must implement a stateless session bean (SLSB) whose remote interface extends the interface that you generated from the WSDL file in the previous section. As per the EJB specification, the SLSB implementation must implement the methods defined in the remote interface. This section shows, as an example, the SLSB used in the `Inbound Connection` demo, including the:

- [Generated Java interface](#)
- [EJB remote interface definition](#)
- [Stateless Session Bean example](#)

Generated Java interface

The following example shows the Java interface, `Greeter`, which was generated from the `hello_world.wsdl` file in the `Inbound Connection` demo:

Example 4: *Greeter Interface*

```
public interface Greeter extends java.rmi.Remote {  
    public String sayHi() throws RemoteException;  
  
    public String greetMe(String me) throws RemoteException;  
}
```

EJB remote interface definition

The following EJB remote interface extends the `Greeter` interface:

Example 5: *Greeter Remote Interface*

```
...  
public interface GreeterRemote extends EJBObject, Greeter {  
}
```

Stateless Session Bean example

The following SLSB implements a method for each operation defined in the `hello_world.wsdl` file:

Example 6: *Greeter Stateless Session Bean*

```
...
public class GreeterBean implements SessionBean {
...
    public String sayHi() throws RemoteException {...
    }
    public String greetMe(String user) throws RemoteException
    {...
    }
    //rest of bean implementation goes here
}
```

Configuring Inbound Connections

Overview

The Artix J2EE Connector creates a servant for each port defined in the WSDL contract and registers it with the Artix bus. Each servant is given a JNDI name for the SLSB home on which to receive the request. You must configure the Artix J2EE Connector with the port-to-JNDI mapping so that it can pass incoming Web service requests to your application. To do this, you must create an `ejb_servants.properties` file that maps the port to the JNDI name.

In this section

This section describes the format of the `ejb_servants.properties` file, provides an example, and describes how to configure the Artix J2EE Connector to find and monitor your `ejb_servants.properties` file. The following topics are covered:

- [Format of `ejb_servants.properties`](#)
 - [Example](#)
 - [Multiple entries](#)
 - [Configuring the location and monitoring of `ejb_servant.properties`](#)
-

Format of `ejb_servants.properties`

The format of the `ejb_servants.properties` file is:

```
jndi_name={namespace}ServiceName,PortName@url_to_wsdl
```

<code>jndi_name</code>	The configured JNDI name of the bean. This is the JNDI name that an external client uses to contact the bean.
<code>ServiceName</code>	The string form of the <code>QName</code> for the Artix service in the WSDL file. The string form uses curly brackets for the namespace and a plain string for the local part. Artix listens on all configured ports for the service.
<code>PortName</code>	The string form for the port name defined in the WSDL file. This is an optional parameter and can be used to specify a particular port. If it is not specified, Artix listens on all ports.

`@url_to_wsdl` The string form of a URL that identifies the WSDL file. This is an optional parameter and does not need to be used if Artix runtime has been configured to locate the WSDL file (using the service `QName`).

For details on how to configure Artix to locate the WSDL contract at runtime, see the *Accessing Contracts and References* chapter in the [Configuring and Deploying Artix Solutions](#).

Example

Artix includes a `ejb_servants.properties` file that you can use as a template for your application. It is located in:

```
ArtixInstallDir/artix/Version/etc
```

The following shows the entry that is added to the `ejb_servants.properties` file for the `Inbound Connection demo`:

```
GreeterBean={http://www.iona.com/hello_world_soap_http}SOAPService@file:C:/IONA/artix/4.0 \
demos/j2ee/inbound_connection/wsdl/hello_world.wsdl
```

Note: The contents must appear on one line.

Multiple entries

You can include more than one entry in an `ejb_servants.properties` file if, for example, you want to deploy multiple J2EE applications as Web services targets.

Configuring the location and monitoring of `ejb_servant.properties`

By default, the Artix J2EE Connector is configured to find the `ejb_servants.properties` file in:

```
ArtixInstallDir/artix/Version/etc
```

If you store your `ejb_servants.properties` file in a different location, you must set the `EJBServicePropertiesURL` configuration property to specify that location. See [“EJBServicePropertiesURL” on page 136](#) for details.

In addition, by default, the Artix J2EE Connector is configured to check the `ejb_servants.properties` file for updates at 30 second intervals. This behavior can be altered by changing the default settings of the

`MonitorEJBServiceProperties` and `EJBServicePropertiesPollInterval` configuration properties. See [“MonitorEJBServiceProperties” on page 138](#) and [“EJBServicePropertiesPollInterval” on page 137](#) for more detail.

Deploying Artix J2EE Connector

How you deploy the Artix J2EE Connector is dependent on the J2EE application server that you are using. In all cases, however, you must set the Artix environment before running your application server. This chapter outlines how to do this and highlights some important points when deploying to JBoss, WebLogic and WebSphere.

For more detailed deployment information, please refer to your J2EE application server documentation.

In this chapter

This chapter discusses the following topics:

Setting the Artix Environment	page 52
Deploying to JBoss	page 54
Deploying to WebLogic	page 57
Deploying to WebSphere	page 60

Setting the Artix Environment

Overview

The Artix shared libraries must be available to the Artix J2EE Connector. To set the Artix environment, you must do either run the `artix_env` script or append the Artix shared library directory to the system environment variable.

Run the `artix_env` script

Run the `artix_env` script located in your `ArtixInstallDir/artix/Version/bin` directory.

For more information on the `artix_env` script, see the getting started chapter in the [Configuring and Deploying Artix Solutions](#) guide.

Append Artix shared library directory to system environment variable

If you do not want to run the `artix_env` script, you must instead append the Artix shared library directory to your system environment variable as follows:

Windows

```
set PATH=%PATH%;ArtixInstallDir\bin
```

UNIX

```
LD_LIBRARY_PATH=ArtixInstallDir/shlib:ArtixInstallDir/shlib/  
default:$LD_LIBRARY_PATH
```

On HP-UX set `SHLIB_PATH` as follows:

```
SHLIB_PATH=ArtixInstallDir/shlib:ArtixInstallDir/shlib/  
default:$SHLIB_PATH
```

Artix J2EE Connector classloader firewall

The Artix J2EE Connector uses a classloader firewall to isolate classes used by Artix and classes used by the application server. If, for example, the application server requires Xerces 2.4 and Artix requires Xerces 2.5, the classloader firewall allows both versions to exist. You do not need to do anything with this classloader firewall. If, however, you are interested in finding out more about Artix classloader firewalls:

- The Artix J2EE Connector classloader firewall configuration file, `artix_j2ee_ce.xml`, is located in the following directory of your Artix installation:

```
InstallDir/artix/Version/etc
```

- See the *Things to Consider when Developing Artix Applications* chapter, in the [Developing Artix Applications in Java](#) guide.

Deploying to JBoss

Overview

This section gives an overview of how to deploy the Artix J2EE Connector to JBoss and points you to a demo that walks you through deployment and shows you a running application. It also provides you with an example of a JBoss-specific Artix J2EE Connector deployment descriptor file.

In addition, to enable JBoss to make the Artix J2EE Connector available to your application, you must include an entry in the application deployment descriptor that binds the resource reference to the resource. This section provides with an example of such an entry. The following topics are covered:

- [Deployment steps](#)
- [Run the Hello World demo](#)
- [Example CFactoryName-ds.xml deployment descriptor](#)
- [Example application-specific deployment descriptor](#)
- [More detail](#)

Deployment steps

To deploy the Artix J2EE Connector to JBoss, complete the following steps:

Step	Action
1	Set the Artix environment before running JBoss. See “Setting the Artix Environment” on page 52 for more detail.
2	Copy the Artix J2EE Connector’s RAR file, <code>artix.rar</code> , from the <code>ArtixInstallDir/lib/artix/j2ee/4.0</code> directory, to your JBoss deployment directory, typically: <code>JBossHome/server/default/deploy</code>
3	Copy a JBoss-specific Artix J2EE Connector deployment descriptor file, <code>CFactoryName-ds.xml</code> , to your JBoss deployment directory: <code>JBossHome/server/default/deploy</code> This file is required to configure the Artix J2EE Connector connection factories. For more details, see Example CFactoryName-ds.xml deployment descriptor .

Run the Hello World demo

To deploy the Artix J2EE Connector and an example application to JBoss, see [“Running the Hello World Demo on JBoss” on page 15](#).

Example CFactoryName-ds.xml deployment descriptor

The JBoss-specific Artix J2EE Connector deployment descriptor, `CFactoryName-ds.xml`, defines the connection factories associated with the Artix J2EE Connector, any dependencies it might have on other services, the JNDI name under which it is registered, and the value of the configuration properties that need to be defined for the connection factories.

The Artix J2EE Hello World demo provides an example of such a deployment descriptor, `artixj2ee_1_5-ds.xml`, for use with JBoss 4:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE connection-factories (View Source for full
doctype...)>
<connection-factories>
  <no-tx-connection-factory>
    <jndi-name>ArtixConnector</jndi-name>
    <rar-name>artix.rar</rar-name>
    <connection-definition>
      com.iona.connector.ArtixConnectionFactory
    </connection-definition>
  </no-tx-connection-factory>
</connection-factories>
```

Example application-specific deployment descriptor

JBoss also requires an application-specific deployment descriptor to bind the resource reference to the resource; that is, to make the Artix J2EE Connector available to the application.

The following example deployment descriptor, `jboss-web.xml`, is used in the Hello World demo to make the Artix J2EE Connector available to the Hello World application:

```
<jboss-web>
  <resource-ref>
    <res-ref-name>eis/ArtixConnector</res-ref-name>
    <res-type>com.iona.connector.ArtixConnectionFactory
    </res-type>
    <jndi-name>java:/ArtixConnector</jndi-name>
  </resource-ref>
</jboss-web>
```

The `jndi-name` of the `resource-ref` element binds the resource reference to the connection factory that has been previously deployed.

When deploying your application, copy it and an application-specific deployment descriptor file to your JBoss deployment directory:

JBossHome/server/default/deploy.

More detail

For more detailed deployment information, please refer to the JBoss documentation.

Deploying to WebLogic

Overview

This section gives an overview of how to deploy the Artix J2EE Connector to WebLogic and points you to a demo that walks you through deployment and shows you a running application. It also highlights how you can avoid having to duplicate the Artix J2EE Connector's API JAR when you are deploying the Artix J2EE Connector to WebLogic. The following topics are covered:

- [Assumption](#)
- [Class sharing between resource adapters and applications](#)
- [Deployment steps](#)
- [Configuring the connection factory](#)
- [Example weblogic-ra.xml](#)
- [Run the Hello World demo](#)
- [More information](#)

Assumption

The information presented in this section is based on the assumption that you are using WebLogic Server version 8.1 Service Pack 3 or higher.

Class sharing between resource adapters and applications

WebLogic uses independent classloaders for each connection factory. The Artix J2EE Connector's API classes must be available to the application's classloader and to the resource adapter's classloader. This can lead to the problem of sharing classes across classloaders.

To prevent such class sharing problems, place the shared API classes on WebLogic's `CLASSPATH`. You can do this by appending the Artix J2EE Connector API JAR file, `artixj2ee.jar`, to WebLogic's `CLASSPATH` or to your global `CLASSPATH` environment variable. The `artixj2ee.jar` file is located in:

```
ArtixInstallDir/lib/artix/j2ee/4.0
```

Deployment steps

To deploy the Artix J2EE Connector to WebLogic, complete the following steps:

1. Set the Artix environment before running WebLogic. See [“Setting the Artix Environment” on page 52](#) for more detail.
2. Configure the connection factory. See [Configuring the connection factory](#) for more detail.
3. Deploy Artix J2EE Connector to WebLogic by copying the `artix.rar` file that you configured in step 2, to your WebLogic auto-deployment directory:

WebLogic 8

```
BEA_Home/user_projects/domains/mydomain/applications
```

WebLogic 9

```
BEA_Home/user_projects/domains/mydomain/autodeploy
```

Configuring the connection factory

Connection factory details are contained in the WebLogic-specific Artix J2EE Connector deployment descriptor, `weblogic-ra.xml`. WebLogic expects to find this file in the Artix J2EE Connector’s RAR file, `artix.rar`. To configure the connection factory, you must add the `weblogic-ra.xml` file to the `artix.rar` file prior to deploying the RAR file. To do this:

1. Make a copy of the `artix.rar` file and place it in the directory that contains the `META-INF\weblogic-ra.xml` subdirectory. The `artix.rar` file is located in:

```
ArtixInstallDir/lib/artix/j2ee/4.0
```

2. Run the following JAR utility from the directory to which you have copied the `artix.rar` file:

```
jar uvf artix.rar META-INF/weblogic-ra.xml
```

The `jar uvf` utility extracts the contents of the `artix.rar` file, adds the `weblogic-ra.xml` file to the `META-INF` directory of the archive file and rebuilds `artix.rar`.

Example weblogic-ra.xml

The WebLogic-specific Artix J2EE Connector deployment descriptor, `weblogic-ra.xml`, defines the connection factories associated with the Artix J2EE Connector, any dependencies it might have on other services, the JNDI name under which it is registered, and the value of the configuration properties that need to be defined for the connection factories.

The following example `weblogic-ra.xml` file is used to deploy the Artix J2EE Connector in the `Hello World` demo:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE weblogic-connection-factory-dd ...>
<weblogic-connection-factory-dd>
  <connection-factory-name>ArtixConnector
</connection-factory-name>
  <jndi-name>ArtixConnector</jndi-name>
</weblogic-connection-factory-dd>
```

Run the Hello World demo

To deploy the Artix J2EE Connector and an example application to WebLogic, see [“Running the Hello World Demo on WebLogic” on page 19](#).

More information

For more detailed deployment information, please refer to the WebLogic documentation.

Deploying to WebSphere

Overview

This section gives an overview of how to deploy the Artix J2EE Connector to WebSphere and points you to a demo that walks you through deployment and shows you a running application. The following topics are covered:

- [Deployment steps](#)
 - [Run the Hello World demo](#)
 - [More information](#)
-

Deployment steps

To deploy the Artix J2EE Connector to WebSphere:

1. Set the Artix environment before running WebSphere. See [“Setting the Artix Environment” on page 52](#) for more detail.
2. Run the following Jython script to deploy the Artix J2EE Connector and create a connection factory:

On Windows:

```
WebSphereHome\bin\wsadmin.bat -lang jython -f
ArtixInstallDir\artix\Version\demos\j2ee\hello_world_soap_http\
etc\rardeploy.py
<nodeName> ArtixInstallDir\lib\artix\j2ee\Version\artix.rar
```

On UNIX:

```
WebSphereHome/bin/wsadmin.sh -lang jython -f
ArtixInstallDir/artix/Version/demos/j2ee/hello_world_soap_http/
etc/rardeploy.py
<nodeName> ArtixInstallDir/lib/artix/j2ee/Version/artix.rar
```

Alternatively, you can use the WebSphere Administrative Console to deploy the Artix J2EE Connector.

Note: On WebSphere 6.0.x a client-view jar for the target EJB(s) must be available to the application server.

Run the Hello World demo

To deploy the Artix J2EE Connector and an example application to WebSphere, see [“Running the Hello World Demo on WebSphere” on page 23](#).

More information

For more detailed deployment information, please refer to the WebSphere documentation.

For more information on Jython, see www.jython.org.

Transactions

Transaction support is an essential part of any enterprise application architecture. The Artix J2EE Connector supports local and global (XA) transactions as specified by the J2EE Connector Architecture.

In this chapter

This chapter covers the following topics:

Transactions Overview	page 64
Local Transactions	page 67
Global Transactions	page 72

Note: Transaction support is not available in all editions of Artix. Please check the conditions of your Artix license to see whether your installation supports transactions.

Transactions Overview

What is a transaction?

A transaction is a single unit of work that can contain several programming steps. When a transaction executes, each step must complete successfully to ensure data integrity. If one step in a transaction fails, all of the steps in that transaction must *roll back*. As a result, data that the transaction was attempting to modify remains unaffected by the failure. If all the steps succeed, the transaction *commits* and all data modifications resulting from the transaction become permanent.

Non-transactional software processes can sometimes proceed and sometimes fail, and sometimes fail after only half completing their task. This can be a disaster for certain applications. The most common example is a bank fund transfer: imagine a failed software call that debited one account but failed to credit another. A transactional process, on the other hand, is secure and reliable because it is guaranteed to succeed or fail in a completely controlled way.

Example

The classical example of a transaction is that of funds transfer in a banking application. This involves two operations: a debit of one account and a credit of another. To combine these operations into a single unit of work, the following properties are required:

- If the debit operation fails, the credit operation should fail, and vice-versa; that is, they should both work or both fail.
- The system goes through an inconsistent state during the process (between the debit and the credit). This inconsistent state should be hidden from other parts of the application.
- The committed results of the whole operation should be permanently stored.

ACID properties

Every transaction must obey what is known as the ACID properties:

Atomic	All of the operations in a transaction must be performed successfully for the transaction to be successful. Data modifications are either all committed or aborted (rolled back) when the transaction completes.
Consistent	A transaction is a unit of work that takes a system from one consistent state to another.
Isolated	While a transaction is executing, its partial results are hidden from other entities accessing the system.
Durable	The results of a transaction are persistent and can be recovered after a system or media failure.

Transaction managers

Most resource managers, for example databases and message queues, support native transactions. If, however, an application requires two or more resource managers to be part of the same transaction, then a third-party transaction manager is needed to coordinate the transaction and to ensure that the ACID properties of the transaction are maintained.

The application uses the transaction manager to create the transaction. Each resource manager accessed during the transaction becomes a participant in the transaction. When the application completes the transaction, either with a commit or rollback request, the transaction manager communicates with each resource manager.

Two-phase commit

When there are two or more participants involved in a transaction the transaction manager uses a two-phase-commit (2PC) protocol to ensure that all participants agree on the final outcome of the transaction despite any failures that may occur. Briefly the 2PC protocol works as follows:

- In the first phase, the transaction manager sends a prepare message to each participant. Each participant responds to this message with a vote indicating whether the transaction should be committed or rolled back.

- The transaction manager collects all the prepare votes and makes a decision on the outcome of the transaction. If all participants voted to commit, the transaction can commit. However, if a least one participant voted to rollback, the transaction is rolled back. This completes the first phase.
 - In the second phase, the transaction manager sends either commit or rollback messages to each participant.
-

One-phase commit

If there is only one participant in the transaction the transaction manager can use a one-phase commit (1PC) protocol instead of the 2PC protocol which can be expensive in terms of the number of messages sent and the data that must be logged. The 1PC protocol essentially delegates the transaction completion to the single resource manager.

Artix transaction support

The Artix J2EE Connector's transaction support is built over Artix transaction support. Artix supports distributed transactions using the following protocols:

- CORBA binding over IIOP.
- SOAP binding over any compatible transport.

The underlying transaction system used by Artix can be replaced within a pluggable framework. Currently, the following transaction systems are supported:

- OTS Lite.
- OTS Encina.
- WS-AtomicTransactions (WS-AT).

For more information on the transaction systems supported by Artix, see the [Artix Transactions Guide, Java Edition](#).

Note: Transaction support is not available in all editions of Artix. Please check the conditions of your Artix license to see whether your installation supports transactions.

Local Transactions

Overview

A local transaction is defined as a transaction that is managed internally by a resource manager, such as the Artix J2EE Connector. An external transaction manager is not involved in the coordination of such transactions.

The Artix J2EE Connector supports local transactions as specified by the J2EE Connector Architecture (J2CA) `LocalTransaction` interface. It supports the `begin()`, `commit()` and `rollback()` transaction demarcation methods. When the Artix J2EE Connector is used in the context of a local transaction, it propagates a transaction with every invocation.

This section discusses how the Artix J2EE Connector's local transaction support works, using the J2EE local transaction demo as an example.

How local transaction support works

The Artix J2EE Connector's local transaction support is based on the local transaction contract defined in the J2CA specification. For more information on this contract, see the J2CA specification on [Sun Microsystems' website](http://java.sun.com/j2ee/connector/download.html) (<http://java.sun.com/j2ee/connector/download.html>).

The runtime use of the local transaction contract is at the discretion of the J2EE application server and is transparent to the J2EE application.

Local transaction demo

Artix includes a simple demo that shows the Artix J2EE Connector participating in a local transaction. It is located in the following directory of your Artix installation:

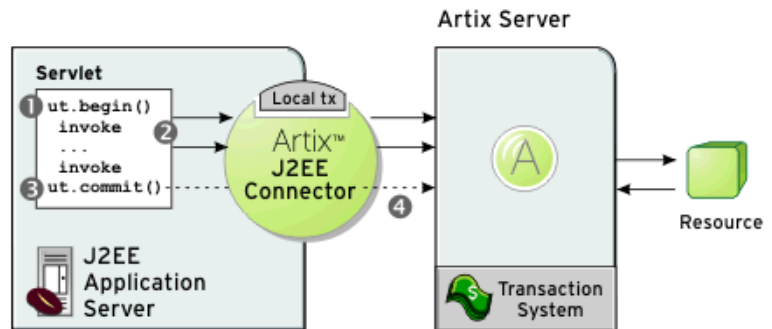
```
InstallDir/artix/Version/demos/j2ee/local_transactions
```

To run the demo, follow the instructions in the `README.txt` file located in the demo directory.

Graphical representation

Figure 4 graphically represents what is happening in the J2EE local transaction demo:

Figure 4: *Artix J2EE Connector Participating in Local Transactions*



1. The servlet calls `ut.begin()` to initiate a transaction.
2. Within the transaction, the servlet calls one or more of the WSDL operations on the remote server, using the Artix J2EE Connector. The WSDL operations are transactional, requiring updates to a persistent resource.
3. The servlet calls `ut.commit()` to make permanent any changes caused during the transaction. Note that the servlet could, alternatively, call `ut.rollback()` to abort the transaction. This scenario is also shown in the demo.
4. The transaction system performs the commit phase by sending a notification to the server that it should perform a one-phase commit.

Demo code example

[Example 7](#) is taken from the local transaction demo servlet code. Sections of the code have been omitted for clarity:

Example 7: Local Transaction Demo Code

```
1 InitialContext ic = new InitialContext();
  ut = (UserTransaction)
    ic.lookup("java:comp/UserTransaction");
  ArtixConnectionFactory factory = (ArtixConnectionFactory)
    ic.lookup(EIS_JNDI_NAME);

  Data data = ...

2 ut.begin();

3     URL wsdlLocation =
  getClass().getResource("/soap_tx_demo.wsdl");
4     QName serviceName = new
  QName("http://www.iona.com/transaction_demo", "DataServiceA");
5     QName portName = new QName("", "DataSOAPPort");

6     data = (Data)factory.getConnection(Data.class, wsdlLocation,
  serviceName, portName);

7     int readValue = data.read();
  data.write(readValue + 1);
  readValue = data.read();

8 ut.commit();
9 ((Connection)data).close();

10 ut.begin();
11     data = (Data)factory.getConnection(Data.class, wsdlLocation,
  serviceName, portName);

12     data.write(readValue + 1);
13     readValue = data.read();

14 ut.rollback();

15     readValue = data.read();

16 ((Connection)data).close();
```

The code shown in [Example 7 on page 69](#) can be explained as follows:

1. Resolves an `ArtixConnectionFactory` for the Artix J2EE Connector resource adapter, a user transaction and data reference.
2. Begins a transaction.
3. Determines the WSDL location URL from the classpath using the JVM runtime.
4. Creates a `QName` that identifies which service in the WSDL file the client wants to use.
5. Creates a `QName` that identifies which port in the WSDL file the client wants to use.
6. Creates a `Connection` object using the `ArtixConnectionFactory` and casts the connection to the `Data` interface.
7. Reads the data value from the Artix server. Adds "1" to the data value and reads the value from the server again. When you run the demo the values are printed to the screen and you can see the data value being increased by one.
8. Commits the transaction.
9. Closes the `Connection`.
10. Begins another transaction.
11. Creates a `Connection` object using the `ArtixConnectionFactory` and casts the connection to the `Data` interface.
12. Reads the data value from the server and adds "1".
13. Reads the new data value from the server.
14. Rolls back the transaction.
15. Reads the data value from the server. This confirms that the transaction did not go ahead and "1" was not added to the original value read from the server.
16. Closes the `Connection`.

Configuring local transactions

The Artix J2EE Connector is configured out of the box to support no transactions—the Artix J2EE Connector deployment descriptor file, `ra.xml`, specifies `NoTransaction`. To configure the Artix J2EE Connector to participate in local transactions you must change the transaction support element in the `ra.xml` to `LocalTransaction` by:

1. Making sure the Artix environment is set. See “[Setting the Artix Environment](#)” on page 52 for more detail.
2. Running one of the following `ant` tasks, depending on whether your J2EE application server supports J2CA 1.0 or 1.5:

J2CA 1.0

Windows:

```
> ant %IT_ARTIX_VER_DIR%\etc\j2ee\build.xml ra.dd.10.local
```

UNIX:

```
ant -f $IT_ARTIX_VER_DIR/etc/j2ee/build.xml ra.dd.10.local
```

J2CA 1.5

Windows:

```
> ant %IT_ARTIX_VER_DIR%\etc\j2ee\build.xml ra.dd.15.local
```

UNIX:

```
ant -f $IT_ARTIX_VER_DIR/etc/j2ee/build.xml ra.dd.15.local
```

The `ant` utility is a Java-based build tool. It is bundled with Artix. The `ant` target is defined in the `build.xml` file, which is located in the `ArtixInstallDir/artix/Version/etc/j2ee` directory. For more information about `ant`, see <http://ant.apache.org/>.

Global Transactions

Overview

A global transaction is one in which two or more resources are involved and an external transaction manager is needed to coordinate the updates to both resource managers. The Artix J2EE Connector supports global transactions for both outbound and inbound transactional invocations. That is, the Artix J2EE Connector supports outbound transactional invocations from a J2EE application to an Artix Web service, as well as supporting inbound transactional invocations to a J2EE application that has been exposed as a Web service by Artix.

The Artix J2EE Connector supports global transactions via the `XAResource` interface as specified by the J2CA specification. For more information, see the J2CA specification available on [Sun Microsystems' website](http://java.sun.com/j2ee/connector/download.html) (<http://java.sun.com/j2ee/connector/download.html>).

In this section

This section includes the following subsections:

Outbound Global Transactions	page 73
Inbound Global Transactions	page 76

Outbound Global Transactions

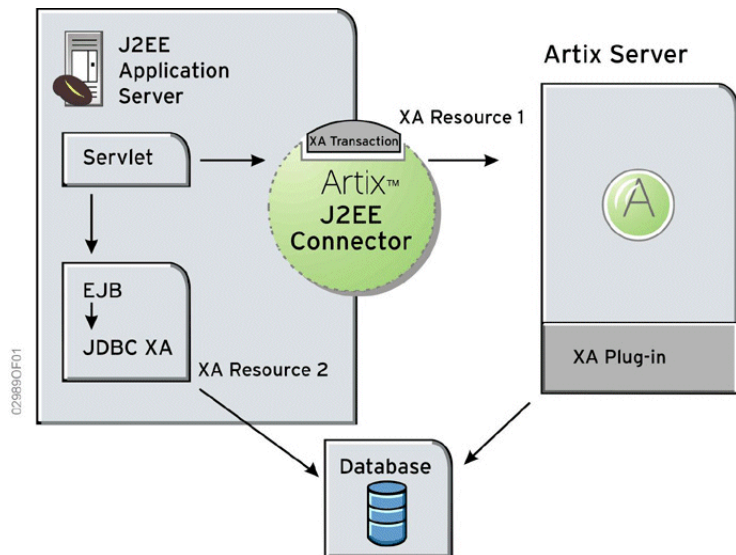
Overview

The application server manages the process of enlisting the Artix J2EE Connector in a global transaction when appropriate. The Artix J2EE Connector uses Artix to begin a subordinate transaction, which it exposes to the application server through the `javax.transactions.xa.XAResource` interface. When the application server completes its transaction, Artix completes the subordinate transaction.

Graphical representation

Figure 5 shows the Artix J2EE Connector participating in an outbound global transaction.

Figure 5: *Artix J2EE Connector in an Outbound Global Transaction*



Association between superior and subordinate transactions

There is an association between the subordinate transaction created by Artix and the superior application server transaction. The XID of the superior transaction is replicated in the subordinate transaction. This means that

operations issued under the control of the subordinate transaction share the global and branch transaction identifiers with the Artix J2EE Connector. This is important when the components taking part in a transaction need to have visibility of data during a global transaction.

Code example

[Example 8](#) shows a simple Java code example in which the Artix J2EE Connector is used in a transaction together with an EJB. Error handling has been omitted for clarity.

Example 8: *Using the Artix J2EE Connector in a Global Transaction*

```

1  ArtixConnectionFactory cf = ...
   MyDataBeanHome beanHome = ...
   UserTransaction ut = ...

2  ut.begin();

3  ArtixConnection conn = cf.getConnection(DataAccess.class);

4  DataAccess data = (DataAccess) conn;
   MyDataBean beanData = beanHome.create(...);
5  beanData.doTransactionalWork();
6  data.doAdditionalDependentTransactionalWork();

7  ut.commit();

8  conn.close();

```

1. Resolve an `ArtixConnectionFactory` for the Artix J2EE Connector resource adapter, a bean and a user transaction reference.
2. Begin a transaction.
3. Create an `ArtixConnection` object using the `ArtixConnectionFactory`.
4. Cast the `ArtixConnection` to an application-specific Java interface, in this case `DataAccess`.
5. Make a transactional invocation on your EJB
6. Make a transactional invocation on the data object.
7. Commit the transaction.
8. Close the `ArtixConnection`.

Configuring outbound global transaction support

The Artix J2EE Connector is configured out of the box as a J2CA 1.0 capable connector that supports no transactions—the Artix J2EE Connector deployment descriptor file, `ra.xml`, to specifies `NoTransaction`. To configure the Artix J2EE Connector to participate in outbound global transactions you must change the transaction support element in the `ra.xml` to `XATransaction` by:

1. Making sure the Artix environment is set. See “[Setting the Artix Environment](#)” on page 52 for more detail.
2. Running one of the following `ant` tasks, depending on whether your J2EE application server supports J2CA 1.0 or 1.5:

J2CA 1.0

Windows:

```
> ant -f %IT_ARTIX_VER_DIR%\etc\j2ee\build.xml ra.dd.10.xa
```

UNIX:

```
ant -f $IT_ARTIX_VER_DIR/etc/j2ee/build.xml ra.dd.10.xa
```

J2CA 1.5

Windows:

```
> ant -f %IT_ARTIX_VER_DIR%\etc\j2ee\build.xml ra.dd.15.xa
```

UNIX:

```
ant -f $IT_ARTIX_VER_DIR/etc/j2ee/build.xml ra.dd.15.xa
```

The `ant` utility is a Java-based build tool. It is bundled with Artix. The `ant` target is defined in the `build.xml` file, which is located in the `ArtixInstallDir/artix/Version/etc/j2ee` directory. For more information about `ant`, see <http://ant.apache.org/>.

Inbound Global Transactions

Overview

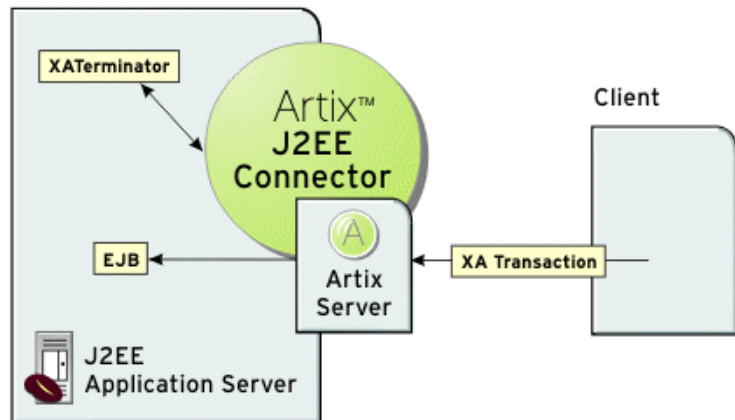
J2CA 1.5 specifies a contract between an application server and a resource adapter that allows a resource adapter to propagate an inbound transaction to the application server, so that the application server and subsequent participants can do work as part of the inbound transaction. For more detail about this contract, see “Transaction Inflow” chapter of the [J2CA 1.5 specification](#).

The Artix J2EE Connector uses this functionality when it exposes an EJB deployed in a J2CA 1.5 capable application server as an Artix service. It wraps the `XATerminator` instance provided by the application server in an `XAResource` object that is enlisted with the Artix transaction manager. This allows the application server to participate in a transaction that originates from an Artix client, and that is managed by an Artix transaction manager.

Graphical representation

Figure 6 shows the Artix J2EE Connector participating in an inbound global transaction.

Figure 6: *Artix J2EE Connector in an Inbound Global Transaction*



Configuring inbound global transaction support

The Artix J2EE Connector is configured out of the box as a J2CA 1.0 capable connector that supports no transactions—the Artix J2EE Connector deployment descriptor file, `ra.xml`, specifies the J2CA 1.0 specification and `NoTransaction`. To configure the Artix J2EE Connector to participate in global transactions you must change the J2CA specification element and the transaction support element in the `ra.xml` to 1.5 and `XATransaction` respectively. You can do this by:

1. Making sure the Artix environment is set. See [“Setting the Artix Environment” on page 52](#) for more detail.
2. Running the following `ant` task:

Windows:

```
> ant -f %IT_ARTIX_VER_DIR%\etc\j2ee\build.xml ra.dd.15.xa
```

UNIX:

```
ant -f $IT_ARTIX_VER_DIR/etc/j2ee/build.xml ra.dd.15.xa
```

The `ant` utility is a Java-based build tool. It is bundled with Artix. The `ant` target is defined in the `build.xml` file, which is located in the `ArtixInstallDir/artix/Version/etc/j2ee` directory. For more information about `ant`, see <http://ant.apache.org/>.

Security

The Artix J2EE Connector supports credentials propagation. It propagates username and password details along with outbound and inbound Web service requests.

In this chapter

This chapter discusses the following topics:

Outbound Security	page 80
Configuring Outbound Security	page 83
Inbound Security	page 89
Configuring Inbound Security	page 92
Configuring a Secure Transport	page 99

Outbound Security

Overview

The Artix J2EE Connector is configured by default to support the propagation of a username and password with Web service requests from the J2EE domain to Artix Web services. The identity is used by Artix on the server side to authenticate the Web service operation.

In this section

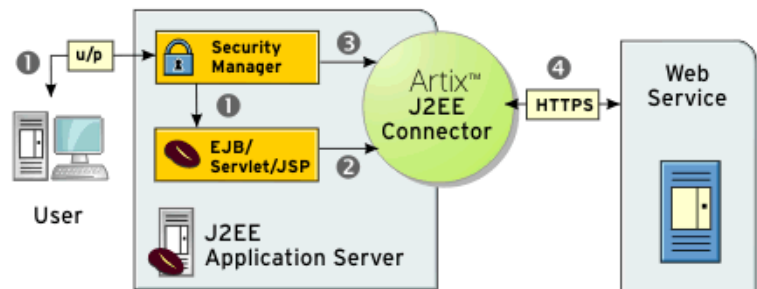
This section gives a high-level overview of how the Artix J2EE Connector outbound security works. The following topics are covered:

- [Graphical representation](#)
- [Scenario description](#)
- [How it works](#)

Graphical representation

[Figure 7](#) illustrates a scenario in which the Artix J2EE Connector propagates username and password credentials with outbound connections:

Figure 7: *Artix J2EE Connector Propagating Credentials with Outbound Connections*



Scenario description

The scenario shown in [Figure 7](#) can be described as follows:

Stage	Description
1	The user logs in to the J2EE application and is authenticated. The J2EE authenticated user invokes on the EJB/Servlet/JSP.
2	The EJB/Servlet/JSP invokes on the Artix J2EE Connector to get a connection to a Web service.
3	The J2EE application server maps the J2EE authenticated user to an appropriate subject for the Artix J2EE Connector. This is known as credentials or principal mapping.
4	The Artix J2EE Connector makes a remote invocation on the Web service and transmits the mapped username and password credentials with the request over a secured transport.

How it works

The Artix J2EE Connector security support details are contained in its deployment descriptor, `ra.xml`, as follows:

Example 9: *Artix J2EE Connector ra.xml file fragment*

```

1 <authentication-mechanism>
  <authentication-mechanism-type>BasicPassword
  </authentication-mechanism-type>
2 <credential-interface>javax.resource.security.
  PasswordCredential
  </credential-interface>
</authentication-mechanism>

```

1. Specifies that the Artix J2EE Connector supports username and password-based authentication.
2. Specifies the interface that the Artix J2EE Connector supports for the representation of the credentials. The `javax.resource.security.PasswordCredential` interface specifies to the application server that it should pass a `Subject` containing a `PasswordCredential` that includes a username and a password to the Artix J2EE Connector.

These entries are defined in the J2EE Connector Architecture specification. For more information, see the specification on [Sun Microsystems' website](http://java.sun.com) (<http://java.sun.com>).

When you deploy the Artix J2EE Connector to your J2EE application server, the authentication-mechanism entry in the deployment descriptor indicates to the application server that the connector supports container-managed sign-on. When an application requests that the Artix J2EE Connector create a new connection, the application server passes any security information associated with that application or user in a `Subject` that contains a `PasswordCredential`. The contents of the `PasswordCredential` are controlled by the application server, based on credentials or principal mapping configuration.

The Artix J2EE Connector uses the `PasswordCredential` to set the Artix bus security context using the Artix context API. It sets the WSSE username and password token. It ensures that the credentials associated with a connection are passed to Artix before each request. How the credentials are propagated over the transport is specific to an Artix binding, and is specified in the WSDL contract. Artix can be configured to send the credentials as a SOAP header or as a HTTP header. For more information, see the [Artix Security](#) guide.

Configuring Outbound Security

Overview

The Artix J2EE Connector is configured by default to support credentials propagation with outbound connections. You must, however, configure your application server to pass the J2EE authenticated username and password to the Artix J2EE Connector with each call to the connector's `getConnection` method. This is known as credentials or principal mapping. If you do not configure your application server with credentials mapping, a `null` subject will be passed to the Artix J2EE Connector with each call to `getConnection` and the Artix J2EE Connector will not propagate a username and password with Web service requests.

In this section

How you configure credentials mapping is specific to the J2EE application server that you are using. This section gives a brief description of credentials mapping. JBoss is used in an example of how to configure credentials mapping. The following topics are covered:

- [Credentials Mapping](#)
- [Configuring Credentials Mapping in JBoss](#)

Credentials Mapping

Overview

When a J2EE Connector Architecture connection factory is configured to perform container managed sign-on, the application server must be configured to map the caller principal to a resource principal. The application server creates a `Subject` instance that contains the configured security domain credentials of the back-end resource. The `Subject` returned by a credential or principal mapping contains a `PasswordCredential` that represents the caller identity for the back-end resource. The application server automatically passes the `Subject` to the J2EE Connector Architecture resource adapter with each call to the resource adapter's `getConnection` method.

In this subsection

This subsection gives a brief description of the types of credentials mapping. Please refer to your application server documentation for exact details of how to perform credentials mapping. The following topics are covered:

- [Credentials passed as is](#)
 - [Many-to-one mapping](#)
 - [One-to-one mapping](#)
-

Credentials passed as is

In the simplest case, the application server is configured to pass the caller's credentials as is to the resource adapter. For example, if the username is `Bob` and the password is `BobsPassword`, then `Bob` and `BobsPassword` are passed to the resource adapter.

Many-to-one mapping

For a many-to-one credentials mapping, the application server is configured to map all callers' credentials to single username and password for the resource adapter. For example:

Table 1: *Many-to-One Mapping*

Caller Credentials (Username/Password)	Resource Credentials (Username/Password)
Bob/BobsPassword	Artix/ArtixPassword
Tom/TomsPassword	Artix/ArtixPassword
Jane/JanesPassword	Artix/ArtixPassword

One-to-one mapping

For a one-to-one credentials mapping, the application server is configured to map the each caller's credentials to a username and password that uniquely identifies them for the resource adapter. For example:

Table 2: *One-to-One Mapping*

Caller Credentials (Username/Password)	Resource Credentials (Username/Password)
Bob/BobsPassword	BobArtix/BobsArtixPassword
Tom/TomsPassword	TomArtix/TomsArtixPassword
Jane/JanesPassword	JaneArtix/JanesArtixPassword

This is the most complex type of credentials mapping and most application servers delegate the mapping to a security provider, such as JAAS or LDAP.

Configuring Credentials Mapping in JBoss

Overview

JBoss uses a Java Authentication and Authorization Service (JAAS) to do credentials or principal mapping. JBoss JAAS configuration details are contained in the JBoss JAAS configuration file, `login-config.xml`.

In this subsection

This subsection gives an overview of JAAS and tells you how to configure credentials mapping in JBoss. The following topics are covered:

- [Java Authentication and Authorization Service \(JAAS\)](#)
 - [Configuring credentials mapping](#)
 - [Example JBoss login-config.xml](#)
 - [Example Artix J2EE Connector deployment descriptor](#)
 - [More information](#)
-

Java Authentication and Authorization Service (JAAS)

JAAS provides an API that represents an extensible authentication and authorization service. The API allows components to remain independent from underlying authentication technologies. The sequence of operations that occur when an authorization attempt is made are dependent on configuration, but remain hidden to the application component.

For more information on JAAS and to see the Javadoc, see Sun Microsystems's website: <http://java.sun.com/products/jaas/overview.html>

Configuring credentials mapping

To configure credentials mapping in JBoss, you must:

1. Add an application-policy element to the JBoss JAAS login configuration file, `login-config.xml`, and specify that it will be used by the Artix J2EE Connector.
2. Indicate to the Artix J2EE Connector that it must use the security domain specified by the application policy. To do this, you must add a security domain element that specifies the application-policy name that you used in the `login-conf.xml` file, to the Artix J2EE Connector deployment descriptor, `CFactoryName-ds.xml`

Example JBoss login-config.xml

For example, the following JBoss `login-config.xml` file shows an application policy that specifies that the `calleridentity` configuration is to be used by the Artix J2EE Connector:

Example 10: JBoss `login-config.xml` fragment

```

<?xml version='1.0'?>
<!DOCTYPE policy PUBLIC
    "-//JBoss//DTD JBOSS Security Config 4.0//EN"
    "http://www.jboss.org/j2ee/dtd/security_config.dtd">
<policy>...
1  <application-policy name="calleridentity">
    <authentication>
2      <login-module code =
3          "org.jboss.resource.security CallerIdentityLoginModule"
4          flag = "required">
        <module-option name = "managedConnectionFactoryName">
            jboss.jca:service=NoTxCM,name=ArtixConnector
        </module-option>
        <module-option name =
            "userName">dummy_user</module-option>
        <module-option name =
            "password">dummy_password</module-option>
        </login-module>
    </authentication>
    </application-policy>
</policy>

```

The entries in this JBoss `login-config.xml` file can be explained as follows:

1. Specifies an `application-policy` element called `calleridentity`.
2. Specifies that the JBoss caller identity login module will be used. This login module implementation simply copies the supplied username and password pair as is into a `PasswordCredential`. For example, if the username is `Bob` and the password is `BobsPassword`, then `Bob` and `BobsPassword` will be propagated to the Artix J2EE Connector.
3. The `managedConnectionFactoryName` module option ties this configuration to a particular deployed `ConnectionFactory` instance of the Artix J2EE Connector.
4. The `dummy_user` and `dummy_password` elements indicate the default credentials that should be used in the absence of an existing authenticated user.

Example Artix J2EE Connector deployment descriptor

For example, the following JBoss 4 `artixj2ee_1_5-ds.xml` file fragment specifies to the Artix J2EE Connector that it must use the `calleridentity` configuration, as defined in the JBoss `login-conf.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<connection-factories>
  <no-tx-connection-factory>
    <jndi-name>ArtixConnector</jndi-name>
    <security-domain>calleridentity</security-domain>
    <rar-name>artix.rar</rar-name>
    <connection-definition>com.iona.connector.
      ArtixConnectionFactory</connection-definition>
    ...
  </no-tx-connection-factory>
</connection-factories>
```

More information

For more information on how to configure credentials mapping for a J2EE Connector Architecture resource adapter in JBoss, please refer to the JBoss documentation.

Inbound Security

Overview

The Artix J2EE Connector can be configured to support J2EE authentication for inbound communications. The username and password propagated with a Web service request can be used to authenticate against the J2EE application server before the request is dispatched to the EJB. The principal identified by the propagated username and password pair must correspond to a J2EE user that has sufficient privileges to execute the requested operation on the EJB.

In this section

This section gives a high-level overview of how the Artix J2EE Connector inbound security works. The following topics are covered:

- [Exposing a J2EE application as a Web service](#)
 - [Graphical representation](#)
 - [Scenario description](#)
 - [How it works](#)
-

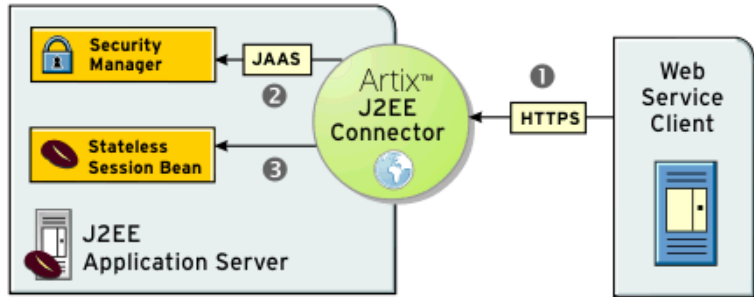
Exposing a J2EE application as a Web service

To understand how the Artix J2EE Connector supports inbound security, you must first understand how the Artix J2EE Connector exposes a J2EE application as a Web service. For details, see [“Exposing a J2EE Application as a Web Service” on page 41](#).

Graphical representation

Figure 8 illustrates a scenario in which the Artix J2EE Connector propagates username and password credentials with inbound connections:

Figure 8: *Artix J2EE Connector Propagating Credentials with Inbound Connections*



Scenario description

The scenario shown in Figure 8 can be described as follows:

Stage	Description
1	An Artix Web service client invokes on the Web service and sends a username and password over HTTPS. The Artix J2EE Connector uses the Artix context API to obtain the username and password from the Artix bus security context. How these are propagated over the transport are specific to an Artix binding and are specified in the WSDL contract.
2	The Artix J2EE Connector uses JAAS to perform a login to the application server.
3	The Artix J2EE Connector uses a JAAS <code>subject.doAs()</code> method to invoke on the target EJB.

How it works

The Artix J2EE Connector uses JAAS to login to the application server. It uses a JAAS configuration that identifies a login module that authenticates against the application server. It uses a JAAS `Subject.doAs()` method to invoke on the target EJB. The `doAs()` method ensures that the calling thread has the appropriate access control information. Using JAAS allows the Artix J2EE Connector to remain application server independent.

Configuring Inbound Security

Overview

To configure inbound security you must secure your EJB; configure the Artix J2EE Connector to enable it to login to your application server; and configure the Artix J2EE Connector with a username and password that identify the principal that will be used to create the EJB.

In this section

This section walks you through these configuration steps. The following topics are covered:

- [Securing the Target EJB](#)
- [Configuring JAAS Login Module](#)
- [Configuring EJB Create Username and Password](#)

Securing the Target EJB

Overview

You must secure the EJB using J2EE access controls. That is, you must specify method permissions in the assembly descriptor element of your EJB deployment descriptor, `ejb-jar.xml`. This subsection provides an example of such a deployment descriptor. The following topics are covered:

- [Example EJB deployment descriptor](#)
- [JBoss example](#)
- [More information](#)

Example EJB deployment descriptor

For example, the following EJB deployment descriptor file fragment declares a role called "BobsRole" that can access all `GreeterBean` methods:

Example 11: *GreeterBean ejb-jar.xml file fragment*

```
...
<assembly-descriptor>
  <security-role>
    <role-name>BobsRole</role-name>
  </security-role>
  <method-permission>
    <role-name>BobsRole</role-name>
    <method>
      <ejb-name>GreeterBean</ejb-name>
      <method-name>*</method-name>
    </method>
  </method-permission>
</assembly-descriptor>
```

JBoss example

JBoss uses JAAS for application server authentication. The corresponding deployment descriptor, `jboss.xml`, must be augmented to include a `security-domain` element that identifies the JAAS configuration that contains the relevant concrete role definitions.

For example, the following `jboss.xml` file fragment specifies the security domain as follows:

```
<jboss>
  <security-domain>java:jaas/other</security-domain>
  <enterprise-beans>
    <session>
      <ejb-name>GreeterBean</ejb-name>
    ...
  </enterprise-beans>
</jboss>
```

More information

For more detail, please refer to your application server documentation.

Configuring JAAS Login Module

Overview

The Artix J2EE Connector uses JAAS to login to the application server. It needs, however, to know which JAAS configuration name it should use in the login procedure. To configure the Artix J2EE Connector to login to your application server, you must set the `JAASLoginConfigName` configuration property to the JAAS configuration name that will be used to locate the appropriate JAAS login module. The configuration name is passed as an argument to the constructor of a `javax.security.auth.login.LoginContext` that is subsequently used by the Artix J2EE Connector to login to the application server.

In this subsection

How JAAS is configured is specific to the application server you are using. This subsection uses JBoss as an example application server to describe how to configure the Artix J2EE Connector with JAAS login module details. The following topics are covered:

- [JAAS configuration in JBoss](#)
 - [Setting JAASLoginConfigName in JBoss](#)
 - [More information](#)
-

JAAS configuration in JBoss

JAAS is configured in JBoss through the JBoss `login-config.xml` JAAS configuration file. This file contains application-policy elements that describe the different configurations. Each application-policy element contains a series of login modules that are used to implement authentication. The Artix J2EE Connector needs to use the preconfigured "client-login" application-policy entry. This entry specifies a login module that enables the application server to authenticate and verify that the Artix J2EE Connector supplied username and password correspond to a valid J2EE principal. This is required because the Artix J2EE Connector dispatches to an EJB that is protected by J2EE access controls.

Setting JAASLoginConfigName in JBoss

To configure the Artix J2EE Connector with details of the JBoss JAAS configuration name that it should use in the JAAS login procedure, set the `JAASLoginConfigName` configuration property to `client-login` in the Artix J2EE Connector deployment descriptor, `CFactory-ds.xml`.

For example, in JBoss 4, you set it as follows in the `artixj2ee_1_5-ds.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<connection-factories>
  <no-tx-connection-factory>
    <jndi-name>ArtixConnector</jndi-name>
    ...
    <config-property name="JAASLoginConfigName"
      type="java.lang.String">client-login</config-property>
    ...
  </no-tx-connection-factory>
</connection-factories>
```

More information

For more information on how JAAS is configured in your application server and for information on how to set J2EE Connector Architecture resource adapter configuration properties, please refer to your application server documentation.

For more information on the `JAASLoginConfigName` configuration property, see [“JAASLoginConfigName” on page 139](#).

Configuring EJB Create Username and Password

Overview

The Artix J2EE Connector must create an instance of this target EJB to determine the method arguments that must be read from an Artix transport. Security information propagated with a request is not available until the read is complete. As a result, the Artix J2EE Connector does not have sufficient dynamic security information available at the point when `EJBHome.ejbCreate` is called. The Artix J2EE Connector must, therefore, be statically configured with a username and password pair that it can use to login to the application server to execute the create method.

In this subsection

This subsection gives details of the configuration properties that you must set. It uses JBoss as an example application server to describe how to configure the Artix J2EE Connector with a username and password pair that it can use to login to the application server to execute the create method. The following topics are covered:

- [Configuration properties](#)
 - [Setting JAASLoginUserName and JAASLoginPassword in JBoss](#)
 - [More information](#)
-

Configuration properties

The Artix J2EE Connector supports the `JAASLoginUserName` and `JAASLoginPassword` configuration properties to allow this static configuration. The values of username and password must identify a valid J2EE user that has the appropriate privileges to execute the `EJBHome.create` method of the target EJB. Even if the target EJB is configured to allow unchecked access to the create method, a valid J2EE identity must be configured for the Artix J2EE Connector to allow the JAAS login to proceed.

Setting JAASLoginUserName and JAASLoginPassword in JBoss

The following example shows a fragment of a JBoss Artix J2EE Connector deployment descriptor, `artixj2ee_1_5-ds.xml`, which sets the username and password properties to `artix`:

Example 12: *Setting JAASLoginUserName and JAASLoginPassword in JBoss*

```
<?xml version="1.0" encoding="UTF-8"?>
<connection-factories>
  <no-tx-connection-factory>
    <jndi-name>ArtixConnector</jndi-name>
    ...
    <config-property name="JAASLoginUserName"
type="java.lang.String">artix</config-property>
    <config-property name="JAASLoginPassword"
type="java.lang.String">artix</config-property>
    ...
  </no-tx-connection-factory>
</connection-factories>
```

More information

For more information on how JAAS is configured in your application server and for information on how to set J2EE Connector Architecture resource adapter configuration properties, please refer to your application server documentation.

For more information on the `JAASLoginUserName` configuration property, see [“JAASLoginUserName” on page 140](#).

For more information on `JAASLoginPassword` configuration property, see [“JAASLoginPassword” on page 141](#).

Configuring a Secure Transport

Overview

To protect the integrity of the username and password, which is in plain text, the transport needs to be secure. For example, if you are using HTTP, you should configure it to use SSL/TLS security (a combination usually referred to as HTTPS). The SSL/TLS technology allows communication over a secured connection. In this secure connection, the data that is being sent is encrypted before being sent, then decrypted upon receipt and prior to processing.

More information

For information on how to configure a secure transport, see the [Artix Security Guide](#).

Part III

Using Artix in a Servlet Container

In this part

This part contains the following chapters:

Exposing Artix Web Services from a Servlet Container
--

page 103

Exposing Artix Web Services from a Servlet Container

You can expose Artix Web services from a servlet container. Client applications can invoke on the Web services through the HTTP port assigned to the servlet container or using any of the transports supported by Artix. This chapter walks you through the typical steps involved.

In this chapter

This chapter discusses the following topics:

Introduction	page 104
Configuring Servlet Container to Run an Artix Application	page 107
Building an Artix Application	page 112
Building and Deploying your Web Application	page 123

Introduction

Overview

Artix provides the servlet component of the Web service. It provides a basic servlet, the `ArtixServlet.class`, and a servlet transport plug-in, which you can use to route HTTP requests to the servlet onto Artix. These components are written in Java and are compiled and archived in a JAR file, `it_artix_servlet.jar`, which is located in:

```
ArtixInstallDir/lib/artix/java_runtime/4.0
```

You must write the Web service implementation class and an Artix Java plug-in. The Artix Java plug-in is required to create an instance of your Web service implementation and register it with the Artix bus.

In this section

This section outlines the steps you must complete to develop and deploy an Artix Web service to a servlet container. The following topics are covered:

- [Implementation steps](#)
 - [Graphical representation](#)
 - [How it works](#)
 - [Demo](#)
-

Implementation steps

The following is a high-level view of the steps that you need to complete to expose your Web service from a servlet container. It assumes that the Web service WSDL file already exists. If, however, you need to develop a WSDL file, please refer to the [Understanding Artix Contracts](#) guide.

Step	Action
1	Configure your servlet container so that it can run Artix applications.
2	Build an Artix Web service application. This includes generating an Artix Java plug-in.

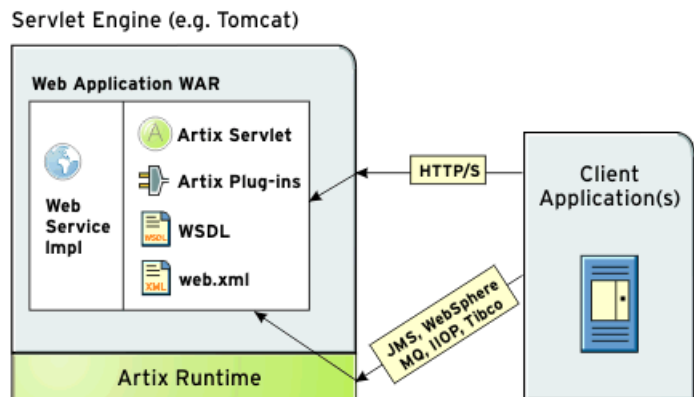
Step	Action
3	Build a Web application WAR file that includes the Artix servlet, the Artix servlet transport plug-in, your application, its deployment descriptor <code>web.xml</code> , the Artix Java plug-in for your application, and the Web service WSDL file.
4	Deploy the WAR file to your servlet container.

The rest of this chapter describes these steps in more detail.

Graphical representation

Figure 9 graphically illustrates how you can expose an Artix Web service from a servlet container.

Figure 9: *Exposing Artix Web Service from a Servlet Container*



How it works

The Artix servlet initializes an Artix bus within its `init()` method. It uses the bus initialization parameters that you provide in the Web service deployment descriptor file, `web.xml`. During initialization, the Artix bus loads the servlet transport plug-in and the Artix Java plug-in that you have created for your application. The role of the Artix Java plug-in is to create an instance of the Web service and register it with the Artix bus. In essence, it associates an Artix servant with a WSDL port.

Client applications use the information in the Web service WSDL file to initialize a proxy to the target Web service. Client requests can be sent to the servlet container TCP/IP port or to any port that is defined in the WSDL contract, using any of the transports supported by Artix, and are processed by the Artix Web service.

Demo

Some of the examples used in this chapter are taken from the `Servlet Container` demo, which can be found in:

```
ArtixInstallDir/artix/Version/demos/j2ee/servlet_container
```

If you want to run this demo, see the `README.txt` file in the demo directory.

Configuring Servlet Container to Run an Artix Application

Overview

Before you can deploy an Artix Web service to your servlet container, you must configure the servlet container so that it can run Artix applications. How you do this is dependent on the servlet container that you are using. This section highlights the key configuration steps that you must complete and uses Tomcat and WebLogic as example servlet containers. The following topics are covered:

- [Setting the Artix Environment](#)
- [Make certain Artix JAR files available to your application](#)
- [Configuring the Artix classloader firewall](#)

Setting the Artix Environment

You must set the Artix environment before starting the servlet container.

Tomcat

To set the Artix environment on Tomcat, create and run a local environment script as shown in [Example 13](#)—it is the script used in the `Servlet Container demo`:

Example 13: Script for Setting the Artix Environment on Tomcat

```
1 call "..\..\..\..\bin\artix_env.bat";
2 set IT_DOMAIN_NAME=tomcat
3 set IT_CONFIG_DOMAINS_DIR=%IT_PRODUCT_DIR%\artix\4.0\demos\j2ee\
  servlet_container\etc
  set CLASSPATH=%CLASSPATH%;.
```

1. Call the Artix environment script, `artix_env`. It is located in the `ArtixInstallDir/artix/Version/bin` directory.
2. Reset the value of `IT_DOMAIN_NAME` to specify the name of the configuration domain that Artix should use.

3. Reset the value of `IT_CONFIG_DOMAINS_DIR` to the location of the configuration file.

Note: Alternatively you can specify a domain name and configuration directory in your web application deployment descriptor file, `web.xml`. See “[Example web.xml file](#)” on page 124 for more detail.

For more information on `artix_env`, see the getting started chapter in the [Configuring and Deploying Artix Solutions](#) guide.

WebLogic

To set the Artix environment on WebLogic, create and run a local environment script as follows:

Example 14: Script for Setting the Artix Environment on WebLogic

```

1  @REM Configure for Artix
2  set PATH=ArtixInstallDir\bin;%PATH%
3  set IT_DOMAIN_NAME=weblogic
4  set IT_LICENSE_FILE=ArtixInstallDir\etc\licenses.txt
5  set IT_CONFIG_DOMAINS_DIR=ArtixInstallDir\artix\4.0\demos\j2ee\
   servlet_container\etc
6  set CLASSPATH=
   ArtixInstallDir\lib\common\classloading\1.2\classloading.jar;
   ArtixInstallDir\lib\IONA\lib\common\concurrency\1.2\concurrency.jar;
   ArtixInstallDir\lib\common\ifc\1.2\ifc.jar;
   ArtixInstallDir\lib\artix\java_runtime\4.0\it_bus-api.jar;
   ArtixInstallDir\lib\ws_common\reflect\1.2\
   it_ws_reflect_types.jar;
   ArtixInstallDir\lib\jaxrpc\jaxrpc\1.1\jaxrpc-api.jar;
   ArtixInstallDir\lib\apache\xerces\2.5.0\xercesImpl.jar
   ArtixInstallDir\lib\sun\saaaj\1.2.1\saaaj-api.jar
6  ArtixInstallDir\artix\4.0\demos\j2ee\servlet_container\tomcat\
   shared\classes;
   %CLASSPATH%

```

1. Adds the Artix `bin` directories to the `PATH`. The `bin` directory contains all of the Artix runtime libraries, which are required by each Artix process.
2. Sets `IT_DOMAIN_NAME`, which specifies the name of the configuration domain used by Artix to locate its configuration.
3. Sets `IT_LICENSE_FILE`, which specifies the location of your Artix license file. The default value is `ArtixInstallDir\etc\licenses.txt`.

4. Sets `IT_CONFIG_DOMAINS_DIR`, which specifies the directory where Artix searches for its configuration files. Together, `IT_DOMAIN_NAME` (2 above) and `IT_CONFIG_DOMAINS_DIR` identify the name and location of the configuration file.
5. Adds the required Artix JAR files to the `CLASSPATH`. Note that you must substitute `ArtixInstallDir` with details of your Artix installation directory; for example, `C:\IONA`.
6. Adds the location of the `artix_ce.xml` file to the `CLASSPATH`. Note that you can place the `artix_ce.xml` file in any convenient location, as long as you ensure that the location is on the `CLASSPATH`

Note: The `CLASSPATH` entry should appear on one line.

Make certain Artix JAR files available to your application

The following Artix JAR files must be available to your servlet container so that they can be used by all Artix applications:

- `ArtixInstallDir/lib/common/classloading/1.2/classloading.jar`
- `ArtixInstallDir/lib/common/concurrency/1.2/concurrency.jar`
- `ArtixInstallDir/lib/common/ifc/1.2/ifc.jar`
- `ArtixInstallDir/lib/jaxrpc/jaxrpc/1.1/jaxrpc-api.jar`
- `ArtixInstallDir/lib/artix/java_runtime/4.0/it_bus-api.jar`
- `ArtixInstallDir/lib/ws_common/reflect/1.2/it_ws_reflect_types.jar`
- `ArtixInstallDir/lib/sun/saaj/1.2.1/saaj-api.jar`

Tomcat

If you are using Tomcat, copy these files to your `TomcatInstallDir/shared/lib` directory. The demo build script provided with the `Servlet Container` demo, copies these files for you.

WebLogic

If you are using WebLogic, the script that you created and ran to set the Artix environment places the Artix JARs on the `CLASSPATH`. You do not need to anything else at this stage.

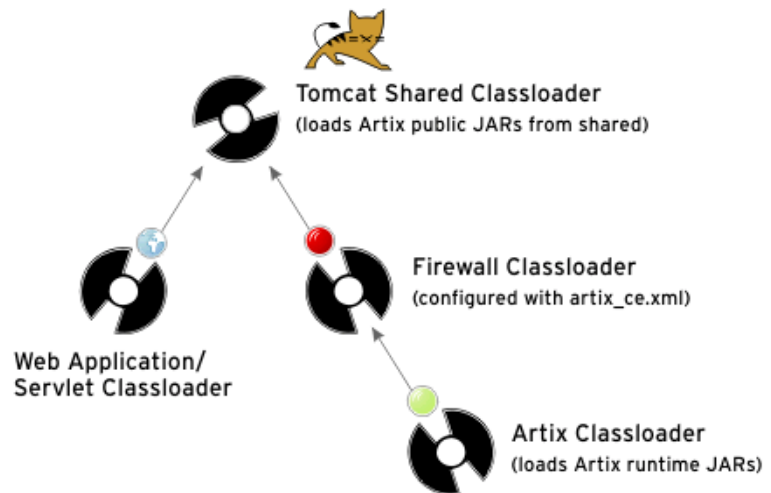
Note: Do not place the Artix JAR files in your Web application's `lib` directory.

Configuring the Artix classloader firewall

Artix requires third-party JAR files that could conflict with different versions of the same JARs required by other servlet container applications. To avoid such issues, you must use of the Artix classloader firewall. The Artix classloader firewall loads specific JARs required by Artix.

Figure 10 shows the classloader configuration. The arrows point to the parent classloader in each case; for example, the Tomcat shared classloader is the parent classloader for the Web application/servlet classloader and the Artix firewall classloader. This setup allows the web application classloader and the Artix classloader to share public classes. It isolates the web application classloader from the Artix classloader, which loads JARs specific to the Artix runtime. With this configuration, the web application classloader which is loading the user code is not polluted with JARs that are needed only by Artix.

Figure 10: *Classloader Configuration*



To enable the Artix classloader firewall, place an `artix_ce.xml` file in a shared location, where it can be detected by Artix. The Artix Servlet Container demo contains an `artix_ce.xml` file that you can use for any Artix application that you are deploying to a servlet container. It is located in the following directory:

```
ArtixInstallDir/artix/Version/demos/j2ee/servlet_container/  
tomcat/shared/classes
```

Tomcat

If you are using Tomcat, copy this `artix_ce.xml` file to your `TomcatInstallDir/shared/classes` directory.

WebLogic

If you are using WebLogic, the script that you created and ran to set the Artix environment places the location of the `artix_ce.xml` file on the `CLASSPATH`. You do not need to do anything else at this stage.

Note: Do not place the `artix_ce.xml` file in your Web application's `classes` directory.

For more information on the Artix classloader firewall, see the *Things to Consider when Developing Artix Applications* chapter, in the [Developing Artix Applications in Java](#) guide.

Building an Artix Application

Overview

This section outlines the steps you must complete to build an Artix application. It includes building an Artix Java plug-in for your application. The role of the Artix Java plug-in is to create an instance of your Web service implementation and register it with the Artix bus. The plug-in must be deployed in your Web application WAR file along with the Web service implementation code.

In this section

This section describes the steps that you must complete to build an Artix Web service application. The following topics are covered:

- [Mapping the WSDL to Java](#)
- [Writing the Implementation Class](#)
- [Developing an Artix Java Plug-in](#)
- [Configuring Artix to Use Your Plug-in](#)

Mapping the WSDL to Java

Overview

The Artix development tools include a `wSDLtojava` command-line utility that you can use to generate Java code from the WSDL file. Artix maps WSDL types to Java using the mapping described in the JAX-RPC specification. This subsection covers the following topics:

- [Syntax of wSDLtojava command](#)
 - [Example](#)
 - [More information](#)
-

Syntax of wSDLtojava command

To generate Java skeleton and plug-in code from a WSDL file, run the following command:

```
wSDLtojava -p package -d <output_dir> -servlet wSDL_contract
```

The parameters shown above are defined as follows:

<code>-p <[wSDL namespace =] Package Name></code>	Specifies the name of the Java package to use for the generated code. You can optionally map a WSDL namespace to a particular package name if your contract has more than one namespace. The <code>-p</code> flag is optional, but is recommended.
<code>-d <output_dir></code>	Specifies the directory to which the generated code is written. The default is the current working directory. The <code>-d</code> parameter is optional.
<code>-servlet</code>	Generates a bus plug-in with the appropriate servant registration code for the generated service implementation and the code required to allow the plug-in to run in a servlet container environment.
<code>wSDL_contract</code>	Specifies the WSDL contract from which the Java code is being generated.

Example

For example, the following `wSDLtoJava` command generates the Java files required to expose the service described in the `hello_world.wsdl` contract in the `Servlet Container demo`. The example shown is run from the directory in which the `hello_world.wsdl` file is stored:

```
wSDLtoJava -p servlet.plugin -d ..\java\servlet\src -servlet
hello_world.wsdl
```

More information

For more information on the `wSDLtoJava` command-line utility, see the [Developing Artix Applications in Java](#) guide.

Writing the Implementation Class

Overview

You can use the skeleton class generated by the Artix `wSDLtojava` utility as the basis for writing your Web service implementation class. All you need to do is add the business logic.

Example

For example, the following `GreeterImpl.java` file is used to implement the Web service in the `Servlet Container` demo:

Example 15: *GreeterImpl.java*

```
package servlet.plugin;

import java.lang.String;
import javax.xml.namespace.QName;
import com.ionajbus.*;

public class GreeterImpl implements java.rmi.Remote {

    public String sayHi() {
        return "Hey Now!";
    }

    public String greetMe(String me) {
        return "Hello " + me;
    }
}
```

Developing an Artix Java Plug-in

Overview

To make your application available to Artix, you must develop an Artix Java plug-in for your application. The purpose of this plug-in is to create an instance of your implementation class and register it with the Artix bus. The code is similar to that of an Artix Java server mainline and it associates your Web service implementation with a WSDL port.

In this subsection

This subsection provides an example plug-in that exposes an Artix Web service over all of the ports defined in the WSDL contract. The following topics are covered:

- [Generating the Artix Java plug-in files](#)
 - [Example of Artix Java plug-in](#)
 - [Example of Artix Java plug-in factory](#)
 - [Exposing a Web service over multiple transports](#)
-

Generating the Artix Java plug-in files

When you map the WSDL to Java, you must use the `-servlet` parameter to generate the Artix Java plug-in code (see [“Mapping the WSDL to Java” on page 113](#) for more information). The following plug-in files are generated for you:

- A plug-in class, which extends the Artix `BusPlugIn` class to implement your application logic.
- A plug-in factory class, which implements the Artix `BusPluginFactory` interface to provide the methods used by the Artix bus to manage your plug-in.

Example of Artix Java plug-in

The code in [Example 16](#) shows an Artix Java plug-in, called `SOAPServicePlugin`. It was generated using the `wsdltojava` utility and the `hello_world.wsdl` contract located in:

```
ArtixInstallDir/artix/Version/demos/j2ee/servlet_container/etc
```

Example 16: An Artix Java Plug-in—`SOAPServicePlugin`

```
package servlet.plugin;

import java.net.URL;
import javax.xml.namespace.QName;

import com.ionajbus.Bus;
import com.ionajbus.BusConstants;
import com.ionajbus.BusException;
import com.ionajbus.BusPlugIn;
import com.ionajbus.Servant;
import com.ionajbus.servants.SingleInstanceServant;

public class SOAPServicePlugin extends BusPlugIn {

    public SOAPServicePlugin(Bus bus) {
        super(bus);
    }

    public void busInit() throws BusException {
        Bus bus = getBus();

        QName serviceName = new QName
            ("http://www.ionajbus.com/servlet/plugin", "SOAPService");
        1 bus.setProperty(BusConstants.ARTIX_SERVLET_SERVICE_QNAME,
            serviceName);
        2 URL url = getClass().getResource("hello_world.wsdl");
        String wsdlLocation = url.toString();
        3 Servant servant = new SingleInstanceServant( new
            GreeterImpl(), wsdlLocation, bus);
        4 bus.registerServant(servant, serviceName);
    }

    public void busShutdown() throws BusException{
    }
}
```

The code shown in [Example 16](#) can be explained as follows:

1. The `bus.setProperty` property is set so that the servlet knows what service is being exposed. The `serviceName` parameter is set the `QName` of the service as defined in the WSDL file. You should only deploy one Artix service per servlet. The servlet uses the value of this property to get the correct WSDL when the `doGet()` method is called on the servlet.
2. Accesses the Web service WSDL file. Note that, in this example, the WSDL file is located within the web application WAR file along with the plug-in. You can, however, retrieve the WSDL file from any location in which it is stored.
3. Creates an instance of the servant.
4. Registers the servant and activates all ports associated with the service.

Example of Artix Java plug-in factory

The code in [Example 17](#) shows an Artix Java plug-in factory class, called `SOAPServicePluginFactory`.

Example 17: *Artix Java Plug-in Factory Implementation—`SOAPServicePluginFactory`*

```
package servlet.plugin

import com.iona.jbus.Bus;
import com.iona.jbus.BusPlugIn;
import com.iona.jbus.BusPlugInFactory;
import com.iona.jbus.BusException;

public class SOAPServicePluginFactory implements
    BusPlugInFactory {

1   public BusPlugIn createBusPlugIn(Bus bus) throws BusException{
    return new SOAPServicePlugin(bus);
    }

2   public void destroyBusPlugIn(BusPlugIn plugin) throws
    BusException{

    }
}
```


The code shown in [Example 17](#) can be explained as follows:

1. The `createBusPlugIn()` method creates an instance of the Artix Java plug-in, `SOAPServicePluginFactory`, and its associated resources, and associates them with particular bus instances.
2. The `destroyBusPlugIn()` method destroys plug-in instances and frees the resources associated with them.

You do not need to modify this code.

Exposing a Web service over multiple transports

If you want to expose your service over transports other than HTTP, all you need to do is add a port definition for the transport to the WSDL contract. You do not need to change the code. Artix supports a number of transports, including IIOP, JMS, WebSphere MQ, TIBCO, and Tuxedo. You can use any of these when deploying an Artix Web service into a servlet container. The following WSDL extract, for example, defines two ports for the `SOAPService`, and specifies that clients should use HTTP to contact `Port1` and IIOP to contact `Port2`:

```
<wsdl:service name="SOAPService">
  <wsdl:port binding="tns:Greeter_SOAPBinding" name="Port1">
    <soap:address location="http://localhost:9000"/>
      <http-conf:client/>
      <http-conf:server/>
    </wsdl:port>
  <wsdl:port binding="tns:GreeterCORBABinding" name="Port2">
    <corba:address
      location="file:../../greeter_service.ior"/>
    </wsdl:port>
</wsdl:service>
```

Both ports are activated when `bus.registerServant(servant, serviceName)` is called, as shown in [Example 16 on page 117](#).

Configuring Artix to Use Your Plug-in

Overview

You must configure Artix so that the Artix bus can load your plug-in. This subsection describes the configuration entries that are required and provides an example configuration file. The following topics are covered:

- [Plug-in configuration](#)
 - [Example configuration file](#)
 - [More information](#)
-

Plug-in configuration

To enable the Artix bus to load your plug-in, add the following configuration entries to your Artix configuration file:

Step	Action
1	Load the Java plug-in loader. Artix Java plug-ins require the Artix bus to use a special Java plug-in loader, <code>java</code> . You need to add this plug-in loader to the <code>orb_plugins</code> list.
2	Specify your application-specific plug-in factory class and the Artix servlet transport plug-in factory class. To load a plug-in, the Artix bus needs to know which factory class is used to create instances of the plug-in's implementation.
3	Add your plug-in and the Artix servlet transport plug-in to the <code>java_plugins</code> list that the Artix bus will load.

Example configuration file

The following is an example of the configuration file used to configure Artix in the `Servlet Container` demo. It defines two Artix configuration scopes: `demos.client` and `tests.servlet_test`

Example 18: Artix Configuration File—`servlet_container.cfg`

```

1 include "../../../etc/domains/artix.cfg";
2 demos {
    servlet_container {
        client {
            # to see transport buffers, use this setting
            #event_log:filters = ["*=FATAL+ERROR+WARNING+INFO_MED"];
            orb_plugins = ["xmlfile_log_stream"];
        };
    };
};
3 tests {
    #uncomment the following configuration entries to see Artix
    #message logging
    #the log will be written into the Tomcat install directory
    #event_log:filters=["*=FATAL+ERROR+WARNING+INFO_MED"];
    #plugins:soap:write_xsi_type="true";

    servlet_test
    {
4     orb_plugins = ["xmlfile_log_stream", "java"];
5     java_plugins = ["servlet_transport", "servlet_demo_plugin"];
6     plugins:servlet_transport:classname="com.iona.jbus.servlet.
    transport.ServletTransportPlugInFactory";
7     plugins:servlet_demo_plugin:classname="servlet.plugin.
    SOAPServicePlugInFactory";
    };
};

```

1. Includes the `artix.cfg` file, which is the standard minimal Artix configuration. It is generated by default when Artix is installed.
2. `demos.client` scope. This is the scope under which the C++ and Java clients run in the `Servlet Container` demo. This scope is not essential—the client applications would run just as well under the global scope in `artix.cfg`.

3. `tests.servlet_test` scope. This is the scope under which the Artix servlet runs within the servlet container. This is essential. The `orb_plugins` and `java_plugins` entries identify Artix plug-ins that need to be loaded by the Artix bus.
4. Note that the Java plug-in loader, `java`, is included in the `orb_plugins` list.
5. Note that the `servlet_transport` and `servlet_demo_plugin` is included in the `java_plugins` list.
6. The `servlet_transport` plug-in is part of Artix. This is contained in the `it_artix_servlet.jar` file and provides the integration between the Artix servlet running in the servlet container and the Artix core. It defines a new Artix transport that wraps the servlet container HTTP stack. This enables Artix Web services to receive invocations on the TCP/IP port used by servlet container.

Note: If you do not want to use the servlet container's HTTP stack, and would prefer instead to use the Artix HTTP stack, do not add the `servlet_transport` plug-in to the list of plug-ins that you want the Artix bus to load.

7. The `servlet_demo_plugin` is the Artix Web services implementation written specifically for the `Servlet Container` demo. This is an example of an application-specific Artix Java plug-in and contains the demo application logic. This is equivalent to the Artix Java plug-in that you must generate for your Web service application. Details of how to write such a plug-in is described in the [Developing an Artix Java Plug-in](#) subsection of this chapter.

More information

For more detailed information on how to configure Artix plug-ins, see the *Configuring Artix Plug-ins* chapter in the [Developing Artix Applications in Java](#) guide.

Building and Deploying your Web Application

Overview

To deploy your application to your servlet container, you must build a Web Archive (WAR) file and deploy it to your servlet container. In addition, if you want to use the servlet container HTTP port to receive messages, you must deploy the Artix servlet transport and ensure that the Web service WSDL file contains the URL on which the servlet will be deployed.

In this section

This section discusses the following topics:

- [Building a WAR file](#)
 - [Example web.xml file](#)
 - [Ensuring the URL assigned to servlet is same as in WSDL](#)
 - [Deploying the WAR file](#)
-

Building a WAR file

Build a WAR file to include:

1. A copy of the Artix supplied `it_artix_servlet.jar` file in the `WEB-INF/lib` directory. This contains the `ArtixServlet` class and the plug-in code that provides the integration between the servlet and the servlet container's HTTP stack. You do not need to change this in any way. It is located in:

```
ArtixInstallDir/lib/artix/java_runtime/4.0
```

2. Your Web service implementation class, your application-specific Artix Java plug-in class, the plug-in factory class, and the Web service WSDL file. If required, other classes generated by the `wsdltojava` command should also be included; for example, application-specific types and the type factory.

You can either build an `ApplicationSpecific.jar` file to package all of these files and include it in the `WEB-INF/lib` directory of your WAR file, or place the files (including the class hierarchy) in the `WEB-INF/classes` directory.

3. A `web.xml` deployment descriptor file in the `WEB-INF` directory. You must include an initialization parameter that the Artix servlet can use when initializing the Artix bus. See [Example web.xml file](#) for more detail.

Example web.xml file

When deploying an Artix Web service to your servlet container, you must include an initialization parameter in your application `web.xml` deployment descriptor file. It is used by the `ArtixServlet` instance when initializing an Artix bus and ensures that the bus is using the correct Artix configuration scope.

For example, the following is used when deploying the `Servlet Container demo`:

Example 19: Servlet Container demo web.xml file

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>Artix Servlet Test App</display-name>
  <description></description>
  <servlet>
    <servlet-name>ArtixServlet</servlet-name>
    <servlet-class>com.iona.jbus.servlet.ArtixServlet
    </servlet-class>
    <init-param>
      <param-name>bus.init.parameters</param-name>
      <param-value>-ORBid SomeUniqueString
      -ORBname tests.servlet_test</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <!-- Action Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>ArtixServlet</servlet-name>
    <url-pattern>/artixServlet</url-pattern>
  </servlet-mapping></web-app>

```

1

The code shown [Example 19](#) can be explained as follows:

1. To make an Artix process run under a particular configuration scope, you specify that scope using the `-ORBname` parameter. It specifies the scope under which the Artix bus should run. In this case the configuration scope is `test.servlet_test`, which has been defined in the Artix configuration file used in the `Servlet Container` demo. See [“Example configuration file” on page 121](#) to view the contents of this file.

In addition, to run multiple Artix servlet applications in the same servlet container, you need to distinguish one application’s bus from another. To do this, set the `-ORBid` parameter to a unique string for each application.

Lastly, you could specify a particular domain name and configuration directory by adding `-ORBdomain_name` and `-ORBconfig_domains_dir` parameters and their values to the `param-value` entry. If you choose to do so, you do not need to set these configuration entries in your environment script.

Ensuring the URL assigned to servlet is same as in WSDL

In order for the servlet to use the servlet container’s HTTP stack, you must ensure that the URL and TCP/IP port number in the Web service WSDL file is the same as that used to deploy the servlet. You can either change the value in the WSDL file to match that of the servlet, or configure the servlet container to use the URL and TCP/IP port number specified in the WSDL.

For example, in the `Servlet Container` demo, the `hello_world.wsdl` file specifies the following URL and Tomcat is configured to use the same port:

```
<wsdl:service name="SOAPService">
<wsdl:port binding="tns:Greeter_SOAPBinding" name="SoapPort">
  <soap:address
    location="http://localhost:9876/artix_demo_servlet/
      artix_servlet"/>
</wsdl:port>
</wsdl:service>
```

Note: If you choose not to use the servlet container’s HTTP stack, and are instead using the Artix HTTP stack, then you must ensure that the TCP/IP port number used in the WSDL file is different from that used by the servlet container.

Deploying the WAR file

You must configure your servlet container to run Artix applications before you deploy your WAR file. Please refer to [“Configuring Servlet Container to Run an Artix Application” on page 107](#) for more detail.

How you deploy your WAR file is dependent on the servlet container that you are using. Please refer to your servlet container documentation for exact details.

Part IV

Reference Information

In this part

This part contains the following chapters:

Artix J2EE Connector Configuration Properties

page 129

Artix J2EE Connector Configuration Properties

You do not have to configure the Artix J2EE Connector for basic connection management. It is configured for you during the Artix installation. You can, however, change the default configuration settings to suit your environment using the configuration properties detailed in this chapter. This chapter also provides some basic information on how to set these configuration properties in JBoss, WebLogic and WebSphere.

In this chapter

This chapter covers the following topics:

Configuration Properties	page 130
Setting Configuration Property Values	page 142

Configuration Properties

Overview

The Artix J2EE Connector supports the following configuration properties:

ArtixInstallDir	page 131
ArtixLicenseFile	page 132
LogLevel	page 133
ConfigurationDomain	page 134
ConfigurationScope	page 135
EJBServicePropertiesURL	page 136
EJBServicePropertiesPollInterval	page 137
MonitorEJBServiceProperties	page 138
JAASLoginConfigName	page 139
JAASLoginUserName	page 140
JAASLoginPassword	page 141

ArtixInstallDir

Overview

The `ArtixInstallDir` configuration property specifies the Artix installation directory. This is set by default when you install Artix.

Value

The value of the `ArtixInstallDir` configuration property is a string specifying the Artix installation directory.

The Artix J2EE Connector is configured by default with details of the directory into which you installed Artix; for example:

```
C:\IONA\artix\4.0
```

Setting

If you want to change the default setting, see [“Setting Configuration Property Values” on page 142](#).

ArtixLicenseFile

Overview

The `ArtixLicenseFile` configuration property specifies the location of the Artix license file. This is set to a default location when you install Artix. If, however, you do not store your Artix license file in the default location, you need to set the `ArtixLicenseFile` configuration property to specify the location that you are using.

Value

The value of the `ArtixLicenseFile` configuration property is a string specifying the location of the Artix license file.

The Artix J2EE Connector is set by default to specify the default location as:

```
InstallDir/etc/licenses.txt
```

where *InstallDir* represents the directory in which you installed Artix. An example could be:

```
C:/IONA/etc/licenses.txt
```

Setting

If you want to change the default setting, see [“Setting Configuration Property Values” on page 142](#).

LogLevel

Overview

The `LogLevel` configuration property specifies the amount of logging that the Artix J2EE Connector produces. The location of the logging output is dependent on the J2EE application server.

Value

The logging support levels from least to most verbose are:

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`
- `FATAL`

The Artix J2EE Connector is configured by default to support the `WARN` logging level.

Setting

If you want to change the default setting, see [“Setting Configuration Property Values” on page 142](#).

ConfigurationDomain

Overview

The Artix J2EE Connector uses the Artix configuration file, `artix.cfg`, by default. An alternative configuration domain can be specified by using the `ConfigurationDomain` configuration property.

Value

The value of the `ConfigurationDomain` configuration property is a string. The Artix J2EE Connector is configured by default with the configuration domain value of `artix`.

Setting

If you want to change the default setting, see [“Setting Configuration Property Values” on page 142](#).

ConfigurationScope

Overview

The `ConfigurationScope` configuration property specifies the Artix configuration scope that the Artix J2EE Connector uses.

Value

The value of the `ConfigurationScope` configuration property is a string, with the `.` (dot) character identifying nested configuration scopes.

The Artix J2EE Connector is configured by default with a configuration scope of `DEFAULT`.

Setting

If you want to change the default setting, see [“Setting Configuration Property Values” on page 142](#).

EJBServicePropertiesURL

Overview

The `EJBServicePropertiesURL` configuration property specifies the location from which the Artix J2EE Connector can retrieve the `ejb_servants.properties` file.

By default, the Artix J2EE Connector is set to check this file for updates at 30 second intervals. This behavior is controlled by the `MonitorEJBServiceProperties` and the `EJBServicePropertiesPollInterval` configuration properties.

Value

The value is a string that specifies a URL.

The Artix J2EE Connector is configured by default with the following file URL:

```
file:ArtixInstallDir/artix/Version/etc/ejb_servants.properties
```

Note: If you want the Artix J2EE Connector to check the `ejb_servants.properties` file for updates, the URL must be a file URL.

Setting

If you want to change the default setting, see [“Setting Configuration Property Values” on page 142](#).

More detail

For more detail on the `ejb_servants.properties` file, see [“Configuring Inbound Connections” on page 47](#).

For more detail on the `MonitorEJBServiceProperties` configuration property, see [“MonitorEJBServiceProperties” on page 138](#).

For more detail on the `EJBServicePropertiesPollInterval` configuration property, see [“EJBServicePropertiesPollInterval” on page 137](#).

EJBServicePropertiesPollInterval

Overview

The `EJBServicePropertiesPollInterval` configuration property specifies the refresh period that the Artix J2EE Connector uses to check the `ejb_servants.properties` file for updates. It is dependent on the `MonitorEJBServiceProperties` configuration property being set to `TRUE`.

Value

The value is an integer and the default value is 30 seconds. This means that, by default, the Artix J2EE Connector checks the `ejb_servant.properties` file every 30 seconds for updates.

Setting

If you want to change the default setting, see [“Setting Configuration Property Values” on page 142](#).

More detail

For more detail on the `ejb_servants.properties` file, see [“Configuring Inbound Connections” on page 47](#).

For more detail on the `MonitorEJBServiceProperties` configuration property, see [“MonitorEJBServiceProperties” on page 138](#).

MonitorEJBServiceProperties

Overview

The `MonitorEJBServiceProperties` configuration property controls whether or not the Artix J2EE Connector checks the `ejb_servants.properties` file for updates.

Value

The value is a boolean and can be set to:

<code>TRUE</code>	This is the default setting and enables the Artix J2EE Connector to monitor the <code>ejb_servants.properties</code> file for updates. For this to work, the location of the <code>ejb_servants.properties</code> file must be specified as a file URL to the <code>EJBServicePropertiesURL</code> configuration property.
<code>FALSE</code>	The Artix J2EE Connector will check the <code>ejb_servants.properties</code> file once on deployment to an application server, but will not check for updates.

The Artix J2EE Connector is configured by default to `TRUE`.

How often it checks the `ejb_servants.properties` file is set by the `EJBServicePropertiesPollInterval` configuration property. The default value of is every 30 seconds.

Setting

If you want to change the default setting, see [“Setting Configuration Property Values” on page 142](#).

More detail

For more detail on the `ejb_servants.properties` file, see [“Configuring Inbound Connections” on page 47](#).

For more detail on the `EJBServicePropertiesURL` configuration property, see [“EJBServicePropertiesURL” on page 136](#).

For more detail on the `EJBServicePropertiesPollInterval` configuration property, see [“EJBServicePropertiesPollInterval” on page 137](#).

JAASLoginConfigName

Overview

The `JAASLoginConfigName` configuration property is used to specify the JAAS configuration name that the Artix J2EE Connector should use to login to a J2EE application server for secure inbound connections. The configuration name is passed as an argument to the constructor of a `javax.security.auth.login.LoginContext` that the Artix J2EE Connector uses to login to the application server.

Value

The value is a string that specifies the JAAS security configuration name that the Artix J2EE Connector uses to login to the application server. The Artix J2EE Connector is configured by default to use a JAAS configuration name of `DEFAULT`.

Setting

For information on how to set the `JAASLoginConfigName` configuration property, see [“Setting Configuration Property Values” on page 142](#)

More detail

For more detail on using the `JAASLoginConfigName` configuration property, see [“Configuring JAAS Login Module” on page 95](#).

JAASLoginUserName

Overview

The `JAASLoginUserName` configuration property is used to identify a valid J2EE username that has the appropriate privileges to execute the `EJBHome.create` method of the target EJB for secure inbound connections.

Value

The value is a string that specifies a valid J2EE username the Artix J2EE Connector can use to create the target EJB for secure inbound connections. The Artix J2EE Connector is configured by default to use a J2EE username of `DEFAULT`.

Setting

For information on how to set the `JAASLoginUserName` configuration property, see [“Setting Configuration Property Values” on page 142](#).

More detail

For more detail on using the `JAASLoginUserName` configuration property, see [“Configuring EJB Create Username and Password” on page 97](#).

JAASLoginPassword

Overview

The `JAASLoginPassword` configuration property is used to specify a password that corresponds to a valid J2EE user that has the appropriate privileges to execute the `EJBHome.create` method of the target EJB for secure inbound connections.

Value

The value is a string that specifies a valid password that the Artix J2EE Connector can use to create the target EJB for secure inbound connections. The Artix J2EE Connector is configured by default to use a J2EE user password of `DEFAULT`.

Setting

For information on how to set the `JAASLoginPassword` configuration property, see [“Setting Configuration Property Values” on page 142](#).

More Detail

For more detail on using the `JAASLoginPassword` configuration property, see [“Configuring EJB Create Username and Password” on page 97](#).

Setting Configuration Property Values

Overview

Artix J2EE Connector configuration property values can be set at deployment time. How you do this is specific to the J2EE application server that you are using. This section provides details of how to set example Artix J2EE Connector properties in JBoss, WebLogic and WebSphere. Please consult your J2EE application server documentation for the most appropriate way in which to set these values.

In this section

The following topics are covered:

Setting Configuration Property Values in JBoss	page 143
Setting Configuration Property Values in WebLogic	page 144
Setting Configuration Property Values in WebSphere	page 145

Setting Configuration Property Values in JBoss

Overview

JBoss provides J2EE Connector Architecture resource adapter factory configuration through a `CFactoryName-ds.xml` deployment descriptor file. This is a separate file from the resource adapter RAR file. You need one `CFactoryName-ds.xml` file per connection factory.

Example

The following example `artixj2ee_1_5-ds.xml` JBoss 4 deployment descriptor file specifies a value of 60 for the Artix J2EE Connector `EJBServicePropertiesPollInterval` configuration property:

```
<connection-factories>
  <no-tx-connection-factory>
    <jndi-name>ArtixConnector</jndi-name>
    ...
    <config-property name="EJBServicePropertiesPollInterval"
      type="java.lang.Integer">60</config-property>
    ...
  </no-tx-connection-factory>
</connection-factories>
```

The `config-property` element is used to specify a value for a configuration property that is supported by the resource adapter being deployed.

More information

For more information on the `EJBServicePropertiesPollInterval` configuration property, see [“EJBServicePropertiesPollInterval” on page 137](#). For more information on how to set J2EE Connector Architecture resource adapter configuration properties in JBoss, see the JBoss documentation.

Setting Configuration Property Values in WebLogic

Overview

WebLogic provides J2EE Connector Architecture resource adapter factory configuration through the `weblogic-ra.xml` deployment descriptor file. This file is typically included in the resource adapter RAR file. Although, WebLogic 8.1 allows the location of the `weblogic-ra.xml` deployment descriptor file to be specified by the deployment tool.

Example

The following example `weblogic-ra.xml` deployment descriptor file specifies a value of 60 for the Artix J2EE Connector `EJBServicePropertiesPollInterval` configuration property:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE weblogic-connection-factory-dd PUBLIC
"-//BEA Systems, Inc.//DTD WebLogic 8.1.0 Connector//EN"
"http://www.bea.com/servers/wls810/dtd/weblogic810-ra.dtd">
<weblogic-connection-factory-dd>
  <connection-factory-name>ArtixConnector
  </connection-factory-name>
  <jndi-name>CORBAConnector</jndi-name>
  <map-config-property>
    <map-config-property-name>EJBServicePropertiesPollInterval
    </map-config-property-name>
    <map-config-property-value>60
    </map-config-property-value>
  </map-config-property>
</weblogic-connection-factory-dd>
```

The `map-config-property` element is used to specify a value for a configuration property that is supported by the resource adapter being deployed.

More information

For more information on the `EJBServicePropertiesPollInterval` configuration property, see [“EJBServicePropertiesPollInterval” on page 137](#). For more information on how to set J2EE Connector Architecture resource adapter configuration properties in WebLogic, see the WebLogic documentation.

Setting Configuration Property Values in WebSphere

Overview

WebSphere requires you to set J2EE Connector Architecture resource adapter factory configuration using the WebSphere Administrative Console GUI or the `wsadmin` command-line tool.

More information

For more information on how to set J2EE Connector Architecture resource adapter configuration properties in WebSphere, see the WebSphere documentation.

Index

Numerics

1PC 66
2PC 65

A

ACID properties 65
API
 connection management 31
application policy
 adding to JBoss login-config.xml 87
 client-login 95
 configuring credentials mapping in JBoss 87
artix.cfg 121, 134
artix.rar
 deploying to JBoss 54
 deploying to WebLogic 57
 deploying to WebSphere 60
Artix bus 43, 47, 82, 105
 accessing directly 35
 context registry 35
 creating a reference 35
ArtixConnectionFactory 32
 usage scenarios 33
ArtixConnectionFactory.getBus() 35
Artix environment
 setting 52
artix_env script 52
Artix HTTP stack 122
ArtixInstallDir 131
artixj2ee_1_5-ds.xml 16, 143
artix_j2ee_ce.xml 53
artixj2ee-ds.xml 55
 deploying to JBoss 54
 example of 55
Artix Java plug-in 116
 configuring 120
ArtixLicenseFile 132
Artix servlet transport 103, 104
 it_artix_servlet.jar 123
Artix shared library
 appending to system environment 52
authentication mechanism
 BasicPassword 81

B

BasicPassword 81
BusPlugIn
 extending 117
BusPlugInFactory
 extending 118

C

CFactoryName-ds.xml 16, 55, 143
 example of 55
classloader firewall 110
client-login 95
configuration
 inbound security 92
 outbound security 83
ConfigurationDomain 134
configuration properties
 ArtixInstallDir 131
 ArtixLicenseFile 132
 ConfigurationDomain 134
 ConfigurationScope 135
EJBServicePropertiesPollInterval 49, 137
EJBServicePropertiesURL 48, 136
JAASLoginConfigName 139
JAASLoginPassword 141
JAASLoginUserName 140
LogLevel 133
MonitorEJBServiceProperties 49, 138
 setting in JBoss 143
 setting in WebLogic 144
 setting in WebSphere 145
ConfigurationScope 135
connection management 31
 API 31
 Artix J2EE Connector 8
 interface definition 32
 J2EE Connector Architecture 4
credentials mapping 83, 84
 in JBoss 86
credentials propagation 79
 outbound security 80

D

- d 113
- deployment
 - interface classes 38
 - to JBoss 54
 - to servlet container 123
 - to Tomcat 123
 - to WebLogic 57
 - to WebSphere 60
- deployment descriptor 7, 124

E

- EJB
 - securing 93
- ejb_servants.properties file
 - configuring inbound connections 47
 - example of 48
 - format of 47
 - multiple entries 48
 - port-to-JNDI mapping 43, 47
- EJBServicePropertiesPollInterval 49, 137
- EJBServicePropertiesURL 48, 136

G

- global transactions 72
 - configuring inbound 77
 - configuring outbound 75

H

- Hello World demo
 - location of 14
 - running on JBoss 15
 - running on WebLogic 19
 - running on WebSphere 23
 - WSDL file 14

I

- IIOp 119
- inbound connections 41
 - configuring 47
 - demo 43
- inbound security
 - configuring 92
- initialization parameter 124
- init param See initialization parameter
- interface classes
 - packaging and deploying 37, 38, 123

it_artix_servlet.jar 123

J

- J2EE application
 - writing 31, 45, 115
- J2EE Connector Architecture 2
 - Common Client Interface, CCI 5
 - connection management 4
 - security management 4
 - system-level contracts 4
 - transaction management 4
- JAAS 86
- JAAS configuration name 95
- JAASLoginConfigName 95, 139
- JAAS login module 95
- JAASLoginPassword 97, 141
 - setting in JBoss 98
- JAASLoginUserName 97, 140
 - setting in JBoss 98
- Java Authentication and Authorization Service 86
- java_plugins 120, 122
- javax.resource.security.PasswordCredential 81
- javax.security.auth.login.LoginContext 95
- JAX-RPC mapping 43
- JBoss
 - 4 deployment descriptor 16, 143
 - configuring property values in 143
 - credentials mapping 86
 - deploying to 54
 - jboss.xml 38
 - login-config.xml 87
 - mapping resource reference 38
 - principal mapping 86
 - running the Hello World demo on 15
- JMS 119

L

- local transactions 67
 - demo 67
 - demo code 69
- login-config.xml 87, 95
- LogLevel 133

M

- MonitorEJBServiceProperties 49, 138

O

one-phase commit 66
 -ORBconfig_domains_dir 125
 -ORBdomain_name 125
 -ORBid 125
 orb_plugins 120, 122
 OTS Encina 66
 OTS Lite 66

P

-p 113
 param-value 125
 principal mapping 83, 84
 in JBoss 86

R

ra.xml 7
 RAR. See artix.rar
 resource adapter archive file. See artix.rar
 resource reference
 declaring 37
 mapping 37

S

security
 configuring inbound 92
 configuring outbound 83
 credentials mapping 83
 credentials propagation 79
 inbound 89
 outbound 80
 principal mapping 83
 security management
 Artix J2EE Connector 8
 J2EE Connector Architecture 4
 -servlet 113
 servlet container 103–126
 Artix Java plug-in 116
 configuring Artix Java plug-in 120
 demo 106
 example Artix configuration file 121
 example of extending BusPlugIn 117
 example of extending BusPlugInFactory 118
 graphical representation 105
 running Artix services in 103
 using multiple transports or protocols 119
 SLSB 43
 implementing 45

stateless session bean 43
 implementing 45

T

TIBCO 119
 transaction management
 Artix J2EE Connector 8
 J2EE Connector Architecture 4
 transaction managers 65
 transactions 63–70
 1PC definition 66
 2PC definition 65
 ACID properties of 65
 configuring inbound global 77
 configuring inbound XA 77
 configuring outbound global 75
 configuring outbound XA 75
 global transactions 72
 local transactions 67
 local transactions demo 67
 local transactions demo code 69
 one-phase commit 66
 OTS Encina 66
 OTS Lite 66
 two-phase commit 65
 WS-AtomicTransactions 66
 XA transactions 72
 Tuxedo 119

W

web.xml
 deploying Web service in servlet container 124
 initialization parameter 124
 WebLogic
 configuring property values in 144
 deploying to 57
 mapping resource reference 38
 running the Hello World demo on 19
 weblogic.xml 38
 weblogic-ra.xml 144
 example of 59
 WebSphere
 configuring property values in 145
 deploying to 60
 mapping resource reference 38
 running the Hello World demo on 23
 WebSphere MQ 119
 WS-AtomicTransactions 66

INDEX

WS-AT See WS-AtomicTransactions

wSDL contract 113

WSDL location

- configuring Artix to resolve at runtime 35

- hardcoding 33

- resolving at runtime 34

WSDL to Java

- mapping 44

wsdltjava utility 30, 44

- generating Java skeleton code 113

WSSE

- username and password 82

X

XA transactions 72

- configuring inbound 77

- configuring outbound 75