Artix™

Deploying and Managing Artix Solutions

Version 3.0, June 2005

Making Software Work Together™

Updated: 08-Nov-2005

Contents

Part II Using Artix Services

List of Tables

List of Figures

Preface

What is Covered in this Book

Deploying and Managing Artix Solutions explains how to deploy and manage Artix services in a runtime environment. It presents different approaches to deployment topography, and the merits of each. This book also provides detailed descriptions of the specific tasks involved in configuring and launching Artix applications and services.

This book does not discuss the specifics of the different middleware and messaging products that Artix interacts with. Any discussion about the features of specific middleware products or transports relates to how Artix interacts with these features. It is assumed that you have a working knowledge of the specific middleware products and transports you are using.

Who Should Read this Book

The main audience of *Deploying and Managing Artix Solutions* is Artix system administrators. However, anyone involved in designing a large scale Artix solution will find the general discussions about Artix deployment topographies and Artix services useful.

Knowledge of specific middleware or messaging transports is not required to understand the general topics discussed in this book. However, if you are using this book as a guide to deploying runtime systems, you should have a working knowledge of the middleware transports that you intend to use in your Artix solutions.

When deploying Artix in a distributed architecture with other middleware, please see the documentation for that middleware product. You may require access to an administrator. For example, a Tuxedo administrator is required to complete a Tuxedo distributed architecture.

How to Use this Book

Part I, Introduction

If you are new to Artix, Chapter 1 and Chapter 2 provide a high-level overview of using Artix to solve integration projects, and how Artix fits into a software environment.

Part II, Using Artix Services

If you are using Artix services, you may want to read one or more of the following:

- Chapter 3 explains how to use the Artix container to deploy and manage Artix Web services.
- Chapter 4 explains how to use the Artix locator service to find references to Artix servers.
- Chapter 5 explains how to use the Artix session manager.

> **Note:** The session manager is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports the session manager.

- Chapter 6 explains how to use Artix switches to bridge between non-Artix enabled applications.
- Chapter 7 explains how to use an Artix service chain.
- Chapter 8 explains how to use the Artix transformer service.
- Chapter 9 explains how to configure high availability (for example, server-side replication and client-side failover).
- Chapter 10 explains how to use the Artix bootstrapping service to locate Artix WSDL contracts and references.

Part III, Integrating Artix

If you are integrating Artix with other products, you may want to read one or more of the following:

- Chapter 11 describes how to deploy Artix into a BEA Tuxedo environment.

> **Note:** Tuxedo integration is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports Tuxedo integration.

- Chapter 12 explains how integrate Artix with Enterprise Management Systems (for example, IBM Tivoli and BMC Patrol).
- Chapter 13 explains how integrate Artix with Computer Associates Web Services Distributed Management (WSDM) software.
- Chapter 14 explains how to use Universal Description, Discovery and Integration (UDDI).

Finding Your Way Around the Library

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

If you are new to Artix

You may be interested in reading:

- Release Notes contains release-specific information about Artix.
- Installation Guide describes the prerequisites for installing Artix and the procedures for installing Artix on supported systems.
- Getting Started with Artix describes basic Artix and WSDL concepts.

To design and develop Artix solutions

Read one or more of the following:

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

To configure and manage your Artix solution

Read one or more of the following:

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

Reference material

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

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

Have you got the latest version?

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

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

Searching the Artix Library

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

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

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

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

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

Online Help

Artix Designer includes comprehensive online help, providing:

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

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

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

Additional Resources

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

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

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

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

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

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

Keying Conventions

This book uses the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, the command prompt is not shown.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the MS-DOS or Windows command prompt.
. . . . . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[]	Brackets enclose optional items in format and syntax descriptions.
{}	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	In format and syntax descriptions, a vertical bar separates items in a list of choices enclosed in {} (braces). In graphical user interface descriptions, a vertical bar separates menu commands (for example, select **File**|**Open**).

Part I

Introduction

In this part

This part contains the following chapters:

CHAPTER 1

Introduction to Artix

Artix enables you to deploy integration solutions that are middleware-neutral.

In this chapter

This chapter contains the following sections:

What is Artix?

Overview

Artix provides a middleware connectivity solution that minimizes invasiveness and prevents an organization from being locked into any one middleware transport. For example, Artix can be used to connect a BEA Tuxedo™ server to a CORBA client. Artix transparently handles the message mapping and transformation between them. The Tuxedo server is unaware that its client is using CORBA. For example, with Artix handling the communication, the client could be changed to an IBM WebSphere MQ™ client without modifying the server.

Scalable infrastructure

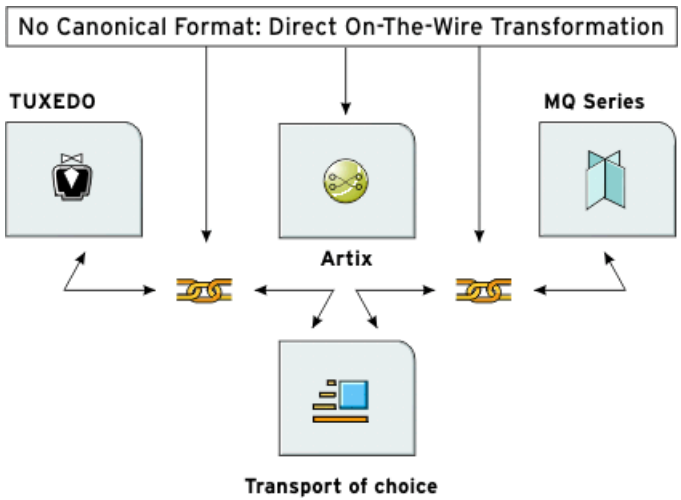
Artix also provides a great deal of configurability because it is built on IONA's Adaptive Runtime architecture (ART). All of Artix's transport and payload format support is encapsulated in individual plug-ins as are all of the services provided with Artix. This allows Artix to be scaled to fit any environment.

Artix message transporting

Artix shields applications from the details of the transports used by applications that they are communicating with, by providing on-the-wire message transformation and mapping. Unlike the approach taken by Enterprise Application Integration (EAI) products, Artix does not use an intermediate canonical format; it transforms the messages only once.

Figure 1 shows a high-level view of how a message passes through Artix. The approach taken by Artix provides a high-level of throughput by avoiding the overhead of making two transformations for each message.

Figure 1: *Artix Message Transporting*



Supported message transports

Artix supports the following message transports:

- HTTP
- BEA Tuxedo
- IBM WebSphere MQ
- Sonic MQ
- IIOP
- TIBCO Rendezvous™
- IIOP Tunnel

Supported payload formats

Artix can automatically transform between the following payload formats:

- G2++
- FML (Tuxedo format)
- GIOP (CORBA format)
- FRL (Fixed Record Length)
- VRL (Variable Record Length)
- SOAP
- TibrvMsg (TIBCO Rendezvous format)

Artix Concepts

Overview

This section explains some of the high-level concepts behind Artix. For example, Artix contracts, and their components, and Artix deployment modes. For more detailed information on Artix, see *Getting Started with Artix*.

Artix contracts

An Artix contract is a WSDL file that defines an interface, and all connection-related information for that interface. Contracts are written using a superset of the standard Web Service Definition Language (WSDL). Following the procedure described by W3C, IONA has extended WSDL to support Artix's advanced functionality, and use of transports and formats other than HTTP and SOAP.

An Artix contract consists of two parts:

Logical

The logical portion of the contract defines the namespaces, messages, and operations that the SAP exposes. This part of the contract is independent of the underlying transports and wire formats. It fully specifies the data structures and possible operation/interaction with the interface. It consists of the `<message>`, `<operation>`, and `<portType>` WSDL tags.

Physical

The physical portion of the contract defines the transports, wire formats, and routing information used to deliver messages to and from SAPs, over the bus. This portion of the contract also defines which messages use each of the defined transports and bindings. The physical portion of the contract consists of the standard `<binding>`, `<port>`, and `<operation>` WSDL tags. It is also the portion of the contract that may contain IONA WSDL extensions.

Deployment modes

Applications that use Artix can be deployed in one of two ways:

Embedded mode

In embedded mode, an application is modified to invoke Artix functions directly and locally, as opposed to invoking a standalone Artix service. This approach is the most invasive to the application, but also provides the highest performance. Embedded mode requires linking the application with Artix-generated stubs and skeletons to connect client and server (respectively) to Artix.

Standalone mode

In standalone mode, Artix runs as a separate process invoked as a service. In this deployment mode, Artix provides a zero-touch integration solution on the application side. When designing a system, you simply generate and deploy the Artix contracts that specify each endpoint. Because a standalone switch is not linked directly with the applications that use it (as in embedded mode), a contract for standalone mode deployment must specify routing information. This is the least efficient of the two modes.

For more detailed information on Artix deployment modes, see Chapter 2.

Advanced Features

Artix also supports the following advanced functionality:

- Message routing based on the operation or port, or port characteristics.
- Transaction support over Tuxedo, WebSphere MQ, and CORBA.
- SSL and TLS support.
- Security support for Tuxedo and WebSphere MQ.
- Deployment with Orbix or higher and Tuxedo.
- Session management.
- Location services.
- High availability (server-side replication and client-side failover).
- Enterprise Management System integration BMC Patrol, IBM Tivoli, and CA WSDM.
- Integration with Computer Associates Web Services Distributed Management (WSDM) software.

CHAPTER 2

Deploying Artix Solutions: An Overview

Artix can be deployed in a number of ways depending on the complexity of your project and your system architecture.

In this chapter

This chapter includes the following sections:

Artix Deployment Modes

Overview

All Artix components have two basic deployment modes:

- Embedded mode
- Standalone mode

Embedded mode

Embedded mode links Artix functionality directly into an application. The application invokes Artix functions directly and locally. Embedded mode requires linking the application with Artix-generated stubs and skeletons.

Standalone mode

Standalone mode places Artix functionality outside of the application space and runs it as a separate process invoked as a service. In standalone mode, Artix is completely described by an Artix contract that specifies which services are to be connected, what transports are in use, and how the services are linked.

These deployment modes can be combined in a number of ways to fit the needs of your applications and environment. For example, you can deploy the Artix session manager and an Artix router in standalone mode while embedding the Artix bus in your client and server applications. Or you could embed all of the Artix components into a server application.

Embedded Application

Overview

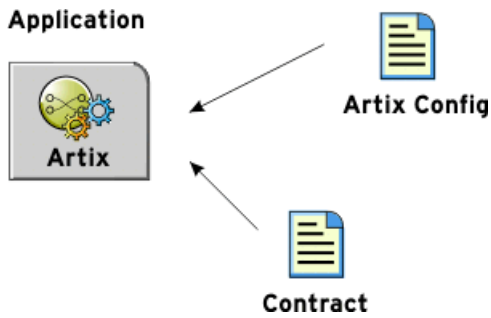
The most basic deployment of Artix is an application with Artix embedded inside. In this scenario, the application can use one of the transports supported by Artix, and the Artix bus is deployed within the application itself. The application gets all of its configuration information from the Artix configuration file and the Artix contract describing the applications interface.

Embedded configuration

Figure 2 shows an application with Artix embedded in it. Artix retrieves its configuration information from two places.

First, when the application first initializes the Artix bus, Artix pulls information about what plug-ins need to be loaded and other runtime information from one of the scopes in the Artix configuration file. Then when the application then registers a servant or instantiates a proxy object, Artix reads in the transport information from the Artix contract describing the application's interfaces.

Figure 2: *Embedded Artix Deployment*



Why use this pattern

This pattern is the most common deployment pattern when deploying a Web service or developing a client application that needs to access a back-end server running on one of the transports Artix supports. Its simplicity makes it easy to configure and deploy. Also, its configuration can be easily modified.

Deploying a simple embedded application

To deploy an application with Artix embedded in it, you would do the following:

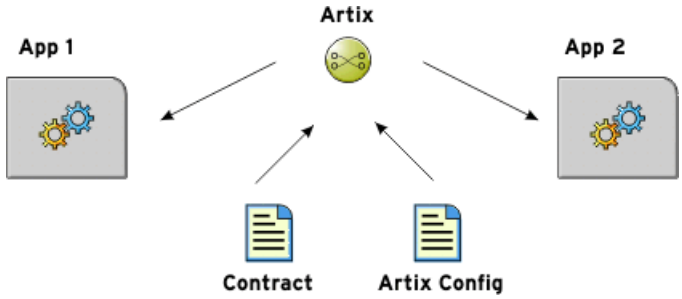
1. Ensure that the system's runtime environment has been properly set up to run Artix applications.
2. Optionally, you can edit the Artix configuration file to create a custom configuration scope for the application. This enables you to control which plug-ins are loaded, the logging level and location, and other runtime features of the bus.
3. Edit the Artix contract for the application to ensure that the transport details are correct for your system.
4. Place the application's contract in the directory where the application will look for it. Typically, this is the same directory as the executable.
5. Run the application with the correct command line arguments.

Standalone Switching Service

Overview

When using Artix as a bridge between applications running on different transports, Artix will often be deployed as a standalone switching service, as shown in Figure 3. When deployed in this scenario, Artix will use at least two transports and route between two or more applications.

Figure 3: *Standalone Artix Deployment*



Standalone configuration

Similar to when Artix is deployed as an embedded piece of an application, Artix loads its configuration from two places. The bus gets its runtime configuration from the Artix configuration file. The transports get their configuration information from the Artix contract describing the interfaces being integrated.

However, both the configuration information and the contract describing the interfaces are more complicated. Artix needs to load more plug-ins to act as a switch. The routing plug-in also needs to be configured to load a contract with the required routing rules, and a control process will need to be configured to ensure that the switch can be shutdown gracefully. The Artix contract will have multiple transport configuration and routing information about how messages are passed through the switch.

Deploying a standalone switch

Artix is configured for the standalone switching service by default. To deploy Artix as a standalone switching service, do the following:

1. Ensure that the system's runtime environment has been properly set up to run Artix applications. See the Artix Configuration Guide.
2. If you are using the switching service as a router, you must add the routing plug-in to the `orb_plugins` list, and configure the location of the WSDL used by the router.
3. Ensure that the Artix contract that describes the your integration contains the correct and routing extensor details (for example, the routing source and destination).
4. Place your application's contract in the directory where the application looks for it. Alternatively, your configuration must specify the location of the router's WSDL relative to where you are running the router.
5. Run the switching service in the Artix container.

For more detailed information, see Chapter 6.

Artix Locator

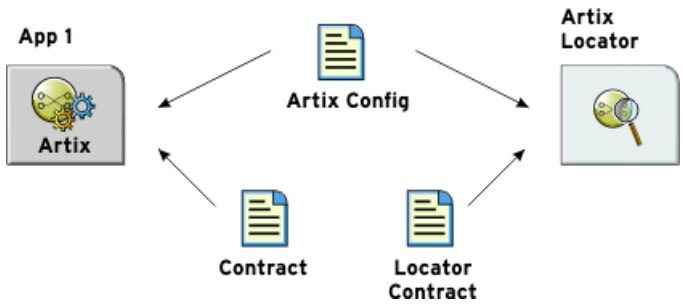
Overview

The Artix locator can be deployed as either a standalone service or an embedded service. The locator differs from Artix applications in that it does not redirect messages and it has a predefined contract.

Standalone locator

Figure 4 shows how system using the Artix locator in standalone mode would look. The locator uses its own contract to configure and advertise on which port it can be contacted. Both the application and the Artix service share a common Artix configuration file. However, they do not share a configuration scope. This style of deploying the locator is beneficial because it does not place additional load on the application. It is best suited for locators that service a number of server processes.

Figure 4: *Standalone Artix Locator*

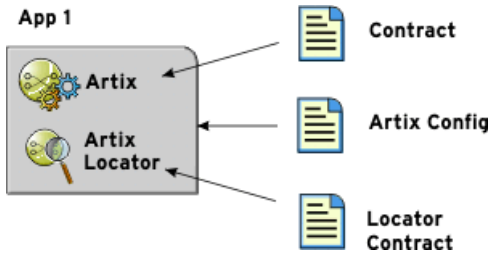


Embedded locator

Figure 5 shows a system in which the Artix locator is embedded in an application. The application still requires two contracts. One for the application and one for the locator. However, when the Artix locator is embedded within an application the application and the locator share a configuration scope.

This style of deployment limits the number of separate processes that need to be deployed on a system. It is useful when the locator instance is only going to be servicing the application that it is embedded in.

Figure 5: *Embedded Artix Locator*



Deploying a standalone Artix locator

To deploy a standalone Artix locator, complete the following steps:

1. Build a standalone Artix locator. This is discussed in Developing Artix Applications with C++.
2. Edit your Artix configuration file to include a configuration scope for your standalone locator.
3. In the locator's configuration scope, ensure that the locator loads the required plug-ins.
4. In the locator's configuration scope, specify the location of the contract for this instance of the locator service.

These steps are discussed in more detail in "Artix Locator" on page 51.

Deploying the Artix locator embedded in an application

To deploy the Artix locator embedded in an application, complete the following steps:

1. Edit your application's configuration scope to specify that the locator plug-ins are loaded at runtime.
2. In the application's configuration scope, specify the location of the contract for locator service instance used by this application.

These steps are discussed in more detail in "Artix Locator" on page 51.

Artix Session Manager

Overview

The Artix session manager enables Web services to engage in statefull communication. It can be deployed as either a standalone service or an embedded service. The session manager, like the Artix locator, has a predefined contract and service specific configuration information.

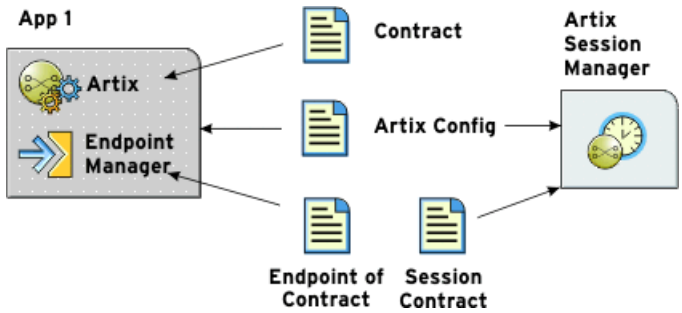
Standalone session manager

Figure 4 shows a system using the Artix session manager in standalone mode. The session manager uses its own contract to configure and advertise how it can be contacted and how its interface is configured.

In addition to the standalone session manager, your application loads an endpoint manager plug-in which also requires a contract defining the interface between the application and the session manager. Both the application and the session manager share a common Artix configuration file.

The standalone session manager instance and the application have separate configuration scopes. However, the configuration information for the endpoint manager is placed in the application's configuration scope. This style of deploying the session manager is best suited for scenarios where the session manager manages a number of endpoints.

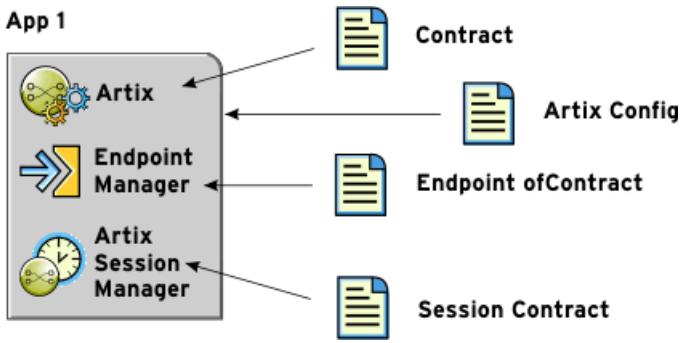
Figure 6: *Standalone Artix Session Manager*



Embedded session manager

Figure 5 shows a system in which the Artix session manager is embedded within an application. The application still requires three contracts. One for the application, one for the session manager, and one for the endpoint manager. However, when the Artix session manager is embedded within an application, the application and the session manager share a configuration scope. This style of deployment limits the number of separate processes that need to be deployed on a system and is useful when the session manger is only servicing the application in which it is embedded.

Figure 7: *Embedded Artix Session Manager*



Deploying a standalone Artix session manager

To deploy a standalone Artix session manager, complete the following steps:

1. Build a standalone Artix session manager. This is discussed in Developing Artix Applications with C++.
2. Edit your Artix configuration to include a configuration scope for your standalone session manager.
3. In the session manager's configuration scope, ensure that the session manager loads the required plug-ins.
4. In the session manager's configuration scope, specify the location of the contract for this instance of the session management service.

5. In the application's configuration scope, edit the `orb_plugins` list to include the required plug-ins for the endpoint manager.
6. In the application's configuration scope, specify the location of the contract for this instance of the endpoint manager.

These steps are discussed in more detail in "Artix Session Manager" on page 69.

Deploying an embedded Artix session manager

To deploy the Artix session manager embedded in an application, complete the following steps:

1. Edit your application's configuration scope to specify that the session manager's plug-ins are loaded at runtime.
2. Edit your application's configuration scope to specify that the endpoint manager's plug-ins are loaded at runtime.
3. In the application's configuration scope, specify the location of the contract for session manager instance used by this application.
4. In the application's configuration scope, specify the location of the contract for this instance of the endpoint manager.

These steps are discussed in more detail in "Artix Session Manager" on page 69.

Part II

Using Artix Services

In this part

This part contains the following chapters:

CHAPTER 3

Artix Container

The Artix container enables you to deploy and manage your services dynamically. For example, you can deploy a new service into a running container, or perform runtime tasks such as start, stop, and list existing services in a container. Artix containers can be used to host C++ or Java services.

In this chapter

This chapter discusses the following topics:

Introduction to the Artix Container

Overview

The Artix container provides a consistent mechanism for deploying and managing Artix services. This section provides an overview the Artix container architecture and its main components.

Artix plug-ins

You can write Artix Web service implementations as C++ or Java plug-ins. An Artix *plug-in* is a code library that can be loaded into an Artix application at runtime.

Artix provides a platform-independent framework for loading plug-ins dynamically, based on the dynamic linking capabilities of modern operating systems (using shared libraries, DLLs, and Java classes).

Benefits

Writing your application as an Artix plug-in means that you need to write less code, and that you can deploy your services into an Artix container. When you deploy your service into a container, this eliminates the need to write your own C++ or Java server mainline. Instead, you can deploy your service by simply passing the location of a generated deployment descriptor to the Artix container's administration client. This provides a powerful programming model where the code is location independent.

In addition, an Artix container retains information about the services that it deploys. This enables the container to reload services dynamically when it restarts.

Main components

The Artix container architecture includes the following main components:

- Artix container server
- Artix container service
- Artix deployment descriptor
- Artix container administration client

How it works

Figure 8 shows how the main Artix container components interact. An Artix container service is deployed into an Artix container server. When an Artix container service is running, you can then use an Artix container administration client to communicate with it at runtime. This client enables you to deploy and manage your services dynamically.

An Artix container service can be run inside any Artix bus. Because it is implemented as an Artix plug-in, it can be loaded into any application. The recommended use it to deploy it into an Artix container server, as shown in Figure 8.

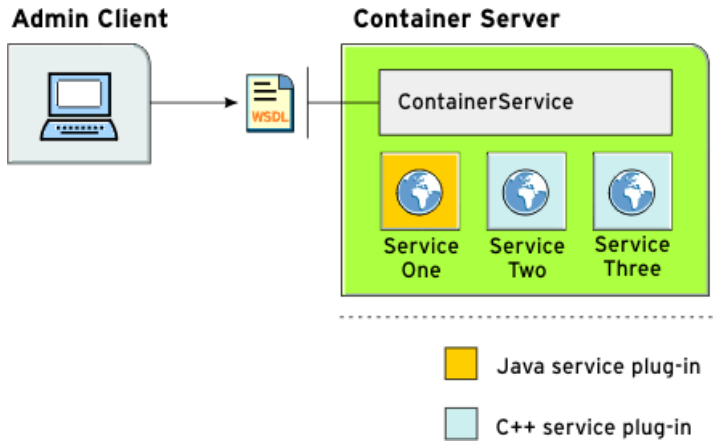


Figure 8: *Artix Container Architecture*

Artix container server

The Artix container server is a simple Artix application that hosts the container service. It consists of a server mainline that initializes a bus and loads the Artix container service, which enables you to remotely deploy and manage your applications.

You can run the Artix container server using the `it_container` command. If your application requires some configuration, you can start the Artix container server with a configuration scope. For more details, see "Running the Artix Container Server" on page 33.

Artix deployment descriptor

When deploying an application into an Artix container, you must pass in a generated Artix deployment descriptor. This is a simple XML file that specifies the details such as:

- Service name.
- Plug-in that implements the service.
- Whether the plug-in is C++ or Java.

You can generate a C++ or Java deployment descriptor by using Artix code generation commands. For more details, see "Generating a Plug-in and Deployment Descriptor" on page 28.

Artix container service

The Artix container service is a remote interface that supports the following operations:

- List all services in the application.
- Stop a running service.
- Start a dormant service.
- Remove a service.
- Deploy a new service.
- Get a reference to a service.
- Get the WSDL for a service.
- Get the URL to a service's WSDL.
- Shut down the container service.

When the Artix container service deploys a new service, it loads the appropriate plug-ins, sets up and activates your service.

The Artix container service assumes that the plug-ins are available in your application environment, so you must ensure that they are in the expected library path. The Artix container service supports C++ and Java applications, provided that they are compiled into plug-ins.

The Artix container service has a WSDL-based interface and so can be used with any binding or transport.

Artix container administration client

Because the Artix container service has a WSDL-based interface with a SOAP/HTTP binding, you can communicate with it using any client. Artix provides a command-line tool that uses the Artix container stub code, and which enables you to manage the container service easily. The Artix container administration client currently supports SOAP/HTTP only.

You can run the Artix container administration client using the `it_container_admin` command. This client makes all the container service operations available through simple command-line options. For more details, see "Running the Artix Container Administration Client" on page 36.

Artix container demos

The following demos in your Artix installation show basic use of the Artix container:

- `...\demos\advanced\container\deploy_plugin`
  This shows how starting with a `.wsdl` file, you can use the `wsdltocpp` or `wsdltojava` command-line tool to generate a C++ or Java plug-in and deployment descriptor. It then shows how to deploy the plug-in into the Artix container.
- `...\demos\advanced\container\deploy_routes`
  This shows how routes are simply advanced services that happen to be implemented by the router plug-in, and whose implementation is just a proxy to a different service. It shows how you can dynamically deploy and manage routes in the Artix container.

Several other advanced Artix demos also use the Artix container, for example:

- `...\demos\advanced\containersecure_container`
- `...\demos\advanced\locator`
- `...\demos\advanced\session_management`
- `...\demos\routing`

Generating a Plug-in and Deployment Descriptor

Overview

Artix services are implemented by C++ or Java plug-ins. When you want to deploy a service into an Artix container, the first step is to generate a plug-in from a `.wsdl` file.

For a C++ service, this generates a dynamic library (Windows), or shared library (UNIX). For a Java service, this generates the Java classes required to implement the plug-in. An XML deployment descriptor is also generated for both C++ and Java service. You can generate a plug-in and deployment descriptor using any of the following commands:

- `wsdltocpp`
- `wsdltojava`
- `wsdd`

Using wsdltocpp

To generate a C++ plug-in library and a deployment descriptor for a specified `.wsdl` file, use the following command:

```
wsdltocpp -n deploy_plugin -impl -server -m NMAKE:library
   -plugin:it_simple_service_cpp_bus_plugin -deployable simple_service.wsdl
```

The `-plugin` and `-deployable` options are the most important. `-plugin` generates a new plug-in, and `-deployable` generates a corresponding deployment descriptor.

The generated plug-in can have an optional name (in this case, `it_simple_service_cpp_bus_plugin`). If a name is specified, the generated plug-in library uses this name. The name is ignored if the `.wsdl` file contains more than one service definition. If no plug-in name is set or ignored, the plug-in name takes the following format: *`ServiceNamePortTypeName`*.

In this example, `-impl` generates the skeleton code for implementing the server defined by the WSDL. `-server` generates code for a server sample implementation, and `-m` generates a makefile. For more details on using the `wsdltocpp` command, see the *Artix Command Line Reference*.

C++ deployment descriptor

The deployment descriptor generated for the example C++ service is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<m1:deploymentDescriptor xmlns:m1="http://schemas.iona.com/deploy">
   <service xmlns:servicens
        ="http://www.iona.com/bus/tests">servicens:SimpleServiceService</service>
   <plugin>
      <name>it_simple_service_cpp_bus_plugin</name>
      <type>Cxx</type>
   </plugin>
</m1:deploymentDescriptor>
```

The `type` element tells the Artix container that this is a C++ service.

Using wsdltojava

To generate a Java plug-in library and a deployment descriptor for a specified `.wsdl` file, use the following command:

```
wsdltojava -impl -server -ant -plugin:it_simple_service_java_bus_plugin
   -deployable simple_service.wsdl
```

The `-plugin` and deployable options are the most important. `-plugin` generates a new plug-in, and `-deployable` generates a corresponding deployment descriptor.

The generated plug-in can have an optional name (in this case, `it_simple_service_java_bus_plugin`). In contrast to C++, the name assigned using the `-plugin` entry only becomes the name of the plugin (as identified in the deployment descriptor). The name of the Java class that implements the plugin factory is derived from the port type name in the WSDL file.

In this example, `-impl` generates the skeleton class for implementing the server defined by the WSDL. `-server` generates code for a server sample implementation, and `-ant` generates an Ant `build.xml` file. For more details on using the `wsdltojava` command, see the *Artix Command Line Reference*.

Java deployment descriptor

The deployment descriptor generated for the example Java service is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<m1:deploymentDescriptor xmlns:m1="http://schemas.iona.com/deploy">
   <service xmlns:servicens
         ="http://www.iona.com/bus/tests">servicens:SimpleServiceService</service>
   <plugin>
      <name>it_simple_service_java_bus_plugin</name>
      <type>Java</type>
     <implementation>com.iona.bus.tests.SimpleServiceServicePluginFactory</implementation>
   </plugin>
</m1:deploymentDescriptor>
```

The `type` element tells the Artix container that this is a Java service.

Using wsdd

For more complex deployment descriptors, you can use the Web services deployment descriptor (`wsdd`) command as an alternative to `wsdltocpp` and `wsdltojava`.

The descriptors generated by `wsdltocpp` and `wsdltojava` do not include all the possible information that descriptors can have—for example, `provider_namespace` (see the `advanced/container/deploy_routes` demo).

The following example uses the `wsdd` command:

```
wsdd -service {http://www.iona.com/test}CustomService
    -pluginName testplugin -pluginType Cxx
```

The full syntax of the `wsdd` command is as follows:

```
wsdd -service QName -pluginName PluginName -pluginType Cxx|Java
   [-pluginImpl Library/ClassName ] [-pluginDir Dir] [-wsdlurl
   WsdlLocation] [-provider ProviderNamespace] [-file
   OutputFile] [-d OutputDir] [-h] [-v] [-verbose] [-quiet]
```

The following arguments are required:

Table 1: *Required Arguments to wsdd*

`-service` *`QName`*	Specifies the name of a service to be deployed.
`-pluginName` *`PluginName`*	Specifies the name that a plug-in is registered as.
`-pluginType Cxx|Java`	Specifies the name of a plug-in type.

The following arguments are optional:

Table 2: *Optional Arguments to wsdd*

`-pluginImpl` *`Library/ClassName`*	Specifies either a library name (`.dll`/`.so`) for a C++ plug-in, or a class name of the plug-in factory for Java plug-ins
`-pluginDir` *`Dir`*	Specifies the location where plug-in library/classes are located. This option, if specified, has no effect on deployment.
`-wsdlurl` *`WsdlLocation`*	Specifies a URL to a service WSDL.
`-provider` *`ProviderNamespace`*	Specifies the provider namespace. Used in the `container/deploy_routes` demo. For example, this can be used by plug-ins to provide servant implementations for more than one service.
`-file` *`OutputFile`*	Specifies the name of the generated descriptor file. The default is `deploy`*`serviceLocalName`*. For example, if `-service {http://www.iona.com/test}CustomService` is used, it is `deployCustomService.xml`
`-d` *`OutputDir`*	The location where a descriptor should be generated.
`-h[elp]`	Displays detailed help information for each option.

Table 2: *Optional Arguments to wsdd*

`-v[ersion]`	Displays the version of the tool.
`-verbose`	Displays output in verbose mode.
`-quiet`	Displays output in quiet mode.

Adding business logic

For both C++ and Java applications, you must still add your business logic code to the servant implementation class.

The supplied Artix demos include a fully implemented servant file instead of the generated file.

Artix deployment descriptors

As well as hosting user-defined services, an Artix container can be used to host IONA services such as the locator. The following is an example generated deployment descriptor for the locator service:

```
<?xml version="1.0" encoding="utf-8"?>
<m1:deploymentDescriptor xmlns:m1="http://schemas.iona.com/deploy">
   <service xmlns:servicens
         ="http://www.iona.com/bus/tests">servicens:SimpleServiceService</service>
   <plugin>
      <name>it_simple_service_java_bus_plugin</name>
      <type>Java</type>
     <implementation>com.iona.bus.tests.SimpleServiceServicePluginFactory</implementation>
   </plugin>
</m1:deploymentDescriptor>
```

For details on deploying the locator in the container, see the *Artix Locator Guide*.

Running the Artix Container Server

Overview

The Artix container server is an Artix server mainline that initializes an Artix bus, and loads the Artix container service.

As well as hosting your own service plugins, the Artix container server can also be used to host well-known Artix services, such as the locator, session manager, router, and so on. You can run as many instances of the Artix container server as your applications require.

Using the it_container command

To run an Artix container server, use the `it_container` command. This has the following syntax:

```
it_container [-s[ervice] Options]  [-d[aemon]] [-p[ort]
   PortNumber] [-publish [-file Filename]] [-deploy
   DeploymentDescriptor] [-deployfolder ] [-v[ersion]] [-h[elp]]
```

`-s[ervice]`	On Windows, runs the container server as a Windows service. Without this parameter, it runs in foreground. See "Running the Container as a Windows Service" on page 46.
`-d[aemon]`	On UNIX, runs the container server as a daemon in the background. Without this parameter, it runs in the foreground.
`-p[ort] PortNumber`	Specifies the port number for the container service.
`-publish [-file Filename]`	Specifies the location to export the container service URL. By default, this is `/ContainerService.url`. You can override the default using `-file`.
`-deploy Descriptor`	Deploys a service using a specified deployment descriptor (for example, at startup). This is instead of deploying with the container service (see "Using the it_container_admin command" on page 36).

`-deployfolder` *Path*	Specifies the location of a local folder to store deployment descriptors. This enables redeployment of existing services on restart (see "Deploying Services on Restart" on page 42).
`-v[ersion]`	Prints version information and exits.
`-h[elp]`	Prints usage summary and exits.

Running the container server in the background

On UNIX, to run a container server in the background, use the `it_container -daemon` command.

If the `-daemon` option is not specified, the container server runs in the foreground of the active command window. This option does not apply on Windows.

Publishing the container service URL in a file

To publish a container service URL, use the `-publish` option, for example:

```
it_container -publish -file
   my_directory/my_container_service.url
```

The `-publish` option tells the container server to publish the container service URL in a local file. This URL can then be later retrieved by the `it_container_admin` command, which uses it to contact the container service, and initialize a container service client proxy.

By default, a `ContainerService.url` file is created in the local directory. Use the `-file` option to override this behavior.

Running the container server on a specified port

To run a container server on a specific port, specify the `-port` option, for example:

```
it_container -port 1111
it_container -port 2222
```

This port is used for the container service. This is also the port for the `wsdl_publish` plug-in. The container administrative client uses `wsdl_publish` to get contracts for the container service and for all other services hosted by the container.

This port number can then be used by a container service administration client when contacting the container server, for example:

```
it_container_admin -port 1111
```

Specifying configuration to the container server

You can run `it_container` without any configuration. This is sufficient for many simple applications. However, if your application requires additional settings, you can start `it_container` with command-line configuration.

For simple applications, the container server loads any plug-ins that you need to instantiate your service, so you do not normally need to configure a plug-ins list, or any other configuration. However, some advanced features may involve launching `it_container` with command-line configuration.

The following example is from the `..demos\advanced\locator` demo and shows running the locator service in the container server:

```
it_container -ORBname demo.locator.service -ORBdomain_name
   locator -ORBconfig_domains_dir ../../etc -publish -file
   ../../etc/ContainerService.url
```

In this example, the locator service picks up specific configuration from its `demo.locator.service` scope. For more details, see the demos for the locator, session manager, and router.

Running the Artix Container Administration Client

Overview

This section explains how to use the Artix container administration client to perform tasks such as deploying a generated plug-in into the Artix container server, and retrieving a service URL. It explains the full syntax of the `it_container_admin` command, which is used to control the Artix container administration client.

Using the it_container_admin command

The full syntax for the `it_container_admin` command is as follows:

`-deploy -file` *`dd.xml`*	Deploys a new service into the container server. This involves loading a plug-in that contains the service implementation. You must specify an Artix deployment descriptor using the `-file` option.
`-listservices`	Displays all services in the application. Shows the state of each service (for example, active, de-activated, or shutting down).
`-startservice -service` *`{Namespace}LocalPart`*	Restarts the specified service that is visible but dormant, or that has been previously stopped.
`-stopservice -service` *`{Namespace}LocalPart`*	Stops the specified running service.
`-removeservice -service` *`{Namespace}LocalPart`*	Removes and undeploys all trace of the specified service from the application.
`-publishreference -service` *`{Namespace}LocalPart`* `[-file` *`Filename`*`]`	Gets a reference to the specified service. The `-file` option publishes the reference to a local file. This can then be used to initialize a client application.

`-publishwsdl -service {Namespace}LocalPart [-file Filename]`	Gets the WSDL for the specified service. The `-file` option publishes the WSDL to a local file. This can then be used to initialize a client application.
`-publishurl -service {Namespace}LocalPart [-file Filename]`	Gets a HTTP URL for the specified service from which you can then download the WSDL. The `-file` option publishes the URL to a local file. This can then be used to initialize a client application.
`-shutdown [-soft]`	Shuts down the entire application. The `-soft` option shuts down gracefully.
`-port ContainerPort`	Contacts the container server on the specified port. See "Running the container server on a specified port" on page 34. This can be used with other options instead of `-container`.
`-host ContainerHostname`	Contacts the container server on the specified host. Defaults to localhost if unspecified. The `-host` option is for use with `-port` only.
`-container File.url`	Runs the specified container service. This can be used with other options instead of `-port` and `-host`.

Note: By default, `it_container_admin` looks in the local directory for the `ContainerService.url` file. If this file is not local, use the `-container` option, or the `-port` and `-host` options, to contact the container.

Deploying the generated plug-in

To deploy a generated plug-in into the container server, use the `-deploy` option, for example:

```
it_container_admin -deploy -file
    ../plugin/deploySimpleServiceService.xml
```

The `-file` option specifies a generated deployment descriptor. This lists the service that this plug-in can provide, the plug-in name, and plug-in type. In this example, the portable C++ plug-in library name is expected to be the same as the plug-in name. The library is expected to be located in the `../plugin` directory.

When a container service loads the plug-in, it registers a servant for the service that is described in the deployment descriptor.

Getting service WSDL

To get the WSDL for a deployed service from the container, use the `-publishwsdl` option, for example:

```
it_container_admin -publishwsdl -service
   {http://www.iona.com/bus/demos}WellWisherService -file
   my_service
```

The `-publishurl` option gets the service's WSDL contract. The `-file` option publishes the URL to a local file. When the client runs, it reads the published WSDL from the local file, and uses it to initialize a client stub, and communicate with a deployed service.

Using the `-publishreference`, `-publishwsdl`, and `-publishurl` options means that you can write WSDL contracts without hard-coded ports, and that your clients will still be able to call against them.

Example 1 shows output from the `-publishwsdl` command that is written to the specified file.

Example 1: *Published WSDL Output*

```
<?xml version='1.0' encoding='utf-8'?>
<definitions name="wellwisher" targetNamespace="http://www.iona.com/demos/wellwisher"
   xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://www.iona.com/demos/wellwisher"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<types xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://www.iona.com/demos/wellwisher"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <schema targetNamespace="http://www.iona.com/demos/wellwisher"
   xmlns="http://www.w3.org/2001/XMLSchema" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
            <element name="responseType" type="xsd:string"/>
    </schema>
</types>

<message name="sayGoodbyeRequest"></message>
<message name="sayGoodByeResponse">
<part element="tns:responseType" name="theResponse"></part>
</message>

<portType name="WellWisher">
<operation name="saySoLong">
<input message="tns:sayGoodbyeRequest" name="saySoLongRequest"></input>
<output message="tns:sayGoodByeResponse" name="saySoLongResponse"></output>
</operation>
</portType>

<binding name="WellWisher_SOAPBinding" type="tns:WellWisher">
<soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http">
</soap:binding>
<operation name="saySoLong">
<soap:operation style="document">
</soap:operation>
<input name="saySoLongRequest">
<soap:body use="literal">
</soap:body>
</input>
<output name="saySoLongResponse">
<soap:body use="literal">
</soap:body>
</output>
</operation>
</binding>
```

Example 1: *Published WSDL Output*

```
<service name="WellWisherService">
<port binding="tns:WellWisher_SOAPBinding" name="WellWisherPort">
<soap:address location="http://10.2.4.52:2223/wellwisher">
</soap:address>
</port>
</service>
</definitions>
```

Getting a service URL

To get a URL for a deployed service from the container service, use the `-publishurl` option, for example:

```
it_container_admin -publishurl -service
   {http://www.iona.com/bus/tests}SimpleServiceService -file
   my_service
```

The `-publishurl` option gets a URL to the service's WSDL contract. The `-file` option publishes the URL to a local file. When the client runs, it reads the published WSDL URL from the local file, and uses it to initialize a client stub, and then communicate with a deployed service.

Listing deployed services

To display a list of the services in your application, use the `-listservices` option, for example:

```
it_container_admin -port 2222 -listservices
{http://www.iona.com/demos/wellwisher}WellWisherService ACTIVATED
{http://www.iona.com/demos/greeter}GreeterService ACTIVATED
```

This example shows the output listed under the `it_container_admin -listservices` command. The `ACTIVATED` state indicates that both services are running. In this example, the `-port` option is used to contact a container server that was already started on port `2222`.

Stopping deployed services

To stop a currently deployed service, use the `-stopservice` option, for example:

```
it_container_admin -port 2222 -stopservice -service
   {http://www.iona.com/demos/wellwisher}WellWisherService
```

This following example shows the output from `-listservices` after the service has been stopped.

```
it_container_admin -port 2222 -listservices
{http://www.iona.com/demos/wellwisher}WellWisherService DEACTIVATED
{http://www.iona.com/demos/greeter}GreeterService ACTIVATED
```

The `WellWisherService` is now listed as `DEACTIVATED`.

Specifying configuration to the administration client

You can run `it_container_admin` without any configuration. This is sufficient for most simple applications. However, if your application requires additional settings, you can start `it_container_admin` with command-line configuration.

For simple applications, the container service loads any plug-ins that you need to instantiate your service, so you do not normally need to configure a plug-ins list, or any other configuration. However, some advanced features may involve launching `it_container_admin` with command-line configuration.

The following example shows shutting down the locator service using the `it_container_admin -shutdown` option:

```
it_container_admin -ORBdomain_name locator -ORBconfig_domains_dir
   ../../etc -container ../../etc/ContainerService.url -shutdown
```

For more details, see the demos for the locator, session manager, and router.

Deploying Services on Restart

Overview

The Artix container can be configured to retain information about the services that it has deployed. This enables it to reload services automatically on restart. This ability to remember deployed services is known as *persistent deployment*.

To enable persistent deployment, you must configure the container to use a local folder to store deployment descriptors. These descriptors specify what the container should deploy at startup. The container ensures that this folder accurately reflects what is deployed in case of a restart.

How it works

To reload services that have been deployed by the container service before shutdown, the container persists all deployment descriptors when processing new deployment requests. The container needs to know the location of a local folder where deployment descriptor files are saved to, and where to read them from on restart.

The container finds the location of this folder from either:

- A command-line argument passed to the container.
- A configuration variable in a configuration file.

Note: Command-line arguments take precedence over configuration variables.

At startup, the container looks in the configured deployment folder and deploys the contents of the folder. It deploys all services that it finds in the folder where possible. If any deployment fails, the container fails to start.

Persistent deployment modes

You can configure the deployment descriptor folder for either read/write or read-only deployment.

Dynamic read/write deployment

In this case, the container adds and removes files from the deployment folder dynamically as services are deployed or removed from the container. When a call to deploy a service is made, a descriptor file is added to the folder. When a call to remove a service is made, a descriptor file is removed, and the service is not redeployed upon restart.

Read-only deployment

The deployment descriptor folder can also be used as a read-only initialization folder that predeploys the same required set of services after every restart.

When a deployment folder is read-only, the container predeploys the same set of services on restart. No deployment descriptors are removed from, or saved into, a read only deployment folder by the container.

By making a deployment folder read-only, you can share deployment descriptors between multiple container instances. In this scenario, you can enable a single container instance to modify the contents of this folder, and all container instances are affected after restart.

Enabling dynamic read/write deployment

You can enable a read/write deployment folder using the following command-line arguments:

```
it_container -deployfolder ../etc
```

Alternatively, you can set the following variable in a configuration file:

```
plugins:container:deployfolder="../etc";
```

This means that the `../etc` folder is used for predeploying services and persisting new descriptors.

Enabling read-only deployment

You can enable a read-only deployment folder using the following command-line arguments:

```
it_container -deployfolder -readonly ../etc
```

Alternatively, you can set the following variables in a configuration file:

```
plugins:container:deployfolder="../etc";
plugins:container:deployfolder:readonly="true";
```

This means that the `../etc` folder is used for predeploying services only.

Predeploying a service on startup

The `it_container` command also provides a `-deploy` argument, which can be used to predeploy a single service on startup, for example:

```
it_container -deploy deployCORBAService.xml
```

The `-deploy` and `-deployfolder` arguments can be used together, for example:

```
it_container -deploy deployMyService.xml -deployfolder ../etc
```

This means that `MyService` identified by `deployMyService.xml`, and all services identified by descriptors in the `../etc` folder, are deployed. The `deployMyService.xml` that is specified using the `-deploy` argument is not copied into a deployment folder. If you wish to copy a descriptor to the deployment folder, use the following command:

```
it_container_admin -deploy -file deployMyService.xml
   -deployfolder -deployfolder ../etc
```

Naming conventions

The Artix container uses the following format when persisting deployment descriptors into files:

```
deploy<i>LocalServiceName</i>.xml
```

You should follow the same pattern when generating custom descriptors where possible. The container expects that all files in the deployment folder that have the `.xml` extension are valid deployment descriptors.

By default, deployment descriptors generated by Artix tools use the name of the service's local part. If you have two services with the same local part but different namespaces, you should use the `wsdd -file` option to avoid the name clashing. For more details, see "Using wsdd" on page 30.

Removing a service

When using a read/write deployment folder, you can remove a service by calling `it_container_admin -removeservice` on a running container. For example:

```
it_container_admin -removeservice -service
   {http://www.iona.com/bus/tests}SimpleServiceService
```

Alternatively, you can remove the deployment descriptor file from the folder. Both of these approaches ensure that the container does not reload the service at startup.

When using a read-only folder, removing a service using `-removeservice` does not prevent it from being redeployed after a restart. Only removing a descriptor file from the folder prevents it from being redeployed.

Note: Copying or removing files from the deployment folder has no impact if the container is already running. The container cannot react to these events. The contents of the folder is read once at startup. This only applies to services that are started using deployment descriptors.

Warnings and exceptions

It is possible that using different descriptors might lead to the container attempting to deploy the same service twice.

In this case, the container logs a warning message and proceeds with deploying other services. An exception is thrown if an attempt to deploy the same service is made from an administration console.

Further information

For a working example of persistent deployment, see the following Artix demo:

`.../demos/advanced/container/deploy_plugin`

Running the Container as a Windows Service

Overview

On Windows, you can install instances of the Artix container server as a Windows service. By default, this means that the installed container will start up when your system restarts.

This feature also enables you to manage the container using the Windows service controls. For example, you can start or stop a container using the Windows Control Panel, or Windows `net` commands, such as `net stop` *`ServiceName`*.

Format of service names

When a container is installed as a Windows service, the container name takes the following format in the Windows registry:

```
ITArtixContainer ServiceName
```

For example, if you call your service `test_service`, the name generated by the install command that appears in the registry is:

```
ITArtixContainer test_service
```

This name is stored under the following entry in the registry:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services
```

Setting your environment variables

Before installing the Artix container as a Windows service, you must ensure that your system environment variables have been set correctly, and that your machine has rebooted. These steps can be performed either when installing Artix, or at any time prior to installing the container as a Windows service.

Your environment variables enable the container to find all the information it needs on restart. They must be set as follows:

Environment Variable	Setting
IT_PRODUCT_DIR	Your Artix installation directory (for example, c:\iona). **Note:** This is needed only if your PATH specifies %IT_PRODUCT_DIR%, instead of the full path to any Artix directories.
PATH	Should include the following: • Any C++ plug-ins that will be deployed by the container. • *InstallDir*\bin and *InstallDir*\artix\3.0\bin. • The JRE libraries, *JDKInstallDir*\jre\bin and *JDKInstallDir*\jre\bin\server.
CLASSPATH	Should include the following: • Any Java plug-ins that will be deployed by the container. If the plug-in is packaged in a JAR, you must list the .jar file. If .class files are used, only the directory needs to be listed. • The Artix runtime JAR, *InstallDir*\artix\3.0\lib\artix-rt.jar • *InstallDir*\etc and *InstallDir*\artix\3.0\etc. • Your JDK/JRE runtime JAR (for example, *JDKInstallDir*\jre\lib\rt.jar).

Note: If you used Microsoft Visual C++ 7.1 to create your service plug-in, include the following in your PATH, in this order:

```
InstallDir\bin\vc71;InstallDir\bin;InstallDir\artix\3.0\bin\vc71
;InstallDir\artix\3.0\bin
```

Installing a container

To install a container as a Windows service, use the `it_container -service install` command:

```
it_container -service install [-ORBParamName [ParamValue]]
   -displayname Name -svcName ServiceName
```

These parameters are described as follows:

`-ORB`*`ParamName`*	Represents zero or more `-ORB`*`ParamName`* command-line options (for example, `-ORBlicense_file`). These specify the location of the Artix license file, domain name, configuration directory, or ORB name. These values must be specified either as command-line parameters or environment variables. However, specifying on the command line allows easier deployment of multiple `it_container` instances as multiple Windows services.
`-displayname`	Specifies the name that is displayed in the Windows **Services** dialog (select **Start**|**Settings**|**Control Panel**|**Application Tools**|**Services**). The `-displayname` parameter is required.
`-svcName`	Specifies the service name that is listed in the Windows registry (select **Start**|**Run**, and type `regedit`). The `-svcName` parameter is required.

In addition to the `-service install` parameters, the following `it_container` parameters also apply:

`-port`	Specifies the port that the container will run on (see "Running the container server on a specified port" on page 34). This parameter is required.
`-deployfolder`	Specifies a local folder to store deployment descriptors. This enables redeployment on startup (see "Deploying Services on Restart" on page 42). This parameter is optional.

Example command

The following example shows all the parameters needed to install a container instance as a Windows service:

```
it_container -service install -ORBlicense_file c:\InstallDir\etc\licenses.txt
   -ORBconfig_dir c:\InstallDir\artix\3.0\etc -ORBdomain_name artix
   -displayName "My Test Service" -svcName my_test_service -port 2222
   -deployfolder C:\deployed_files
```

If you do not set your license file, domain name, and configuration directory, as environment variables, you must set them as `-ORB`*`ParamName`* entries (the recommended approach). The `-ORBname` parameter is optional.

Example service

The installed Windows service is listed in the **Services** dialog, as shown in Figure 9.

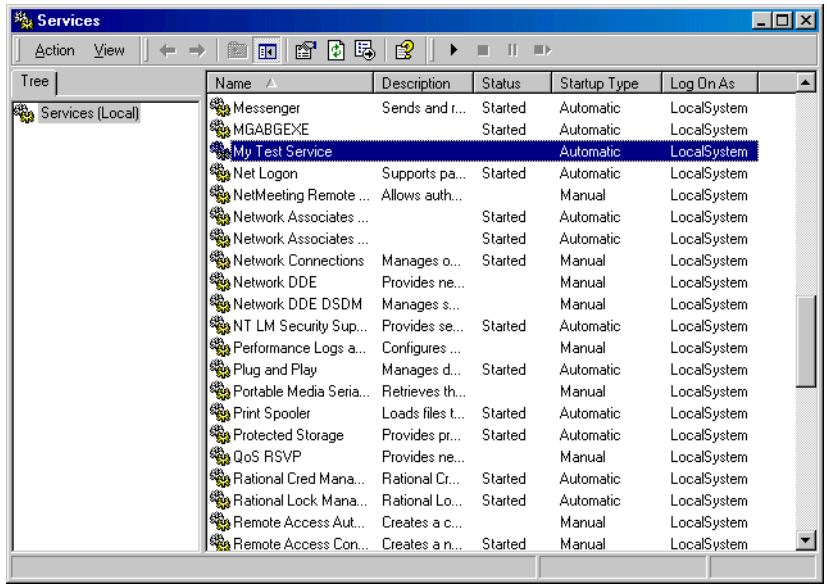


Figure 9: *Installed Windows Service*

Clicking on `My Test Service` displays the properties shown in Figure 10.

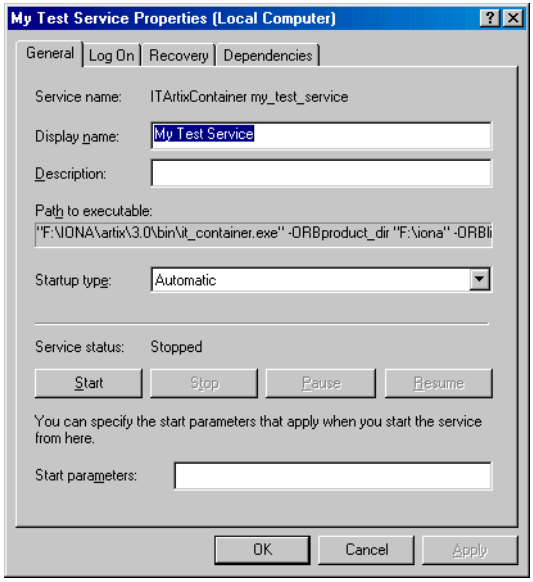


Figure 10: *Service Properties*

After running the `it_container -service install` command, you must start the services manually. However, when your computer is restarted, the installed services are configured to restart automatically.

Uninstalling a container

To uninstall a container as a Windows service, use the `it_container uninstall` command.

```
it_container -service uninstall -svcName ServiceName
```

For example:

```
it_container -service uninstall -svcName my_artix_test
```

CHAPTER 4

Artix Locator

The Artix locator service is a service endpoint registry. It enables Artix servers to publish their references for dynamic lookup by Artix clients.

In this chapter

This chapter discusses the following topics:

Overview of the Artix Locator

Overview

A system with many servers can not afford the overhead of manually propagating each server's contact information to the clients that need to contact them. Given the large number of clients and the distributed nature of enterprise deployments, the time required to propagate contact details, and the room for error, are too great. Over time, hardware upgrades, machine failures, or site reconfiguration will require you to move servers and repeat the exercise of propagating the server's information to all clients.

The Artix locator service isolates clients from changes in a server's contact information. The Artix WSDL contract defines how the client contacts the server, and contains the address of the Artix locator. The locator provides the client with a reference to the server. Servers are automatically registered with the locator when they start-up.

Locator service components

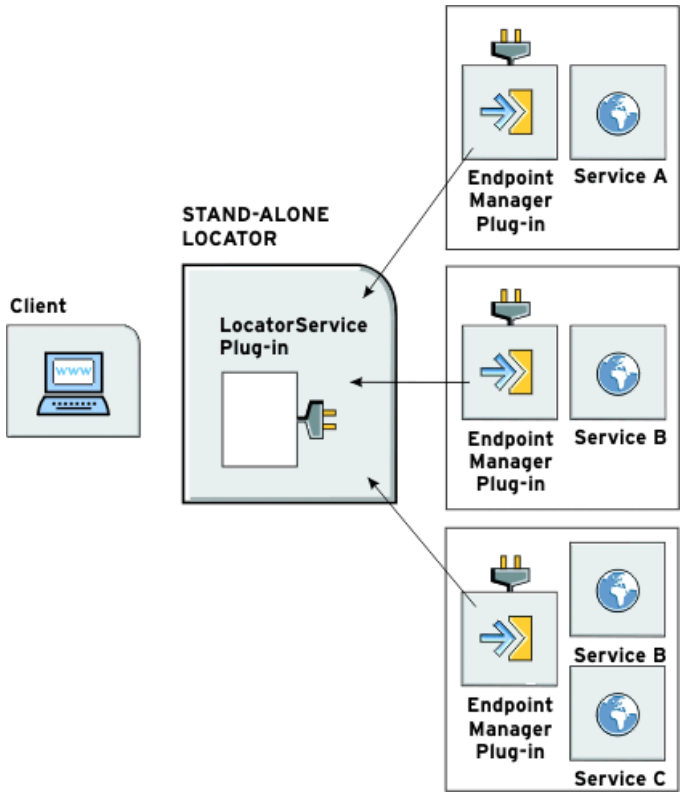
The Artix locator's functionality is built into two plug-ins:

Locator Service Plug-in (`service_locator`)	This is the central service plug-in. It accepts service registrations, performs service look-ups, hands out references to clients who request them, and controls the load balancing of service groups.
Locator Endpoint Manager Plug-in (`locator_endpoint`)	This is the portion of the locator that resides in a registered service. It registers its location with the locator service plug-in, and monitors the health of the locator service plug-in to ensure fault tolerance.

How do the plug-ins interact?

Figure 11 shows an overview of how the locator plug-ins are deployed in an Artix system. In this example, the locator service plug-in is deployed in a standalone service, however, it can be deployed in any Artix process.

Figure 11: *The Locator Plug-ins*



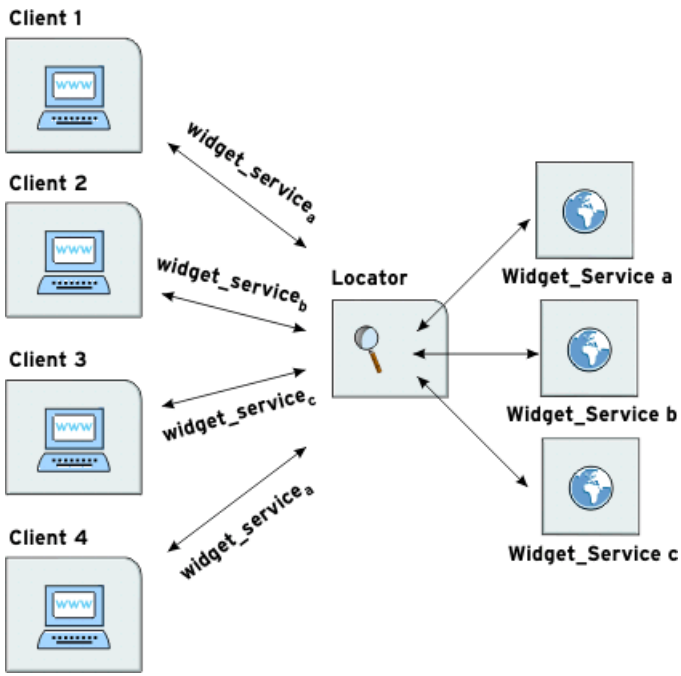
The locator endpoint manager plug-ins are deployed in the server processes that contain services registered with the locator. A server process can host two services, (for example, *Service C* and *Service D* in Figure 11), but the process can have only one endpoint manager. The endpoint manager plug-ins are in constant communication with the locator service plug-in to report on endpoint health, and to check on the health of the locator service.

Load balancing

The locator also provides load balancing functionality. When a group of services register with the locator using the same service name, the locator considers the services as a single service and uses a round-robin load balance algorithm to hand out references to the separate instances.

Figure 12 shows a simplified example. When each client makes a request for `widget_service` (using the locator), the locator cycles through the pool of registered `widget_service` instances. For example, when `client4` makes a request, the locator starts handing out references from the top of the pool (`widget_service a`).

Figure 12: *Locator Load Balancing*



Note: The client must first contact the locator to resolve a reference, and use that reference to instantiate a proxy (see "Obtaining References from the Locator" on page 63). For simplicity, this is not shown the diagram.

In addition, services can also implement their own load balancing internally by using calls to the Artix locator service that temporarily remove them from the pool of active references.

Deploying the Locator

Overview

The Artix locator is implemented using Artix plug-ins. This means that any Artix application can host the locator service by loading the `service_locator` plug-in. However, it is recommended that you deploy the locator using the Artix container. For information on the Artix container, see Chapter 3.

Configuring the locator to run in the container

To configure the locator to run in the Artix container, ensure that the `service_locator` plug-in is included in the locator's configuration scope, for example:

```
locator_demo {
   ...
   locator{
      orb_plugins = ["xmlfile_log_stream", "service_locator"];
    };
```

The `service_locator` plug-in implements the locator service functionality.

The `soap` and `at_http` plug-ins are loaded automatically when the process parses the locator's WSDL contract. The `iiop_profile`, `giop`, and `iiop` plugins are not required when you are using the Artix container, or an Artix process.

Configuring a dynamic port

By default, the locator is configured to deploy on a dynamic port. In the default locator WSDL contract, the addressing information is as follows:

Example 2: *Locator Service on Dynamic Port*

```
<service name="LocatorService">
  <port binding="ls:LocatorServiceBinding" name="LocatorServicePort">
   <soap:address
   location="http://localhost:0/services/locator/LocatorService"/>
 </port>
</service>
```

The highlighted part shows the address. The `localhost:0` port means that when you activate the locator service, the operating system assigns a port dynamically on startup.

Configuring a fixed port

There are two ways of deploying the locator on a well-known fixed port. You can either edit the default `locator.wsdl` contract, or you can create a new `locator.wsdl` contract for your application.

Editing the default locator contract

To edit the default `locator.wsdl` contract, perform the following steps:

1. Open the `locator.wsdl` contract in any text editor. This is in the following directory:

```
InstallDir\artix\3.0\wsdl\locator.wsdl
```

2. Edit the `soap:address` attribute at the bottom of the contract to specify the correct address. Example 3 shows a modified locator service contract entry. The highlighted part has been modified to point to the desired address.

Example 3: *Locator Service on Fixed Port*

```
<service name="LocatorService">
  <port name="LocatorServicePort" binding="ls:LocatorServiceBinding">
    <soap:address
   location="http://localhost:8080/services/locator/LocatorService"/>
  </port>
</service>
```

Creating a new locator contract

To create a new `locator.wsdl` contract, perform the following steps:

1. Copy the default `locator.wsdl` contract to another location, and open it in any text editor.
2. Edit the `soap:address` attribute at the bottom of the contract to specify the correct address. Example 3 shows a modified locator service contract entry. The highlighted part has been modified to point to the desired address.

3. In your configuration file, in the application's scope, add a new `bus:initial_contract:url:locator` variable that points to your edited WSDL contract. For example:

```
bus:initial_contract:url:locator = "c:\myapp/wsdl/locator.wsdl";
```

   The default `bus:initial_contract:url:locator` variable is in the global scope, which ensures that every application has access to the contract. Specifying a new contract in your application scope overrides the global locator contract for your application.

When the locator has been correctly configured, it can be started like any other application. The only difference is that the session manager must be started before any servers that need to register with it.

Deploying the locator in the container

The locator can be started in the Artix container, just like any other application. To deploy the default locator in the container, perform the following steps:

1. Run the locator in the Artix container, for example:

```
it_container -ORBname demo.locator.service -ORBdomain_name
   locator -ORBconfig_domains_dir ../../etc -publish
```

2. Ask the container to publish the live version of the locator WSDL that you use to initialize your clients. For example:

```
it_container_admin -container ../../etc/ContainerService.url
   -publishwsdl -service
   {http://ws.iona.com/locator}LocatorService -file
   ..\..\etc\locator-activated.wsdl
```

   This retrieves the locator's activated WSDL contract. This is the contract in which `0` ports are dynamically updated with the actual port that the service runs on. In this example, `it_container_admin` writes the contract to the `locator-activated.wsdl` file in the `etc` subdirectory

3. Finally, you must ensure that your clients use the updated WSDL file at runtime.

Deploying the locator in the container on a fixed port

Alternatively, you can use the `-port` option to specify that the container runs a service on a fixed port. For example:

```
it_container -port 9000 -ORBname demo.locator.service
   -ORBdomain_name locator -ORBconfig_domains_dir ../../etc
   -publish
```

In this example, any services that run in the container, and have default contracts with a port of `0`, will not use port `9000`.

You can manually update the WSDL used by your client to `9000`, or you can publish the WSDL from the container using `it_container_admin` with the `-publishwsdl` option, shown in "Deploying the locator in the container" on page 58.

Note: Instead of using the container to deploy the locator, you can also do this by creating a custom Artix service mainline. For details, see Developing Artix Applications with C++.

Shutting down the locator

To shut down the locator, use the Artix container's shutdown option, for example:

```
it_container_admin -ORBdomain_name locator -ORBconfig_domains_dir
   ../../etc -container ../../etc/ContainerService.url -shutdown
```

Registering a Server with the Locator

Overview

A server does not need to have its implementation changed to work with the Artix locator. All that is required is that the server be configured to load the correct plug-ins, and to reference the correct locator contract.

If you require more fine-grained control, you can filter the service endpoints that are registered.

Configuring the server

Any server that wishes to register itself with the locator must load the `locator_endpoint` plug-in. The `locator_endpoint` plug-in enables the server to register with the running locator. The following shows the configuration scope of a server that registers with the locator service.

```
my_server
{
  orb_plugins = ["xmlfile_log_stream", "locator_endpoint"];
 };
```

`my_server` provides its services using SOAP over IIOP, so in addition to the locator plug-in, it loads these plug-ins transparently.

Using a copy of locator.wsdl

If you are using a copy of the default locator contract to specify a fixed port, the server configuration must also specify the location of the contract. For example:

```
bus:initial_contract:url:locator="c:\my_server/locator.wsdl";
```

This is not necessary if you are using a dynamic port, or have updated the default contract with a fixed port. The global `bus:initial_contract:url:locator` setting is used instead.

For more details, see the Artix Configuration Guide.

Filtering service endpoints

By default, any service activated in an Artix bus that loads the `locator_endpoint` plug-in is automatically registered in the locator.

However, you may not want every service registered or exposed to the locator. Artix enables you to filter the endpoints that are registered by the locator endpoint manager. You can do this by explicitly including or excluding endpoints.

Excluding endpoints to be registered

If there are a small number of endpoints that you want to be filtered, you can explicitly exclude those endpoints from the locator.

For example, if you do not want to register the container service, but want to register all the endpoints that are activated in that container, use the following setting:

```
plugins:locator_endpoint:exclude_endpoints =
   ["{http://ws.iona.com/container}ContainerService"];
```

For more details, see the `located_router` demo.

Including endpoints to be registered

If you have a small number of endpoints that you want to be added, and want to filter out all others, you can use the filter that lists the included endpoints.

For example, if you only want to register the session manager, but not any of the endpoints that it manages, use the following setting:

```
plugins:locator_endpoint:include_endpoints =
   ["{http://ws.iona.com/sessionmanager}SessionManagerService"];
```

For more details, see the `located_sessions` demo.

Note: Combining the exclude and include configuration variables is ambiguous. If you do this, the application will fail to initialize.

Filtering endpoints using wildcards

When filtering endpoints, you can also wildcard your service names. This enables you to filter based on a specified namespace.

You can specify that all services defined in a particular namespace should be included. For example:

```
plugins:locator_endpoint:include_endpoints =
   ["{http://www.sample.com/finance}*"];
```

Alternatively, you can use the following setting to exclude all services defined in a particular namespace:

```
plugins:locator_endpoint:exclude_endpoints =
   ["{http://www.sample.com/finance}*"];
```

Server registration

When a properly configured server starts up, it automatically registers with the locator that is specified by the contract pointed to by `bus:initial_contract:url:locator`.

You can register multiple instances of the same server with a locator. The locator generates a pool of references for the server type. When clients make a request for a server, the locator supplies references from this pool using a round-robin algorithm. For more information on load balancing see "Load Balancing" on page 66.

Obtaining References from the Locator

Overview

Unlike servers, clients must be specifically written to work with the Artix locator. There are three steps a client must take to obtain a server reference from the Artix locator:

1. Instantiate a proxy for the locator service.
2. Look up the desired server's endpoint using the locator service proxy.
3. Create a proxy for the desired server using the returned endpoint.

The examples shown in this section are C++. For Java examples, see Developing Artix Applications in Java.

Instantiating a locator service proxy

Before a client can invoke any of the look up methods on the locator service, it must create a proxy to forward requests to the running locator. To do this, the client creates an instance of `LocatorServiceClient` using the locator service QName.

Example 4 shows how to instantiate a locator service proxy. The parameters used to create the locator service's QName (`LocatorService` and `http://ws.iona.com/locator`) should never be modified.

Example 4: *Instantiating a Locator Service Proxy*

```
// C++
QName ls_service_name(
     "",
     "LocatorService",
     "http://ws.iona.com/locator");

Reference locator_ref;

if (!bus->resolve_initial_reference(ls_service_name,locator_ref))
{
   // error handling here
}

locator_client = new LocatorServiceClient(locator_ref);
```

Example 4 uses the `resolve_initial_reference()` method. This is the recommend way of instantiating a locator service proxy. This approach is location independent, so the client is not concerned about the specific location details of the WSDL contract.

Alternatively, you can hard code the filename, for example:

```
// C++
QName locator_service_name("", "LocatorService",
                           "http://ws.iona.com/locator");
locator_proxy = new LocatorServiceClient("locator.wsdl",
                                         locator_service_name,
                                         "LocatorServicePort");
```

However, specifying the filename is less flexible and scalable than using `resolve_initial_reference()`, and requires more information, for example:

- The locator service contract name (`locator.wsdl`).
- The locator service QName.
- The port name used in the locator service contract, `LocatorServicePort`.

For more information on Artix proxy constructors, see Developing Artix Applications with C++ or Developing Artix Applications in Java.

Looking up a server's endpoint

After instantiating a locator service proxy, a client can then look up servers using the proxy's `lookup_endpoint()` method. This method has the following signature:

```
//C++
void lookup_endpoint(
    const IT_Bus::QName &service_qname,
    IT_Bus::Reference &service_endpoint
```

`service_qname`	Contains the input QName of the server that the client is looking up.
`service_endpoint`	Contains the reference to the server that is output. If the locator cannot find a registered instance of the requested server, `lookup_endpoint()` returns an `endpointNotExistFault` exception.

For example:

```
//C++
locator_client->lookup_endpoint(service_name,endpoint);
```

Creating a server proxy

The client uses the reference returned in the output parameter of `lookup_endpoint()` to instantiate a server proxy for making requests on the requested server. To instantiate the proxy, use the correct proxy class for the server you have requested and pass the return value of the returned `service_endpoint` to the proxy class' constructor.

For example, the client code for creating a proxy server from the results of the look up performed in "Looking up a server's endpoint" is as follows:.

```
// C++
SimpleServiceClient simple_client(endpoint);
```

Further information

The code examples in this section are from the `SimpleServiceClientSample.cxx` file in the following directory:

`InstallDir``\artix\3.0\demos\advanced\locator\cxx\client`

For more information on writing Artix client code, see Developing Artix Applications with C++.

Load Balancing

Overview

The Artix locator provides a lightweight mechanism for balancing workloads among a group of servers. When a number of servers with the same service name register with the Artix locator, it automatically creates a list of the references and hands out the references to clients using a round-robin algorithm. This process is invisible to both the clients and the servers.

Starting to load balance

When the locator is deployed and your servers are properly configured, you must bring up a number of instances of the same service. This can be accomplished by one of the following methods depending on your system topology:

- Create an Artix contract with a number of ports for the same service and have each server instance start up on a different port.
- Create a number of copies of the Artix contract defining the service, change the port information so each copy has a separate port address, and then bring up each server instance using a different copy of the Artix contract.

Note: The locator determines if it is part of a group using the name specified in the `<service>` tag of the server's Artix contract. If you are using the Artix locator to load balance, your services should be associated with the same binding and logical interface.

As each server starts up, it automatically registers with the locator. The locator recognizes that the servers all have the same service name specified in their Artix contracts and creates a list of references for these server instances.

As clients make requests for the service, the locator cycles through the list of server instances to hand out references.

Fault Tolerance

Overview

Enterprise deployments demand that applications can cleanly recover from occasional failures. The Artix locator is designed to recover from the two most common failures faced by a look-up service:

- Failure of a registered endpoint.
- Failure of the look-up service.

Endpoint failure

When an endpoint gracefully shuts down, it notifies the locator that it is no longer available. The locator removes the endpoint from its list so it cannot give a client a reference to a dead endpoint. However, when an endpoint fails unexpectedly, it can not notify the locator, and the locator can unknowingly give a client an invalid reference causing the failure to cascade.

To decrease the risk of passing invalid references to clients, the locator service occasionally pings all of its registered endpoints to see if they are still running. If an endpoint does not respond to a ping, the locator removes that endpoint's reference.

You can adjust the interval between locator service pings by setting the `plugins:locator:peer_timeout` configuration variable. The default setting is 4 seconds. For more information, see the *Artix Configuration Guide*.

Service failure

When the locator service fails, all the references to the registered endpoints are lost and the active endpoints are no longer registered with the locator. If the locator misses its ping interval, the endpoints periodically attempt to reregister with the locator until they are successful. This ensures that the active endpoints reregister with the locator when it restarts.

You can adjust the interval at which the endpoint pings the locator by setting the `plugins:session_endpoint_manager:peer_timeout` configuration variable. The default setting is 4 seconds. For more information, see the *Artix Configuration Guide*.

Replicating the locator

Replicating the locator service involves specifying the same configuration used for any other Artix service. There are also some additional configuration variables apply to the locator.

For full details of all the configuration steps, see Chapter 9.

CHAPTER 5

Artix Session Manager

The Artix session manager enables you to manage service resources (for example, how clients access a group of services).

In this chapter

This chapter discusses the following topics:

Introduction to Artix Session Management

Overview

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

Note: The Artix session manager is not available in all editions of Artix. Please check the conditions of your Artix license to see whether your installation supports the Artix session manager.

The two main session manager plug-ins are:

Session manager service plug-in (`session_manager_service`)	This is the central service plug-in. It accepts and tracks service registration, hands out sessions to clients, and accepts or denies session renewal.
Session manager endpoint plug-in (`session_endpoint_manager`)	This is the portion of the session manager that resides in a registered service. It registers its location with the service plug-in, and accepts or rejects client requests based on the validity of their session headers. This provides control over the allowable duration for a session and the maximum number of concurrent sessions allowed for each group.

The session manager also includes a simple policy plug-in:

Session manager simple policy plug-in (`sm_simple_policy`)	This provides control over the allowable duration for a session and the maximum number of concurrent sessions allowed for each group.

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

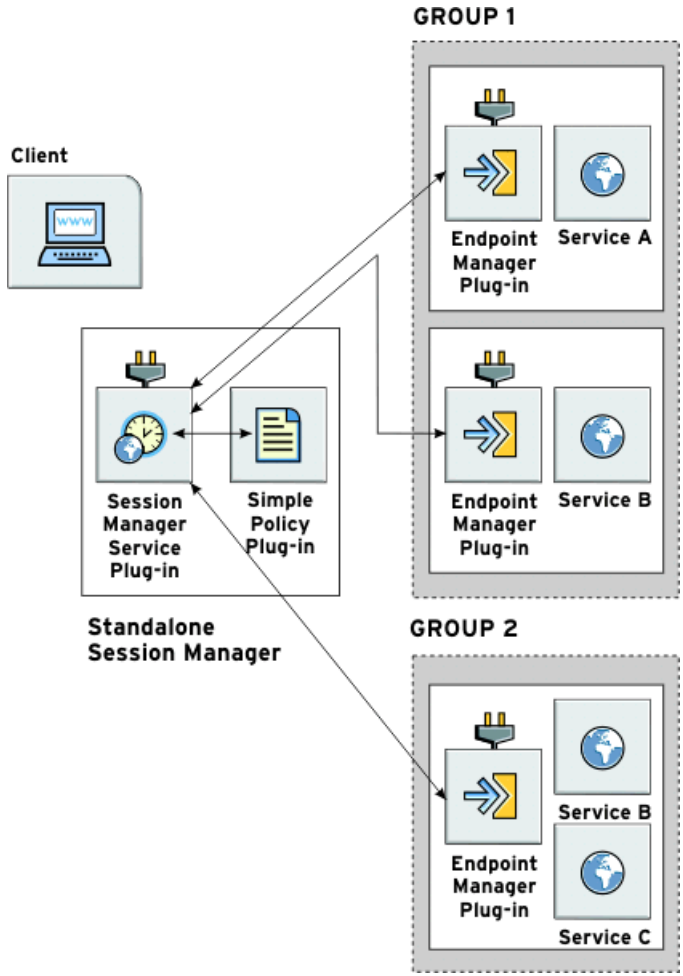
How do the plug-ins interact?

Figure 13 shows how the session manager plug-ins are deployed in an Artix system. The session manager service plug-in and the policy callback plug-in are both deployed into the same Artix bus process.

In this example, while these plug-ins are deployed into a standalone service, they can be deployed in any Artix process. The session manager service plug-in and the policy plug-in interact to ensure that the session manager does not hand out sessions that violate the policies established by the policy plug-in. The simple policy plug-in makes all the decisions on which sessions are permitted. The session manager service queries this policy on all decisions. Artix provides a default implementation in the simple policy plug-in. However, you can also write your own policy plug-in.

The endpoint manager plug-ins are deployed into the server processes that contain session managed services. A process can host two services (for example, *Service C* and *Service D* in Figure 13), but the process can have only one endpoint manager. The endpoint manager plug-ins are in constant communication with the session manager service plug-in to report on endpoint health. They also receive information on new sessions that have been granted to the managed services, and check on the health of the session manager service.

Figure 13: *The Session Manager Plug-ins*



What are sessions?

The session manager controls access to services by handing out *sessions* to clients who request access to the services. A session is a pass that provides access to the services in a specific group for a specific time.

For example, the following process is used when a client application wants to use the services in a group named `sales`:

1. The client application asks the session manager for a session with the `sales` group.
2. The session manager checks and see if the `sales` group has an available session, and if so, it returns a session ID and the list of `sales` service references to the client.
3. The session manager notifies the endpoint managers in the `sales` group that a new session has been issued. It also supplies a new session ID, and a duration for which the session is valid.
4. When the client makes requests on the services in the `sales` group, it must include the session information as part of the request.
5. The endpoint manager for the services checks the session information to ensure it is valid. If it is, the request is accepted. If it is not, the request is rejected.
6. If the client wants to continue using the `sales` services beyond the duration of its lease, the client must ask the session manager to renew its session before the session expires.
7. Finally, when a client's session has expired, it must request a new one.

What are groups?

The Artix session manager does not pass out sessions for each individual service that is registered with it. Instead, services are registered as part of a *group*, and sessions are handed out for the group. A group is a collection of services that are managed as one unit by the session manager. While the session manager does not specify that the services in a group must be related, it is recommended that the endpoints have some relationship.

A service's group affiliation is controlled by the configuration scope in which it is run. To change a service's group, edit the following value in the process configuration scope:

```
plugins:session_endpoint_manager:default_group
```

This specifies the default group name for the services instantiated by the server.

Deploying the Session Manager

Overview

The Artix session manager is implemented using Artix plug-ins. This means that any Artix application can host the session manager's core functionality by loading the `session_manager_service` plug-in. However, it is recommended that you deploy the session manager using the Artix container. For information on the Artix container, see Chapter 3.

This section describes how to specify configuration for the Artix session manager. You may also need to create a copy of `session-manager.wsdl`. This is the contract that describes the session manager and contains the session manager's contact information.

Configuring the session manager service to run in the container

To configure the session manager service to run in the Artix container, ensure that the `session_manager_service` plug-in is included in the session manager service configuration scope, for example:

```
session_management {
  ...
   sm_service{
      orb_plugins = ["xmlfile_log_stream", "session_manager_service"];
  ...
    };
```

The `session_manager_service` plug-in implements the session manager service functionality.

The `soap` and `at_http` plug-ins are loaded automatically when the process parses the session manager's WSDL contract. The `iiop_profile`, `giop`, and `iiop` plugins are not required when you are using the Artix container, or an Artix process.

If no policy plug-in is specified, the `sm_simple_policy` plug-in is loaded automatically by the session manager service. If you wish to customize settings for this policy, see "Configuring the Simple Policy Plug-in" on page 82.

Configuring a dynamic port

By default, the session manager is configured to deploy on a dynamic port. In the default session manager WSDL contract, the addressing information is as follows:

Example 5: *Session Manager Service on Dynamic Port*

```
<service name="SessionManagerService">
  <port name="SessionManagerPort" binding="sm:SessionManagerBinding">
    <soap:address
   location="http://localhost:0/services/sessionManagement/sessionMa
   nagerService"/>
  </port>
</service>
```

The highlighted part shows the address. The `localhost:0` port means that when you activate the session manager service, the operating system assigns a port dynamically on startup.

Because the port is assigned dynamically, you must ensure that your clients obtain a reference to the updated contract when it is assigned a port. For details of using the Artix locator to do this, see "Obtaining References from the Locator" on page 63.

Configuring a fixed port

There are two ways of deploying the session manager on a well-known fixed port. You can either edit the default `session-manager.wsdl` contract, or you can create a new `session-manager.wsdl` contract for your application.

Editing the default session manager contract

To edit the default `session-manager.wsdl` contract, perform the following steps:

1. Open the `session-manager.wsdl` contract in any text editor. This is located as follows:

```
InstallDir\artix\3.0\wsdl\session-manager.wsdl
```

2. Edit the `soap:address` attribute at the bottom of the contract to specify the correct address. Example 6 shows a modified session manager service contract entry. The highlighted part has been modified to point to the desired address.

Example 6: *Session Manager Service on Fixed Port*

```
<service name="SessionManagerService">
  <port name="SessionManagerPort" binding="sm:SessionManagerBinding">
    <soap:address
   location="http://localhost:8080/services/sessionManagement/session
   ManagerService"/>
  </port>
</service>
```

Creating a new session manager contract

To create a new `session-manager.wsdl` contract, perform the following steps:

1. Copy the default `session-manager.wsdl` contract to another location, and open it in any text editor.
2. Edit the `soap:address` attribute at the bottom of the contract to specify the correct address. Example 6 shows a modified session manager service contract entry. The highlighted part has been modified to point to the desired address.
3. In your configuration file, in the application's scope, add a new `bus:initial_contract:url:sessionmanager` variable that points to your edited WSDL contract. For example:

```
bus:initial_contract:url:sessionmanager =
   "c:\myapp/wsdl/session-manager .wsdl";
```

   The default `bus:initial_contract:url:sessionmanager` variable is in the global scope, which ensures that every application has access to the contract. Specifying an a new contract in your application scope overrides the global session manager contract for your application.

When the session manager has been correctly configured, it can be started like any other application. The only difference is that the session manager must be started before any servers that need to register with it.

Deploying the session manager in the container

The session manager can be started in the Artix container like any other application. To deploy the default session manager in the container, perform the following steps:

1. Run the session manager in the Artix container, for example:

```
it_container -ORBname demos.session_management.sm_service
   -ORBdomain_name session_management -ORBconfig_domains_dir
   ../../etc -publish
```

2. Ask the container to publish the live version of the session manager WSDL that you use to initialize your clients. For example:

```
it_container_admin -container ../../etc/ContainerService.url
   -publishwsdl -service
   {http://ws.iona.com/sessionmanager}SessionManagerService
   -file ..\..\etc\sessionmanager-activated.wsdl
```

   This retrieves the session manager's activated WSDL contract. This is the contract in which `0` ports are dynamically updated with the actual port that the service runs on. In this example, `it_container_admin` writes the contract to the `sessionmanager-activated.wsdl` file in the `etc` subdirectory

3. Finally, you must ensure that your clients use the updated WSDL file at runtime.

Deploying the session manager in the container on a fixed port

Alternatively, you can use the `-port` option to specify that the container runs a service on a fixed port. For example:

```
it_container -port 9000 -ORBname demo.sessionmanager.service
   -ORBdomain_name session_management -ORBconfig_domains_dir
   ../../etc -publish
```

In this example, any services that run in the container, and have default contracts with a port of `0`, will not use port `9000`.

You can manually update the WSDL used by your client to `9000`, or you can publish the WSDL from the container using `it_container_admin` with the `-publishwsdl` option, shown in "Deploying the session manager in the container" on page 78.

Note: Instead of using the container to deploy the session manager, you can also do this by creating a custom Artix service mainline. For details, see Developing Artix Applications with C++.

Shutting down the session manager

To shut down the session manager, use the Artix container's shutdown option, for example:

```
it_container_admin -shutdown
```

Registering a Server with the Session Manager

Overview

Services that wish to be managed by the session manager must register with a running session manager. Servers instantiating these services must load the `session_endpoint_manager` plug-in and correctly configure themselves. They do not require any special application code.

When registered with a session manager, the services only accept requests that contain a valid session header. All clients wishing to access the services must be written to support session managed services.

Configuring the server

Any server hosting services that are to be managed by the session manager must load the `session_endpoint_manager` plug-in. The `session_endpoint_manager` enables the server to register with a running session manager.

Example 7 shows the configuration scope of a server that hosts services managed by the session manager.

Example 7: *Server Configuration Scope*

```
acme_server
{
  orb_plugins = ["xmlfile_log_stream", "session_endpoint_manager"];

   plugins:session_endpoint_manager:default_group="acme_group";
 };
```

In this example, a server loaded into the `acme_server` configuration scope is managed by the session manager at the location specified in your `session-manager.wsdl` contract. Its endpoint manager comes up at the address specified in `session-manager.wsdl`. In this example, by default, all services instantiated by the server belongs to the `acme_group` session manager group.

Using a copy of session-manager.wsdl

If you are using a copy of the default session manager contract to specify a fixed port, your server configuration must also specify the location of the contract. For example:

```
bus:initial_contract:url:sessionmanager =
    "c:\myapp/wsdl/session-manager .wsdl";
```

This is not necessary if you are using a dynamic port, or have updated the default contract with a fixed port. The global `bus:initial_contract:url:locator` setting is used instead.

Server registration

When a properly configured server starts up, it automatically registers with the session manager specified by the contract pointed to by `bus:initial_contract:url:sessionmanager`.

Configuring the Simple Policy Plug-in

Overview

The Artix session manager provides a simple policy callback plug-in (`sm_simple_policy`). This enables you to control the allowable duration for a session, and the maximum number of concurrent sessions allowed for each group.

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

Session properties

The simple policy plug-in provides default values for the following session properties:

- Maximum number of concurrent sessions in a given group (default is `1`).
- Maximum allowed timeout for a session (default is `600` seconds).
- Minimum allowed timeout for a session (default is `5` seconds).

You can override these defaults using the following configuration variables:

```
plugins:sm_simple_policy:max_concurrent_sessions
plugins:sm_simple_policy:min_session_timeout
plugins:sm_simple_policy:max_session_timeout
```

All values must be non-negative. You must configure the `max_session_timeout` to be greater than or equal to `min_session_timeout`. A value of `0` means an unlimited timeout.

Further information

For more information on working with sessions, see Developing Artix Applications with C++. This explains how to perform tasks such as the following:

- Instantiate a proxy for the session management service.
- Start a session for a service's group using the session manager proxy.
- Obtain the list of endpoints available in the group.
- Create a service proxy from one of the endpoints in the group.
- Build a session header to pass to the service.
- Invoke requests on the endpoint using the proxy.
- Renew the session as needed.
- End the session using the session manager proxy.

Fault Tolerance

Overview

Enterprise deployments demand that applications can cleanly recover from occasional failures. The Artix session manager is designed to recover from the two most common failures:

- Failure of a registered endpoint.
- Failure of the session manager itself.

Endpoint failure

When an endpoint gracefully shuts down, it notifies the session manager that it is no longer available. The session manager removes the endpoint from its list so it can not give a client a reference to a dead endpoint. However, when an endpoint fails unexpectedly, it cannot notify the session manager and the session manager can unknowingly give a client an invalid reference causing the failure to cascade.

To decrease the risk of passing invalid references to clients, the session manager occasionally pings all of its registered endpoint managers to see if they are still running. If an endpoint manager does not respond to a ping, the session manager removes that endpoint manager's references.

You can adjust the interval between session manager pings by setting the `plugins:session_manager:peer_timeout` configuration variable. The default setting is 4 seconds. For more information, see the *Artix Configuration Guide*.

Service failure

If the session manager fails, all of the references to the registered services are lost and the active services are no longer be registered. After the session manager misses its ping interval, the endpoint managers periodically attempt to reregister with the session manager until they are successful. This ensures that the active services reregister with the session manager when it restarts.

You can adjust the interval between the endpoint manager's pings of the session manager by setting the configuration variable `plugins:session_endpoint_manager:peer_timeout`. The default setting is 4 seconds. For more information, see the *Artix Configuration Guide*.

CHAPTER 6

Artix Switches

An Artix switch acts as a bridge between non-Artix enabled applications. The Artix standalone service performs transport switching, message routing, and middleware bridging.

In this chapter

This chapter discusses the following topics:

The Artix Switch

Overview

Artix switches are a minimally invasive means of connecting applications that use different communication transports and message formats. It does not require that any Artix-specific code be compiled or linked into existing applications. A switch is created by loading the Artix routing plug-in into the Artix container server. For more information on the Artix container server, see Chapter 3.

How it works

An Artix switch is a routing daemon that listens for traffic on access points specified in an Artix contract. It re-directs messages based on the routing rules that you provide, and performs any transport switching and message formatting needed for the receiving application. Neither application is aware that its messages are being intercepted by Artix and no application development is required.

Note: Artix requires that services being integrated use equivalent message layouts. For example, a service expecting a `long` cannot be sent a `float`.

The switch's behavior is controlled by a combination of an Artix contract and the Artix configuration file.

For more information on Artix contracts see *Designing Artix Solutions*. For more information on configuring the Artix runtime, see the Artix Configuration Guide.

Deployment patterns

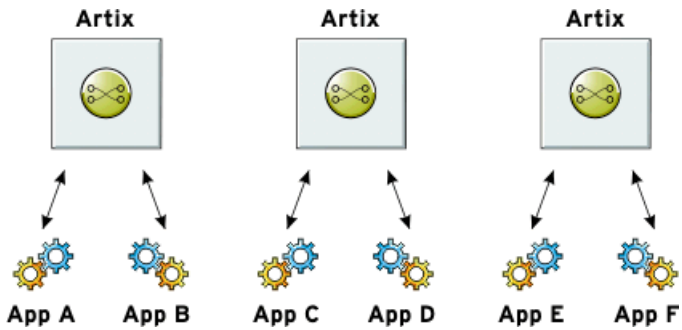
Artix switched can be deployed in a number of ways. Two common deployment patterns are:

- Deploying multiple switches—each bridging between two applications.
- Deploying one switch to bridge between all applications in a domain.

Deploying multiple switches—each bridging between two applications.

This approach simplifies designing integration solutions and provides faster processing of each message (shown in Figure 14). Using this approach, the Artix contract describing the interaction of the applications is simpler because it contains only the logical interfaces shared by the two applications, the bindings for each payload format, and the routing rules.

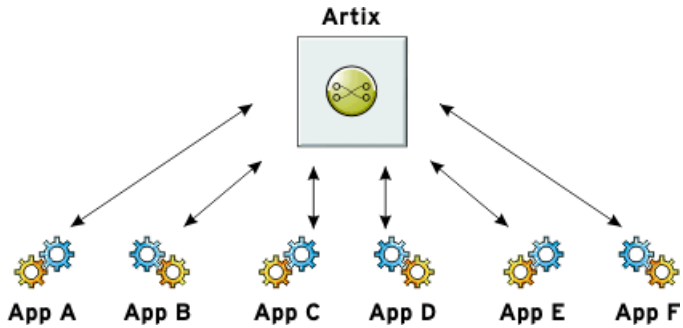
Figure 14: *Using Multiple Artix Switches*



Because most applications use only one network transport, the number of ports is minimal and the routing rules are simple. Keeping the contract simple also enhances the performance of each switch because it has less processing to do. In this approach, each switch's resource usage can also be limited by tailoring its configuration to optimize the switch for the integration task for which it is responsible.

Deploying one switch to bridge between all applications in a domain. This approach limits the number of external services required in your deployment environment (shown in Figure 15). This can simplify monitoring and installation of deployments. It also reduces the number of moving parts in an integration solution.

Figure 15: *Using a Single Artix Switch*



Configuring a Switch

Overview

Because Artix's routing functionality is implemented as a plug-in, you can make any Artix application a switch by adding routing rules to its contract and adding the routing plug-in to its `orb_plugins` list. To deploy a switch that does not require any modifications to existing applications, you can deploy a switch using the Artix container server (`it_container`). For more information, see Chapter 3.

orb_plugins list

An Artix switch must include the `routing` plug-in name in its `orb_plugins` list:

WARNING: The routing plug-in must always be the last plug-in listed in `orb_plugins`.

In addition, if one of the transports used by the client or server is IIOP, the following plug-ins are also required:

- `iiop_profile`
- `iiop`
- `giop`

Plug-ins related to all bindings, and all transports other than IIOP, are not required. These are loaded automatically when the routing plug-in parses the WSDL file.

Switch plug-in settings

You need to add configuration information to point the switch to the contract, or contracts, that contain the routing information it is to use. This is done with the `plugins:routing:wsdl` variable. This variable specifies the contracts the switch will parse for routing rules. The contract names are relative to the location from which the Artix switch is started.

For example, if a switch's configuration contained the following entry:

```
plugins:routing:wsdl=["route1.wsdl", "../route2.wsdl",
                      "/artix/routes/route3"];
```

The switch would expect that `route1.wsdl` was located in the directory in which it was started and `route2.wsdl` was located one directory level higher.

CHAPTER 7

Deploying a Service Chain

Artix provides a chain builder that enables you to create a series of services to invoke as part of a larger process.

In this chapter

This chapter includes the following sections:

The Artix Chain Builder

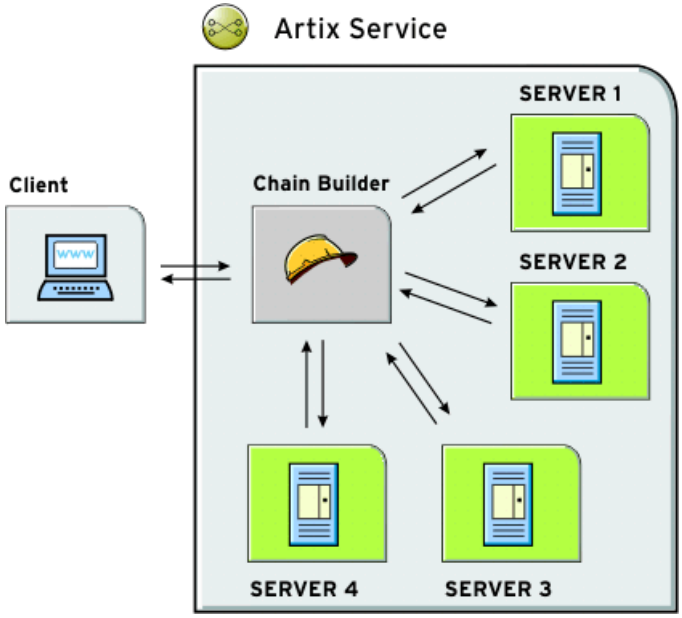
Overview

The Artix chain builder enables you to link together a series of services into a multi-part process. This is useful if you have processes that require a set order of steps to complete, or if you wish to link together a number of smaller service modules into a complex service.

Chaining services together

For example, you may have four services that you wish to combine to service requests from a single client. You can deploy a service chain like the one shown in Figure 16.

Figure 16: *Chaining Four Servers to Form a Single Service*



In this scenario, the client makes a single request and the chain builder dispatches the request along the chain starting at `Server1`. The chain builder takes the response from `Server1` and passes that to the next endpoint in the chain, `Server2`. This continues until the end of the chain is reached at `Server4`. The chain builder then returns the finished response to the client.

The chain builder is implemented as an Artix plug-in so it can be deployed into any Artix process. The decision about which process that you deploy it in depends on the complexity of your system, and also how you choose to allocate resources for your system.

Assumptions

To make the discussion of deploying the chain builder as straightforward as possible, this chapter assumes that you are deploying it into an instance of the Artix container server. However, the configuration steps for configuring and deploying a chain builder are the same no matter which process you choose to deploy it in.

Configuring the Artix Chain Builder

Overview

To configure the Artix chain builder, complete the following steps:

1. Add the chain builder's plug-in to the process' `orb_plugins` list.
2. Configure all the services that are a part of the chain.
3. Configure the chain so that it knows what servants to instantiate and the service chain for each operation implemented by the servant.

Adding the chain builder in the orb_plugins list

Configuring the application to load the chain builder's plug-in requires adding it to the application's `orb_plugins` list. The plug-in name for the chain builder is `ws_chain`. Example 8 shows an `orb_plugins` list for a process hosting the chain builder.

Example 8: *Plug-in List for Using a Web Service Chain*

```
orb_plugins={"ws_chain", "xml_log_stream"};
```

Configuring the services in the chain

Each service that is a part of the chain, and the client that makes requests through the chain service, must be configured in the chain builder's configuration scope. For example, you must supply the service name and the location of its contract.

This provides the chain builder with the necessary information to instantiate a servant that the client can make requests against. It also supplies the information needed to make calls to the services that make up the chain.

To configure the services in the chain, use the configuration variables in Table 3.

Table 3: *Artix Service Configuration*

Variable	Function
`bus:qname_alias:`*`service`*	Specifies a service name using the following syntax: `{`*`service_qname`*`}`*`service_name`* For example: `{http://www.mycorp.com}my_service`
`bus:initial_contract:url:`*`service`*	Specifies the location of the contract describing this service. The default is the current working directory.

Configuring the service chains

The chain builder requires you to provide the following details

- A list of services that are clients to the chain builder.
- A list of operations that each client can invoke.
- Service chains for each operation that the clients can invoke.

Specifying the servant list

The first configuration setting tells the chain builder how many servants to instantiate, the interfaces that the servants must support, and the physical details of how the servants are contacted. You specify this using the `plugins:chain:servant_list` variable. This takes a list of service names from the list of Artix services that you defined earlier in the configuration scope.

Specifying the operation list

The second part of the chain builder's configuration is a list of the operations that each client to the chain builder can invoke. You specify this using `plugins:chain:`*`endpoint`*`:operation_list` where *`endpoint`* refers to one of the endpoints in the chain's service list.

`plugins:chain:`*`endpoint`*`:operation_list` takes a list of the operations that are defined in <operation> tags in the endpoint's contract. You must list all of the operations for the endpoint or an exception will be thrown at runtime. You must also be sure to enter a list of operations for each endpoint specified in the chain's service list.

Specifying the service chain

The third piece of the chain builder's configuration is to specify a service chain for every operation defined in the endpoints listed in `plugins:chain:servant_list`. This is specified using the `plugins:chain:`*`endpoint`*`:`*`operation`*`:service_chain` configuration variable. The syntax for entering the service chains is shown in Example 9.

Example 9: *Entering a Service Chain*

```
plugins:chain:endpoint:operation:service_chain=["op1@endpt1", "op2@endpt2", ..., "opN@endptN"];
```

For each entry, the syntax is as follows:

endpoint	Specifies the name of an endpoint from the chain builder's servant list
operation	Specifies one of the operations defined by an `operation` entry in the endpoints contract. The entries in the list refer to operations implemented by other endpoints defined in the configuration.
opN	Specifies one of the operations defined by an `operation` entry in the contract defining the service specified by *endptN*. The operations in the service chain are invoked in the order specified. The final result is returned back to the chain builder which then responds to the client.

Instantiating proxy services

The chain invokes on other services, and for this reason, it instantiates proxy services. It can instantiate proxies when the chain servant starts (the default), or later, when a call is made. The following configuration variable specifies to instantiate proxy services when a call is made:

```
plugins:chain:init_on_first_call ="true";
```

This defaults to `false`, which means that proxies are instantiated when the chain servant starts. However, you might not be able to instantiate proxies when the chain servant is started because the servant to call has not started. For example, this applies when using the Artix locator or UDDI.

Configuration example

Example 4 shows the contents of a configuration scope for a process that hosts the chain builder.

Table 4: *Configuration for Hosting the Artix Chain Builder*

```
colaboration {
  orb_plugins = ["ws_chain"];

  bus:qname_alias:customer= "{http://needs.com}POC";
  bus:initial_contract:url:customer = "order.wsdl";

  bus:qname_alias:pm = "{http://ORBSrUs.com}prioritize";
  bus:initial_contract:url:pm = "manager.wsdl";

  bus:qname_alias:designer = "{http://ORBSrUs.com}design";
  bus:initial_contract:url:designer = "designer.wsdl";

  bus:qname_alias:builder = "{http://ORBSrUs.com}produce";
  bus:initial_contract:url:builder = "engineer.wsdl";

  plugins:chain:servant_list = ["customer"];

  plugins:chain:customer:requestSolution:service_chain =
    ["estimatePriority@pm", "makeSpecification@designer",
     "buildORB@builder"];
};
```

Configuration guidelines

When Web services are chained, the following rules must be obeyed:

- The input type of the chain service (in this example, `customer`) must match the input of the first service in the chain (`pm`).
- The output type of a previous service in the chain must match the input type of the next service in the chain.
- The output type of the last service in the chain must match the output of the chain service.
- One configuration entry must exist for each operation in the `portType` of the chain service (for example, `customer`). This simple example shows only one entry, and the `portType` for the customer endpoint has only one operation (`requestSolution`).
- The chain service can invoke only on services that have one port.
- Finally, not all operations must be configured in the chain, only those that are invoked upon. This means that no check is made when all operations are mapped to a chain. If a client invokes on an unmapped operation, the chain service throws a `FaultException`.

CHAPTER 8

Deploying the Artix Transformer

Artix provides an XSLT transformer service that can be configured to run as a servant process that replaces an Artix server.

In this chapter

This chapter discusses the following topics:

The Artix Transformer

Overview

The Artix transformer provides a means of processing messages without writing application code. The transformer processes messages based on XSLT scripts and returns the result to the requesting application. XSLT stands for *Extensible Stylesheet Language Transformations*.

These XLST scripts can perform message transformations, such as concatenating two string fields, reordering the fields of a complex type, and truncating values to a given number of decimal places. XSLT scripts can also be used to validate data before passing it onto a Web service for processing, and a number of other applications.

Deployment Patterns

The Artix transformer is implemented as an Artix plug-in. Therefore, it can be loaded into any Artix process. This makes it extremely flexible in how it can be deployed in your environment. If the speed of calls or security is an issue, the transformer can be loaded directly into an application. If you need to spread resources across a number of machines, the transformer plug-in can be loaded in a separate process.

There are two main patterns for deploying the Artix Transformer:

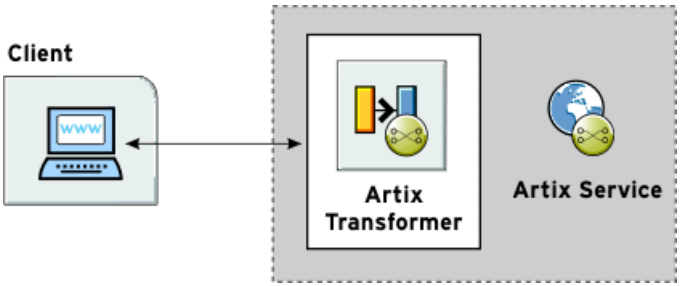
- Standalone deployment
- Deployment as part of a chain

Standalone deployment

The first pattern is to deploy the transformer by itself. This is useful if your application is doing basic data manipulation that can be described in an XSLT script. The transformer replaces the server process and saves you the cost of developing server application code. This style of deployment can also be useful for performing data validation before passing requests to a server for processing.

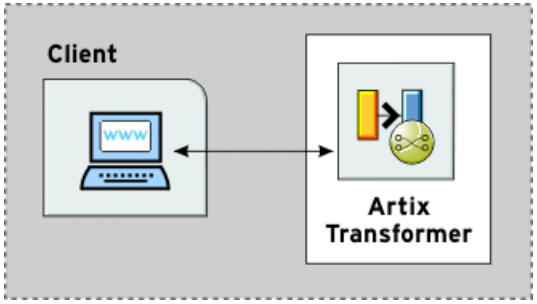
The most straightforward way to deploy the transformer is to deploy it as a separate servant process hosted by the Artix container server. When deployed in this way the transformer receives requests from a client, processes the message based on supplied XSLT scripts, and replies with the results of the script. In this configuration, shown Figure 17, the transformer becomes the server process in the Artix solution.

Figure 17: *Artix Transformer Deployed as a Servant*



You can modify the deployment pattern shown in Figure 17 by eliminating the Artix container server and having your client directly load the transformer's plug-in as shown in Figure 18. This saves the overhead of making calls outside of the client process to reach the transformer However, it can reduce the overall efficiency of your system if the transformer requires a large amount of resources to perform its work.

Figure 18: *Artix Transformer Loaded by Client*

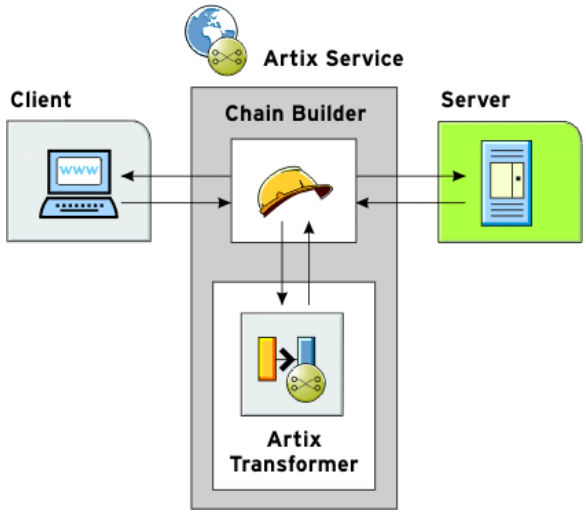


Deployment as part of a chain

The second pattern is to deploy the Artix transformer as part of a Web service chain controlled by the Web Service Chain Builder. This deployment is useful if you need to connect legacy clients to updated servers whose interfaces may have changed or are connecting applications that have different interfaces. It can also be useful for a range of applications where data transformation is needed as part of a larger set of business logic.

Figure 19 shows an example of this type of deployment where the transformer and the chain builder are both hosted by the Artix container server. The chain builder directs the requests to the transformer which transforms messages. When the transformer returns the processed data, the chain builder then passes it onto the server. In this example, the server returns the results to the client without further processing, but the results can also be passed back through the transformer. Neither the client nor the server need to be aware of the processing.

Figure 19: *Artix Transformer Deployed with the Chain Builder*



You could modify this deployment pattern in a number of ways, depending on how you allocate resources. For example, you can configure the client process to load the chain builder and the transformer. You can also load the chain builder and the transformer into separate processes.

Standalone Deployment

Overview

To deploy an instance of the Artix transformer you must first decide what process is hosting the transformer's plug-in. You must then add the following to the process configuration scope:

- The transformer plug-in, `xslt`.
- An Artix endpoint configuration to represent the transformer.
- The transformer's configuration information.

Updating the orb_plugins list

Configuring the application to load the transformer requires adding it to the application's `orb_plugins` list. The plug-in name for the transformer is `xslt`. Example 10 shows an `orb_plugins` list for a process hosting the transformer.

Example 10: *Plug-in List for Using XSLT*

```
orb_plugins={"xslt", "xml_log_stream"};
```

Adding an Artix endpoint definition

The transformer is defined as a generic Artix endpoint. To instantiate it as a servant, Artix must know the following details:

- The location of the Artix contract that defines the transformer's endpoint.
- The interface that the endpoint implements.
- The physical details of its instantiation.

This information is configured using the configuration variables in the `artix:endpoint` namespace. These variables are described in Table 5.

Table 5: *Artix Endpoint Configuration*

Variable	Function
`artix:endpoint:endpoint_list`	Specifies a list of the endpoints and their names for the current configuration scope.
`artix:endpoint:`*`endpoint_name`*`:wsdl_location`	Specifies the location of the contract describing this endpoint.

Table 5: *Artix Endpoint Configuration*

Variable	Function

`artix:endpoint:`*`endpoint_name`*`:wsdl_port`	Specifies the port that this endpoint can be contacted on. Use the following syntax:
	`[{`*`service_qname`*`}]`*`service_name`*`[/`*`port_name`*`]`
	For example:
	`{http://www.mycorp.com}my_service/my_port`

Configuring the transformer

Configuring the transformer involves two steps that enable it to instantiate itself as a servant process and perform its work.

- Configuring the list of servants.
- Configuring the list of scripts.

Configuring the list of servants

The name of the endpoints that will be brought up as transformer servants is specified in `plugins:xslt:servant_list`. The endpoint identifier is one of the endpoints defined in `artix:endpoint:endpoint_list` entry. The transformer uses the endpoint's configuration information to instantiate the appropriate servants

Note: `artix:endpoint:endpoint_list` must be specified in the same configuration scope.

Configuring the list of scripts

The list of the XSLT scripts that each servant uses to process requests is specified in `plugins:xslt:`*`endpoint_name`*`:operation_map`. Each endpoint specified in the servant list has a corresponding operation map entry. The operation map is specified as a list using the syntax shown in Example 11.

Example 11: *Operation Map Syntax*

```
plugins:xslt:endpoint_name:operantion_map = ["wsdlOp1@filename1"
    , "wsdlOp2@filename2", ..., "wsdlOpN@filenameN"];
```

Each entry in the map specifies a logical operation that is defined in the service's contract by an `operation` element, and the XSLT script to run when a request is made on the operation. You must specify an XSLT script for every operation defined for the endpoint. If you do not, the transformer raises an exception when the unmapped operation is invoked.

Configuration example

Example 12 shows the configuration scope of an Artix application, `transformer`, that loads the Artix Transformer to process messages. The transformer is configured as an Artix endpoint named `hannibal` and the transformer uses the endpoint information to instantiate a servant to handle requests.

Example 12: *Configuration for Using the Artix Transformer*

```
transformer
{
orb_plugins = ["local_log_stream","xslt"];

artix:endpoint:endpoint_list = ["hannibal"];

artix:endpoint:hannibal:wsdl_location = "transformer.wsdl";
artix:endpoint:hannibal:wsdl_port = "{http://transformer.com/xslt}WhiteHat/WhitePort";

plugins:xslt:servant_list=["hannibal"]
plugins:xslt:hannibal:operation_map = ["op1@../script/op1.xsl", "op2@../script/op2.xsl",
   "op3@../script/op3.xsl"]
}
```

Deployment as Part of a Chain

Overview

Deploying the Artix Transformer as part of Web service chain allows you to use it as part of an integration solution without needing to necessarily modify your applications. The Artix Web Service Chain Builder facilitates the placement of the transformer into a series of Web service calls managed by Artix.

The plug-in architecture of the transformer and the chain builder allow for you to deploy this type of solution in a variety of ways depending on what is the best fit for your particular solution. The most straightforward way to deploy this type of solution is to deploy both the transformer and the chain builder into the same process. This is the deployment that will be used to outline the steps for configuring the transformer to be deployed as part of a Web service chain. In general, you will need to complete all of the same steps regardless of how you choose to deploy your solution.

Procedure

To deploy the transformer as part of a Web service chain you need to complete the following steps:

1. Modify your process's configuration scope to load the transformer and the chain builder.
2. Configure Artix endpoints for each of the applications that will be part of the chain.
3. Configure an Artix endpoint to represent the transformer.
4. Configure the transformer.
5. Configure the service chain to include the transformer at the appropriate place in the chain.

Updating the orb_plugins list

Configuring the application to load the transformer plug-in and the chain builder plug-in requires adding them to the process's `orb_plugins` list. The plug-in name for the transformer is `xslt` and the plug-in name for the chain builder is `ws_chain`. Example 13 shows an `orb_plugins` list for a process hosting the transformer and the chain builder.

Example 13: *Loading the Artix Transformer as Part of a Chain*

```
orb_plugins={"xslt", "ws_chain", "xml_log_stream"};
```

Configuring the endpoints in the chain

The Artix Web Service Chain Builder uses generic Artix endpoints to represent all of the applications in a chain, including the transformer. Table 5 on page 103 shows the configuration variables used to configure a generic Artix endpoint.

Configuring the transformer

The transformer requires the same configuration information regardless of how it is deployed. You must provide it with the name of the endpoints it will instantiate from the list of endpoints and provide each instantiation with an operation map. For more information about providing this information see "Configuring the transformer" on page 104.

Placing the transformer in the chain

The chain builder instantiates a servant for each endpoint specified in its servant list. Each servant can have a multiple operations. For each operation that will be involved in a Web service chain, you need to specify a list of endpoints and their operations that make up the chain. This list is specified using `plugins:chain:`*`endpoint_name`*`:`*`operation_name`*`:service_chain`.

To include the transformer in one of the chains, you add the appropriate operation and endpoint names for the transformer at the appropriate place in the service chain.

For more information on configuring the chain builder see "Deploying a Service Chain" on page 91.

Specifying an XSLT trace filter

You can use the `plugins:xslt:`*`endpoint_name`*`:trace_filter` variable to trace and debug the output of the XSLT engine. This configuration variable is optional. For example:

```
plugins:xslt:endpoint_name:trace_filter =
   "INPUT+TEMPLATE+ELEMENT+GENERATE+SELECT";
```

These settings are described as follows:

`INPUT`	Traces the XML input passed to the XSLT engine.
`TEMPLATE`	Traces template matches in the XSLT script.
`ELEMENT`	Traces element generation.
`GENERATE`	Traces generation of text and attributes.
`SELECT`	Traces node selections in the XSLT script.

Configuration example

Example 14 shows a configuration scope that contains configuration information for deploying the transformer as part of a Web service chain.

Example 14: *Configuring the Artix Transformer in a Web Service Chain*

```
transformer
{
  orb_plugins = ["ws_chain", "xslt"];

  event_log:filters = ["*=FATAL+ERROR+WARNING", "IT_XSLT=*"];

  bus:qname_alias:oldClient = "{http://bank.com}ATM";
  bus:initial_contract:url:oldClient = "bank.wsdl";

  bus:qname_alias:newServer = "{http://bank.com}newATM";
  bus:initial_contract:url:newServer = "bank.wsdl";

  artix:endpoint:endpoint_list = ["transformer"];

  artix:endpoint:transformer:wsdl_location = "bank.wsdl";
  artix:endpoint:transformer:wsdl_port =
   "{http://bank.com}transformer/transformer_port";

  plugins:xslt:servant_list = ["transformer"];
  plugins:xslt:transformer:operation_map =
   ["transform@transformer.xsl"];
```

Example 14: *Configuring the Artix Transformer in a Web Service Chain*

```
  plugins:chain:servant_list = ["oldClient"];
  plugins:chain:oldClient:client_operation:service_chain =
   ["transform@transformer", "withdraw@newServer"];
};
```

Note: Even though a list of servants can be specified, only one servant is currently supported in a process.

CHAPTER 9

Artix High Availability

Artix uses Berkeley DB high availability to provide support for replicated services. This chapter explains how to configure high availability in Artix.

In this chapter

This chapter discusses the following topics:

Introduction

Overview

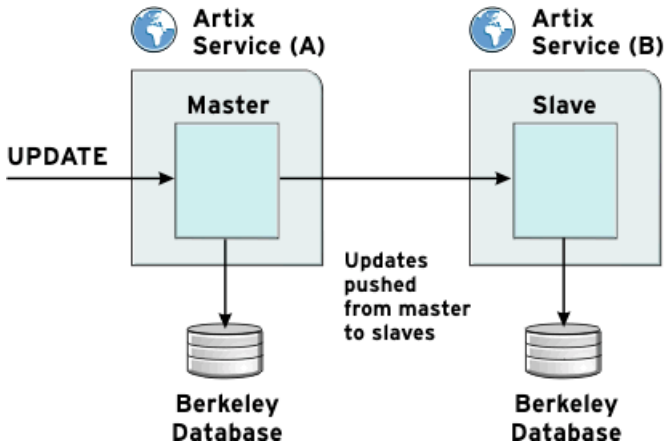
Scalable and reliable Artix applications require high availability to avoid any single point of failure in a distributed system. You can protect your system from single points of failure using *replicated services*.

A replicated service is comprised of multiple instances, or *replicas*, of the same service; and together, these act as a single logical service. Clients invoke requests on the replicated service, and Artix routes the requests to one of the member replicas. The routing to a replica is transparent to the client.

How it works

Artix high availability support is built on Berkeley DB, and uses the Berkeley DB replication features. Berkeley DB has a master-slave replica model where a single replica is designated the master, and can process both read and write operations from clients. All other replicas are slaves and can only process read operations. Slave replicas automatically forward write requests to master replicas.

Figure 20: *Artix Master Slave Replication*



Electing a master

Using Artix high availability, when members of a replicated cluster start up, they all start up as slaves. When the cluster members start up and start talking to each other, they hold an election to select a master.

Election protocol

The protocol for selecting a master is as follows:

1. For an election to succeed, a majority of votes must be cast. This means that for a group of three replicas, two replicas must cast votes. For a group of four, three replicas must cast votes; for a group of five, three must cast votes, and so on.
2. If a slave exists with a more up-to-date database than the other slaves, it wins the election.
3. If all the slaves have equivalent databases, the election result is based on the configured priority for each slave. The slave with the highest priority wins.

Note: Because voting is done by majority, it is recommended that high availability clusters have an odd number of members. The recommended minimum number of replicas is three.

After the election

When a master is selected, elections stop. However, if the slaves lose contact with the master, the remaining slaves hold a new election for master. If a slave can not get a majority of votes, nobody is promoted.

At this point, the database remains as a slave, and keeps holding elections until a master can be found. If this is the first time for the database to start up, it blocks until the first election succeeds, and it can create a database environment on disk.

If this is not the first time that it has started up, it starts as a slave (using the database files already on disk from its previous run), and continues holding elections in the background anyway.

If a high availability cluster is configured to have 5 members and a network partition separates two slaves from the rest of the cluster, neither of the separated slaves can be promoted to master because neither can get 3 votes.

Request forwarding

Slave replicas automatically forward write requests to the master replica in a cluster. Because slaves have read-only access to the underlying Berkeley DB infrastructure, only the master can make updates to the database. This feature works as follows:

1. When a replicated server starts up, it loads the `request_forwarder` plug-in.
2. When the client invokes on the server, the `request_forwarder` plug-in checks if it should forward the operation, and where to forward it to. The server programmer indicates which operations are write operations using an API.
3. If the server is running as a slave, it tries to forward any write operations to the master. If no master is available, an exception is thrown to the client, indicating that the operation cannot be processed.

Because the forwarding works as an interceptor within a plug-in, there is minimal code impact to the user. No servant code is impacted.

Note: Write request forwarding is currently (as of Artix 3.0.2) not supported by the CORBA binding, or by Java applications.

Setting up replication

You can configure replication settings in an Artix configuration file (see "Setting up a Persistent Database" on page 115, and "Configuring Persistent Services for High Availability" on page 117).

Replication is supported for C++ and Java service development, and by the Artix locator (see "Configuring Locator High Availability" on page 121).

Setting up a Persistent Database

Overview

To enable a service able to take advantage of high availability, it needs to work with a persistent database. This section explains how to set up an persistent database in Artix.

Using the Persistence API

Artix provides set of C++ and Java APIs for manipulating persistent data. For example, the C++ API uses the `PersistentMap` template class. This class stores data as name value pairs. This API is defined in `it_bus_pdk\persistent_map.h`.

This API enables you to perform tasks such as the following:

- Creating a `PersistentMap` database.
- Inserting data into a `PersistentMap`.
- Getting data from a `PersistentMap`.
- Removing data from a `PersistentMap`.

For more details, see the Developing Artix Applications with C++. For details of the Java implementation, see Developing Artix Applications in Java.

Configuring your database environment

Services that use persistent databases need some basic configuration to set up Berkeley DB (for example, the database filename). This information is stored in the following configuration variables:

`plugins:artix:db:env_name`

Specifies the filename for the Berkeley DB environment file, which is used to store `PersistentMaps`. Defaults to `db_env`.

Note: Each replica in a group must use the same name.

`plugins:artix:db:home`	Specifies the directory where Berkeley DB stores all the files for the service databases. Each service should have a dedicated folder for its data stores. This is especially important for replicated services. Defaults to `"."` (current working directory). It is recommended that you set this variable to a specific directory.

Example

The following example shows how these variables are used in an Artix configuration file:

```
persist_service {
  plugins:artix:db:env_name = “myDB.env”;
  plugins:artix:db:home = “/etc/dbs/persisting_service”;
…
};
```

Further information

For detailed information on the Berkeley DB database environment, see http://www.sleepycat.com/

Artix ships Berkeley DB 4.2.52. Alternatively, you can download and build Berkeley DB to obtain additional administration tools (for example, `db_dump`, `db_verify`, `db_recover`, `db_stat`).

Configuring Persistent Services for High Availability

Overview

For a service to participate in a high availability cluster, it must first be designed to use persistent maps ("Setting up a Persistent Database" on page 115). However, services that use persistent maps are not replicated automatically; you must configure your service to be replicated.

Configuring a service for replication

To replicate a service, you must add a replication list to your configuration, and then add scopes for each replicated instance of your service. Typically, you would create a scope for your replica cluster, and then create sub-scopes for each replica. This avoids duplicating configuration settings that are common to all replicas, and separates the cluster from any other services configured in your domain.

Specifying a replication list

To specify a cluster of replicas, use the following configuration variable:

```
plugins:artix:db:replicas
```

This takes a list of replicas specified using the following syntax:

```
ReplicaName=HostName:PortNum
```

For example, the following entry configures a cluster of three replicas spread across machines named `jimi`, `noel`, and `mitch`.

```
plugins:artix:db:replicas=["rep1=jimi:2000", "rep2=mitch:3000",
   "rep3=noel:4000"];
```

Specifying your orb_plugins list

Because IIOP is used for communication between replicas, you must include the following plug-ins in your replica's `orb_plugins` list:

- `iiop_profile`
- `giop`
- `iiop`

In addition, to enable automatic forwarding of write requests from slave to master replicas, include the `request_forwarder` plug-in. You must also specify this plug-in as a server request interceptor. The following example shows the required configuration:

```
orb_plugins = ["xmlfile_log_stream", "request_forwarder",
   "local_log_stream", "iiop_profile", "giop", "iiop"];

binding:artix:server_request_interceptor_list= "request_forwarder";
```

This configuration is loaded when the replica service starts up.

Note: To enable forwarding of write requests, programmers must have already specified in the server code which operations can write to the database. For details, see "Forwarding write requests" on page 129.

Specifying replica priorities

In each of the sub-scopes for the replicas, you must give each replica a priority, and configure the IIOP connection used by the replicas to conduct elections. This involves the following configuration variables:

`plugins:artix:db:priority`

Specifies the replica priority. The higher the priority the more likely the replica is to be elected as master. You should set this variable if you are using replication.

There is no guarantee that the replica with the highest priority is elected master. The first consideration for electing a master is who has the most current database.

Note: Setting a replica priority to `0` means that the replica is never elected master.

`plugins:artix:db:replica_name`	Specifies which replica in the `plugins:artix:db:replicas` list that this configuration refers to.
`plugins:artix:db:iiop:port`	Specifies the IIOP port the replica starts on. This entry must match the corresponding entry in the replica list.

Configuration example

The following example shows how these replication variables are used in an Artix configuration file:

```
ha_cluster{
  plugins:artix:db:env_name = “myCluster.env”;
  plugins:artix:db:replicas = [“rep1=jimi:2000”,
   “rep2=mitch:3000”, “rep3=noel:4000”];

  rep1{
    plugins:artix:db:home = “/etc/dbs/replica_1”;
    plugins:artix:db:replica_name = “rep1”;
    plugins:artix:db:priority = 80;
    plugins:artix:db:iiop:port = 2000;
  };

  rep2{
    plugins:artix:db:home = “/etc/dbs/replica_2”;
    plugins:artix:db:replica_name = “rep2”;
    plugins:artix:db:priority = 20;
    plugins:artix:db:iiop:port = 3000;
  };

  rep3{
    plugins:artix:db:home = “/etc/dbs/replica_2”;
    plugins:artix:db:replica_name = “rep3”;
    plugins:artix:db:priority = 0;
    plugins:artix:db:iiop:port = 4000;
  };
};
```

Configuration guidelines

You should observe the following:

- Ensure that each replica has its own dedicated home directory for its database files (for example, `/etc/dbs/replica_1`).
- It is not required that the value of the `replica_name` and the containing scope have the same name, but this is good practice.
- The configured `replica_name` for each replica must match the name of the WSDL port used for that service in a WSDL file. For example, the following WSDL fragment uses WSDL port names that match the replica names in "Configuration example" on page 119:

```
<wsdl:service name="SOAPService">
     <wsdl:port binding="tns:Greeter_SOAPBinding" name="rep1">
         <soap:address location="http://jimi:9551/SOAPService/rep1"/>
     </wsdl:port>
     <wsdl:port binding="tns:Greeter_SOAPBinding" name="rep2">
        <soap:address location="http://mitch:9552/SOAPService/rep2"/>
     </wsdl:port>
     <wsdl:port binding="tns:Greeter_SOAPBinding" name="rep3">
         <soap:address location="http://noel:9553/SOAPService/rep3"/>
     </wsdl:port>
 </wsdl:service>
```

- Finally, all replicas must be represented by separate WSDL ports in the same WSDL service.

Note: Currently (as of Artix 3.0.2), automatic forwarding of write requests only occurs for one service/port combination per bus.

Configuring request forward logging

Optionally, you can also specify to output logging from the `request_forwarder` plug-in.

To do this, specify the following logging subsystem in your event log filter:

```
event_log:filters =
   ["IT_BUS.SERVICE.REQUEST_FORWARDER=INFO_LOW+WARN+ERROR+FATAL"];
```

Configuring Locator High Availability

Overview

The Artix locator supports replication like any other Artix service. Replicating the locator involves specifying the same configuration that you would use for any other Artix service. However, there are some additional configuration variables that also apply to the locator.

Setting locator persistence

To enable persistence in the locator, set the following variable:

```
plugins:locator:persist_data="true";
```

This specifies if the locator uses a persistent database to store references. This defaults to `false`, which means that the locator uses an in-memory map to store references.

When replicating the locator, you must set `persist_data` to `true`. If you do not, replication is not enabled.

Setting load balancing

When `persist_data` is set to `true`, the load balancing behavior of the locator changes. By default, the locator uses a round robin method to hand out references to services that are registered with multiple endpoints. Setting `persist_data` to true causes the locator to switch from round robin to random load balancing.

You can change the default behavior of the locator to always use `random` load balancing by setting the following configuration variable:

```
plugins:locator:selection_method = “random”;
```

Setting a locator WSDL port

To enable forwarding of write requests from a slave to a master locator, you must set a locator WSDL port for each locator replica. This allows the locator to specify the WSDL port that it uses when registering its own servant.

To set a locator WSDL port, set the following configuration variable for each locator replica, for example:

```
plugins:locator:wsdl_port=Locator1;
```

Configuration example

The following example shows the configuration required for a cluster of three locator replicas.

Example 15: *Settings for Locator High Availability*

```
service {
...
bus:initial_contract:url:locator = "../../../etc/locator.wsdl";

orb_plugins = ["local_log_stream", "wsdl_publish", "request_forwarder",
   "service_locator", "iiop_profile", "giop", "iiop"];

binding:artix:server_request_interceptor_list= "request_forwarder";

plugins:locator:persist_data = "true";
plugins:artix:db:env_name = "locator";

plugins:artix:db:replicas = ["Locator1=localhost:7876",
   "Locator2=localhost:7877", "Locator3=localhost:7878"];

one {
   plugins:locator:wsdl_port = "Locator1";
   plugins:artix:db:replica_name = "Locator1";
   plugins:artix:db:priority = "100";
   plugins:artix:db:home = "one_db";
   plugins:artix:db:iiop:port = "7876";
};
```

Example 15: *Settings for Locator High Availability*

```
two{
   plugins:locator:wsdl_port = "Locator2";
   plugins:artix:db:replica_name = "Locator2";
   plugins:artix:db:priority = "75";
   plugins:artix:db:home = "two_db";
   plugins:artix:db:iiop:port = "7877";
};

three{
   plugins:locator:wsdl_port = "Locator3";
   plugins:artix:db:replica_name = "Locator3";
   plugins:artix:db:priority = "0";
   plugins:artix:db:home = "three_db";
   plugins:artix:db:iiop:port = "7878";
};
```

Using multiple locator replica groups

A highly available locator consists of a group of locators, one of which is active. The rest are replicas, which are used only when the active locator becomes unavailable. The locator group is represented by a locator WSDL file that contains multiple endpoints—one for each locator. When the `ha_conf` plug-in is loaded by Artix clients, it uses this WSDL file to resolve and connect to a locator. It tries the first endpoint, and if this does not yield a valid connection, it tries the second endpoint, and so on.

Using the `ha_conf` plugin, Artix client applications can fail over between locators in the same replica group. However, if you are using two separate replica locator groups, you will want your clients to try one group first, and then the other. In this case, you can use one of the following approaches to fail over between two separate replica locator groups:

Combine the two groups

You can combine two groups by taking the locator endpoints from the second replica group's WSDL file, and adding them to the list of endpoints in the first replica group's WSDL file. You now have a single WSDL file that contains all the locator endpoints. The `ha_conf` plug-in will try to contact locators in the order specified in this WSDL file.

Change the configured contract

First, set your Artix configuration so that `group1.wsdl` is the first replica group's WSDL file, for example:

```
bus:initial_contract:url:locator = "group1.wsdl";
```

Then if a connection cannot be made to any endpoint from this file, change the configured WSDL file to `group2.wsdl`, re-initialize the bus, and try again.

In this way, by using an extra try/catch statement in the client, you can achieve failover between two replica locator groups.

Further information

For a working example of Artix locator high availability, see the `...advanced/high_availability_locator` demo.

Configuring Client-Side High Availability

Overview

When you have implemented a highly available service using a group of replica servers, a suitably configured client can talk to the master replica. In the event that the master replica fails, one of the other replicas takes over as master, and the client fails over to one of the other replicas.

As far as the client application logic is concerned, there is no discernible interruption to the service. This section shows how to configure the client to use high availability features. It also explains the impact on the server.

Configuration steps

In most cases, configuring high availability on the client side consists of two steps:

- Create a service contract that specifies the replica group.
- Configure the client to use the high availability service.

Specifying the replica group

Before your client can contact the replicas in a replica group, you must tell the client how to contact each replica in the group. You can do this by writing the WSDL contract for your service in a particular way.

Example 16 shows the `hello_world.wsdl` contract from the `...\advanced\high_availability_persistent_servers` demo.

Example 16: *Specifying a Replica Group in a Contract*

```
?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="HelloWorld" targetNamespace="http://www.iona.com/hello_world_soap_http"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:http-conf="http://schemas.iona.com/transports/http/configuration"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.iona.com/hello_world_soap_http"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

Example 16: *Specifying a Replica Group in a Contract*

```
    <wsdl:types>
        <schema targetNamespace="http://www.iona.com/hello_world_soap_http"
            xmlns="http://www.w3.org/2001/XMLSchema">
            <element name="responseType" type="xsd:boolean"/>
            <element name="requestType" type="xsd:string"/>
            <element name="overwrite_if_needed" type="xsd:boolean"/>
       </schema>
    </wsdl:types>
   ...
   <wsdl:service name="SOAPService">
        <wsdl:port binding="tns:Greeter_SOAPBinding" name="Server1">
            <soap:address location="http://localhost:9551/SOAPService/Server1"/>
        </wsdl:port>
        <wsdl:port binding="tns:Greeter_SOAPBinding" name="Server2">
            <soap:address location="http://localhost:9552/SOAPService/Server2"/>
        </wsdl:port>
        <wsdl:port binding="tns:Greeter_SOAPBinding" name="Server3">
            <soap:address location="http://localhost:9553/SOAPService/Server3"/>
        </wsdl:port>
    </wsdl:service>

</wsdl:definitions>
```

In Example 16, the `SOAPService` service contains three ports, all of the same port type. The contract specifies fixed port numbers for the endpoints. By convention, you should ensure that the first port specified by the service corresponds to the master server.

Configuring the client to use high availability

To configure your client for high availability, perform the following steps:

1. In your client scope, add the high availability plug-in (`ha_conf`) to the `orb_plugins` list. For example:

```
client {
  orb_plugins = [...,"ha_conf"];
};
```

2. Configure the client so that the Artix bus can resolve the service contract. You can do this by specifying the following configuration in the client scope:

```
client {
  bus:qname_alias:soap_service = "{http://www.iona.com/hello_world_soap_http}SOAPService";
  bus:initial_contract:url:soap_service = "../../etc/hello_world.wsdl";
};
```

Alternatively, you can achieve the same effect by using the `-BUSservice_contract` command line parameter as follows:

```
myclient -BUSservice_contract ../../etc/hello_world.wsdl
```

For more details on configuring initial contracts, see Chapter 10.

Impact on the server

In Example 16, the contract specifies three separate ports in the same service named `SOAPService`. The implication is that each port is implemented by a different process, and if one of these processes fails, the client switches to one of the others.

Because the servers use the same contract, the server-side code must be written so the server can be instructed to instantiate a particular port. Example 17 shows some relevant code. Depending on which argument the server is started with (`1`, `2`, or `3`), it instantiates either `Server1`, `Server2` or `Server3`.

Example 17: *Server Code Chooses which Port to Instantiate*

```
//C++
String cfg_scope = "demos.high_availability_persistent_servers.server.";
String wsdl_url = "../../etc/hello_world.wsdl";
String server_number = argv[1];
String service_name = "SOAPService";
String port_name = "Server";

if (server_number == "1")
{
    cfg_scope += "one";
    port_name += "1";
}
else if (server_number == "2")
{
    cfg_scope += "two";
    port_name += "2";
}
else if (server_number == "3")
{
    cfg_scope += "three";
    port_name += "3";
}

else
{
    cerr << "Error: you must pass 1, 2 or 3 as a command line argument" <<
   endl;
    return -1;
}

    IT_Bus::Bus_var bus = IT_Bus::init(argc, argv, cfg_scope.c_str());

    IT_Bus::QName service_qname(
       "",
       service_name,
      "http://www.iona.com/hello_world_soap_http"
    );
```

Example 17: *Server Code Chooses which Port to Instantiate*

```
GreeterImpl servant(bus, service_qname, port_name, wsdl_url);

    bus->register_servant(
    servant,
    wsdl_url,
    service_qname,
    port_name
);

    cout << "Server Ready" << endl;
    IT_Bus::run();
}
catch (const IT_Bus::Exception& e)
{
   cerr << "Error occurred: " << e.message() << endl;
   return -1;
}
catch (...)
{
cerr << "Unknown exception!" << endl;
return -1;
}
return 0;
```

Forwarding write requests

When a client sends a write request to a slave replica, the slave must forward the write request to the master replica. The server programmer must use the `mark_as_write_operations()` method specify which WSDL operations can write to the database.

```
// C++
void
mark_as_write_operations(
    IT_Vector<IT_Bus::String>    operations,
    const IT_Bus::QName&         service,
    const IT_Bus::String&        port,
    const IT_Bus::String&        wsdl_url
) IT_THROW_DECL((DBException));
```

For a detailed example, see Developing Artix Applications with C++ and Developing Artix Applications in Java.

Server-side state

Client-side failover can be used with both stateful and stateless servers. If your servers are stateful, server-side high availability must be enabled for the servers. But this has no impact on the client configuration.

If your servers are stateless, no server-side configuration is necessary. However, your servers can share state using some other mechanism (for example, a shared database). In this case, client-side failover can still be used.

Further details

For working examples of high availability in Artix, see the following demos:

- `...advanced/high_availability_persistent_servers`
- `...advanced/high_availability_locator`

CHAPTER 10

Artix Bootstrapping Service

Artix enables you to bootstrap WSDL contracts and Artix references to server and client applications. This avoids hard-coding contracts in your applications. This chapter explains the benefits, and shows how to use different bootstrapping mechanisms.

In this chapter

This chapter discusses the following topics:

Introduction

Overview

Bootstrapping in Artix refers to enabling client and server applications to find WSDL service contracts and references.

This section introduces the Artix bootstrapping service and explains reasons for using it. It shows the benefits of using bootstrapping instead of hard coding WSDL into your client and server applications.

Hard coding WSDL in servers

Hard coding WSDL in servers limits the portability of your application, and can make it more difficult to develop and deploy.

For example, you have developed a Web service application that includes a client and a service implemented in a server process. When you first write the application, you have a local copy of the WSDL, and you have hard coded the WSDL location into your application.

Example C++ server

```
QName service_qname("", "SOAPService",
   http://www.iona.com/hello_world_soap_http);

HelloWorldImpl servant(bus);
  bus->register_servant(
      "../../etc/hello.wsdl",
     service_qname
  );
```

Example Java server

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

Servant servant = new SingleInstanceServant(new SoapImpl(),
   "../../etc/hello.wsdl", bus);
 bus.registerServant(servant,serviceQName,"SoapPort");
```

Hard coding WSDL in clients

Similarly, you have also hard-coded your client with the location of your local WSDL:

Example C++ client

```
HelloWorldClient proxy("../../etc/hello.wsdl");
proxy.sayHello();
```

Example Java client

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

URL wsdlLocation = null;
   try {
         wsdlLocation = new URL("../../etc/hello.wsdl");
   } catch (java.net.MalformedURLException ex) {
         wsdlLocation = new File(wsdlPath).toURL();
   }

Soap impl =
   (Soap)bus.createClient(wsdlLocation,serviceQName,portName,Soap.class);

String returnVal = impl.sayHi();
```

Note: For simplicity, this example uses the Artix bus helper to create proxies. You can also use JAXRPC.

Deploying your application

However, when your application is no longer a demo, and you want to deploy it in multiple locations, your hard-coded application may make this difficult. For example, if your client is no longer run from the same directory or machine as the server.

To solve this problem, the Artix bootstrapping core service enables you to write code that is location independent, and therefore easy to distribute and deploy.

Note: Artix bootstrapping is designed for WSDL-based services. It does not provide mechanisms for resolving local objects. For details of how to do this, see Developing Artix Applications with C++ and Developing Artix Applications in Java.

Bootstrapping Servers and Clients

Overview

Artix bootstrapping service addresses two main use case scenarios:

- Enabling server applications to find WSDL.
- Enabling client applications to find references.

Artix provides support for both of these use cases in C++ and Java.

Enabling servers to find WSDL

When you want to activate your service in a mainline or a plug-in, you should not hard code the WSDL location. Instead, use the Artix bootstrapping API to decouple the bootstrapping of WSDL from your application logic.

C++ example

The C++ bootstrapping API is as follows:

```
/**
 * Return pointer to object that corresponds to in-memory
 * representation of a specified service.
 * @param QName of the desired service.
 * @return pointer to IT_WSDL::WSDLService.
 */
virtual IT_WSDL::WSDLService*
get_service_contract(
   const QName& service_name
) IT_THROW_DECL((Exception)) = 0;
```

You can change your old hard-coded application to use this API. Your C++ server becomes:

```
QName service_qname("", "SOAPService",
   http://www.iona.com/hello_world_soap_http);

HelloWorldImpl servant(bus);
WSDLService* contract =
    bus->get_service_contract(service_qname);
bus->register_servant(
    *contract,
    servant
);
```

For simplicity, this example does not show any error handling. For details, see Developing Artix Applications with C++.

Java example

The Java bootstrapping API is as follows:

```
/**
 * Obtains the WSDL URL associated with a Service.
 * @param serviceName The QName of the Service.
 * @return String The URL of the WSDL.
 * @throws BusException If the URL cannot be found.
 */
public abstract String getServiceWSDL(QName serviceName) throws
   BusException;
```

Your Java server becomes:

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

String hwWsdl = bus.getServiceWSDL(serviceQName);

Servant servant = new SingleInstanceServant(new SoapImpl(), hwWsdl, bus);
bus.registerServant(servant,serviceQName,"SoapPort");
```

The bootstrapping of your server to a specific WSDL contract is not addressed in your application code. This is specified at runtime instead. The available bootstrapping options are explained in "Bootstrapping WSDL Contracts" on page 137.

Enabling clients to find references

When you want to initialize your client proxies in your applications, you should no longer depend on local WSDL files or static stub code information to properly instantiate a proxy. Instead, use the Artix bus API to decouple the bootstrapping of client references from your application logic.

C++ example

The C++ bootstrapping API is as follows:

```
virtual IT_Bus::Boolean
resolve_initial_reference(
    const QName & service_name,
    Reference & endpoint_reference
) IT_THROW_DECL((Exception)) = 0;
```

You can change your old hard-coded client application as follows:

```
QName service_qname("", "SOAPService",
   http://www.iona.com/hello_world_soap_http);

Reference result;
bus->resolve_initial_reference(
    service_qname, result
);

HelloWorldClient proxy(result);
proxy.sayHello();
```

Java example

The Java bootstrapping API is as follows:

```
/*** Resolve a Reference object that refers to the specified service.
* @param serviceName The name of the service.
* @return Reference An object that can be sent over the wire to refer
* to the given service. Return null if no object found.
* @throws BusException If there is an error resolving a Reference. */

public abstract Reference resolveInitialReference(QName serviceName)
   throws BusException;
```

You can change your old hard-coded Java client as follows:

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

Reference ref = bus.resolveInitialReference(serviceQName);

Soap impl = (Soap)bus.createClient(ref,Soap.class);
String returnVal = impl.sayHi();
```

Note: This Java code could use the JAXRPC programming model for creating proxies which is more portable.

The bootstrapping of your client to a specific Artix reference is not addressed in your application code. This is specified at runtime instead. The available bootstrapping options are explained in "Bootstrapping Artix References" on page 143.

Bootstrapping WSDL Contracts

Overview

When your application calls the Artix bus to find a WSDL contract for a service, the Artix bus uses several available bootstrapping resolver mechanisms to find the requested WSDL. The Artix core tries each of these in turn until it finds an appropriate contract, and returns the first result. If a resolver mechanism is configured with a bad contract URL, no others are called.

Bootstrapping WSDL is a two-step process:

1. You must first use the C++ or Java API to resolve the WSDL (see "Enabling servers to find WSDL" on page 134).
2. You must then use one of the bootstrapping resolver mechanisms to configure the WSDL at runtime. This is explained in this section.

WSDL bootstrapping mechanisms

The possible ways of bootstrapping WSDL at runtime are as follows:

1. Command line.
2. Artix configuration file.
3. Well-known directory.
4. Stub WSDL shared library.

These are listed in order of priority, which means that if you configure more than one, those higher up in the list override those lower down. See "Order of precedence for bootstrapping WSDL" on page 141.

Configuring WSDL on the command line

You can configure WSDL by passing URLs as parameters to your application at startup. WSDL URLs passed at application startup take precedence over settings in a configuration file. The syntax for passing in WSDL to any Artix application is:

```
-BUSservice_contract url
```

For example, assuming your application is using WSDL bootstrapping API, you can avoid configuration files by starting your application as follows:

```
./server -BUSservice_contract ../../etc/hello.wsdl
```

This means that the Artix bus parses the URLs that you pass into it on startup. It finds any services that are in this WSDL, and caches them for any users that want WSDL for any of those services.

Parsing WSDL on demand

If you do not want the Artix bus to parse the document until it is needed, you can specify what services are contained in the WSDL, which results in the URL being parsed only on demand. The syntax for this is:

```
-BUSservice_contract {namespace}localpart@url
```

For example, the application would be started as follows:

```
./server -BUSservice_contract
   {http://www.iona.com/demos}HelloWorldService@../../etc/hello.wsdl
```

Specifying the WSDL URL on startup enables the Artix bus to avoid parsing the WSDL until it is requested.

Configuring WSDL in a configuration file

You can also configure the location of your WSDL in an Artix configuration file, using the following configuration variable syntax.

```
bus:qname_alias:service-name = "{namespace}localpart";
bus:initial_contract:url:service-name = "url";
```

These variables are described as follows:

- `bus:qname_alias:service-name` enables you to assign an alias or shorthand version of a service QName. You can then use the short version of the service name in other configuration variables. The syntax for the service Qname is "`{namespace}localpart`".
- `bus:initial_contract:url:service-name` uses the alias defined using `bus:qname_alias` to configure the location of the WSDL contract. The WSDL location syntax is "`url`". This can be any valid URL, it does not have to be a local file.

The following example configures a service named `SimpleService`, defined in the `http://www.iona.com/bus/tests` namespace:

```
bus:qname_alias:simple_service = "{http://www.iona.com/bus/tests}SimpleService";
bus:initial_contract:url:simple_service = "../../etc/simple_service.wsdl";
```

Configuring WSDL in a well-known directory

You can also configure an Artix application to search in a well-known directory when it needs to find WSDL. This enables you to configure multiple documents without explicitly configuring every document on the command line, or using `bus:initial_contract` configuration. If you specify a well-known directory, you only need to copy the WSDL documents into this directory before the application needs them.

You can configure the directory location in a configuration file or by passing a command-line parameters to your C++ or Java application.

Configuring a WSDL directory in a configuration file

To set the directory in configuration, use the following variable:

```
bus:initial_contract_dir=["."];
```

The value "." means use the directory from where the application was started. The specified value is a list of directories, which enables you to specify multiple directories.

Configuring a WSDL directory using command-line parameters

If you do not wish to use a configuration file, you can configure the WSDL directory using command line parameters. The command line overrides any settings in a file. The syntax is as follows:

```
-BUSservice_contract_dir directory
```

For example, to configure Artix to look in the current directory, and in the "`../../etc`" directory, use the following command:

```
server -BUSservice_contract_dir . -BUSservice_contract_dir ../../etc/
```

Configuring multiple WSDL directories

You can configure multiple well-known directories for your application to search. However, it is not recommended that you put too many files in the directory.

The more files you put in the directory, the longer it may take to find the contract that you are looking for. The directory search is optimized to first do a quick file scan to see if any of the files potentially contain the target service requested. The documents are not properly parsed unless a match has been found.

If you use multiple directories, the ordering makes a difference if both directories contain the same service definitions. The resolver mechanisms search the directories in the order that they are configured in.

You can add WSDL documents to the well-known directories after the application has started. The file must only be present in the directory before the application requests it.

Bootstrapping in a stub WSDL shared library

It is also possible to encode a WSDL document inside a C++ shared library. Just like in Java, where resources are added to a `.jar` file, Artix can embed a WSDL document inside a shared library. This enables you to resolve WSDL contracts for Artix services without using a file system or any remote calls.

When a WSDL document is encoded inside a shared library, this is called a *stub WSDL shared library*. Artix provides stub WSDL shared libraries for the following Artix services:

- Locator
- Session manager
- Peer manager
- Container

This means that you can deploy these services into environments without using any other resources like WSDL documents. Artix does not provide APIs to enable you to encode your own documents into stub libraries.

Stub WSDL shared libraries are the last bootstrapping mechanisms to be called. If you configure any others, the stub WSDL shared library is not used.

All the Artix stub WSDL libraries contain WSDL endpoints with SOAP HTTP port addresses of `0`. This means that if these versions are used to activate a service, the endpoint is instantiated on a dynamic port. This is the recommended approach for internal services like the container and peer manager.

Order of precedence for bootstrapping WSDL

Because there are several available options for bootstrapping WSDL to an application, the bootstrapping service searches each resolver mechanism in turn for a suitable document. It returns the first successful result to the user.

The order of precedence for bootstrapping WSDL is as follows:

1. Contract passed on the command line.
2. Contract specified in a configuration file.
3. Well-known directory passed on the command line.
4. Well-known directory specified in a configuration file.
5. Stub WSDL shared library.

Example

You have four WSDL contracts that contain a definition for a service named `SimpleService`:

```
one/simple.wsdl
two/simple.wsdl
three/simple.wsdl
four/simple.wsdl
```

1. Configure the following in your configuration file:

```
bus:qname_alias:simple_service =
   "{http://www.iona.com/bus/tests}SimpleService";
bus:initial_contract:url:simple_service = "two/simple.wsdl";
bus:initial_contract_dir=["four"];
```

2. Start your server as follows:

```
server -BUSservice_contract_dir three -BUSservice_contract one/simple.wsdl
```

The contract in `one/simple.wsdl` is returned to the application because WSDL configured using `-BUSservice_contract` takes precedence over all other sources.

If you start your server as follows:

```
server
```

The contract in `two/simple.wsdl` is returned to the application because the order that the bootstrapping resolver mechanisms are called in means that the contract specified in a configuration file is the first successful one.

Bootstrapping standard Artix services

For details of bootstrapping WSDL for standard Artix services such as the locator or session manager, see "Bootstrapping Well-Known Artix Services" on page 149.

Bootstrapping Artix References

Overview

An Artix *reference* is an object that encapsulates the endpoint and contract information for a particular WSDL service. A serialized Artix reference is an XML document that refers to a running service instance, and contains a URL pointer to where the service's WSDL can be retrieved. You can serialize a reference to any service by deploying it into the Artix container and calling `it_container_admin -publishreference`. Alternatively, you can use APIs to publish a reference directly.

When your client application uses the Artix bus to look up a reference using the service QName, it calls the bootstrapping service. Bootstrapping references works the same way as bootstrapping WSDL, and you have several options for configuring the reference that the client uses. Like with WSDL contracts, Artix tries each bootstrapping mechanism in turn until it gets a successful result or an error. If any of these return null, the core tries the next one. If you have a badly configured reference, the bootstrapping mechanism returns an error or exception.

Bootstrapping references is a two-step process:

1. You must first use the C++ or Java API to resolve the reference (see "Enabling clients to find references" on page 135).
2. You must then use one of the resolver mechanisms to configure the reference at runtime. This is explained in this section.

For details of how to use the Artix container to publish references for a client, see Chapter 3.

Reference bootstrapping mechanisms

The possible ways of configuring references at runtime are as follows:

1. Colocated service.
2. C++ programmatic configuration.
3. Command line
4. Configuration file.
5. WDSL contract.

These are listed in order of precedence, so if you configure more than one, those higher up in the list override those lower down. The bootstrapping service searches each in turn for a suitable match and returns the first successful result.

Using a colocated service

The most convenient place to find a reference to a service that a client has requested is in the local Artix bus. When the activated service is colocated (available locally in the same process), the client can easily find a local reference to invoke. In this case, the client's `resolve_initial_reference()` function returns a reference to the colocated service.

This is the first bootstrapping mechanism that the runtime checks. You can expect resolution to always succeed for services that are activated locally.

Configuring references in C++ code

In C++, you can register an initial reference programmatically using the Artix bus. You can register a reference in one C++ plug-in that would enable another plug-in (Java or C++) to resolve that reference using the bus API.

The bootstrapping service checks the bus for local services, so it would be unusual for an application to require the programmatic configuration unless it uses multiple buses. You can not programmatically configure a reference in one bus and have it resolved in another.

In addition, you can not activate a service in one bus, and have it resolved in another. If you wish a client in one bus to use a reference from an active service in another bus you should programmatically register the reference from one bus to the next.

For example:

```
QName service_qname("", "SOAPService",
   http://www.iona.com/hello_world_soap_http);

// Activate the service on bus one
HelloWorldImpl servant(bus_one);

WSDLService* contract =
   bus_one->get_service_contract(service_qname);

bus_one->register_servant(
    *contract,
     servant
);

Service_var service = bus_one->get_service(service_qname);

// Register the service reference on bus two
bus_two->register_initial_reference(service->get_reference());
```

Configuring references on the command line

You can also pass in reference URLs as parameters to the application on startup. Reference URLs passed to the application on startup take precedence over settings in a configuration file. The syntax for passing in a reference to any Artix application is:

```
-BUSinitial_reference url
```

For example, assuming your application is using the bootstrapping API, you could avoid configuration files by starting your application as follows:

```
./server -BUSinitial_reference ../../etc/hello.xml
```

This means that the Artix bus parses the URLs passed into it on startup. It caches them for any users that request references of this type at runtime.

Parsing references on demand

If you do not want to parse the reference XML until it is needed, you can specify the service name that the reference maps to. This means that the XML is not parsed until it is first requested. The syntax for this is

```
-BUSinitial_reference {namespace}localpart@url
```

For example, the application is started as follows:

```
./server -BUSinitial_reference
   {http://www.iona.com/demos}HelloWorldService@../../etc/hello.xml
```

Configuring references in a configuration file

You can also configure a reference in a configuration file. The reference must be serialized in an XML format. You can use a configuration variable syntax to configure the URL or contents of a serialized reference.

Format of a serialized XML reference

The following shows an example contents of a serialized reference:

```
<?xml version='1.0' encoding='utf-8'?>
<m1:reference service="m2:AccountService"
                       wsdlLocation="file:./bank.wsdl"
                       xmlns:xs="http://www.w3.org/2001/XMLSchema"
                       xmlns:m1="http://www.iona.com/bus"
                       xmlns:m2="http://www.iona.com/bus/tests"
                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <port name="AccountPort" binding="m2:AccountBinding">
        <m3:address xsi:type="m3:tAddress"
            location="http://localhost:999/AccountService/AccountPort/"
            xmlns:m3="http://schemas.xmlsoap.org/wsdl/soap/">
        </m3:address>
    </port>
</m1:reference>
```

Configuring serialized reference URLs

You can also configure the location of your WSDL in an Artix configuration file, using the following configuration variable syntax.

```
bus:qname_alias:service-name = "{namespace}localpart";
bus:initial_references:url:service-name = "url";
```

These variables are described as follows:

- `bus:qname_alias:`*`service-name`* enables you to assign an alias or shorthand version of a service QName. You can then use the short version of the service name in other configuration variables. The syntax for the service Qname is "`{`*`namespace`*`}`*`localpart`*".
- `bus:initial_contract:url:`*`service-name`* uses the alias defined using `bus:qname_alias` to configure the location of the reference. The XML location syntax is "*`url`*". The URL value can be any valid URL, it does not have to be a local file, but under most circumstances the reference is local.

The following example configures a service named `SimpleService`, defined in the `http://www.iona.com/bus/tests` namespace:

```
bus:qname_alias:simple_service = "{http://www.iona.com/bus/tests}SimpleService";
bus:initial_contract:url:simple_service = "../../etc/simple_service.xml";
```

Configuring inline references

Instead of configuring a URL, you can also inline the reference XML in a configuration file. This is similar to configuring CORBA initial references in Orbix, and it effectively hard codes the addressing. This should only be used for static services where you do not expect anything to change (for example, details such as the endpoint address and transport information).

The following is an example inline reference:

```
bus:qname_alias:simple_service = "{http://www.iona.com/bus/tests}SimpleService";
bus:initial_references:inline:simple_service = "<?xml version='1.0' encoding='utf-8'?> ....";
```

The reference appears on one line in an XML document.

Configuring references using WSDL

Artix reference bootstrapping is built on its WSDL bootstrapping. When bootstrapping a reference, you can use all the options available for bootstrapping WSDL. When you locate a WSDL document that contains the `wsdl:service` you are looking for, you can convert it to a reference and return it to the client.

If Artix fails to find a suitable reference using the reference bootstrapping mechanisms, it falls back to those used for WSDL. This is useful in certain scenarios. For example, when you only want to configure well-known Artix services (such as the locator). If you configure the WSDL, both the service and the client can benefit from a single configuration source and bootstrap successfully.

Implications of resolving references using WSDL

When no references are found, the bootstrapping mechanisms for references call those used for WSDL. This means that you can rely on WSDL to configure client references.

However, the default WSDL contracts for well-known Artix services have SOAP/HTTP endpoints with a port of zero. For example:

```
<service name="LocatorService">
  <port binding="ls:LocatorServiceBinding" name="LocatorServicePort">
   <soap:address location="http://localhost:0/services/locator/LocatorService"/>
 </port>
</service>
```

If you resolve a reference with a port of zero, you get an error when you try to invoke the proxy created from the reference. The exception says that the address is invalid.

These contracts with ports of zero are intended for use by servers rather than clients, and enable servers to run on a dynamic port. Therefore, in general, your client should not rely these contracts. If the server is using this type of contract, you should publish the activated form of the contract, which contains the port assigned dynamically at startup. Your client can then bootstrap this activated version of the contract instead.

Further information

For more detailed information on Artix references, see Developing Artix Applications with C++, or Developing Artix Applications in Java.

Bootstrapping Well-Known Artix Services

Overview

Artix includes WSDL contracts for all its well-known services. This section shows the default bootstrapping provided for these services.

Pre-configured WSDL

Artix provides pre-configured aliases and WSDL locations for all of its services. By default, the Artix configuration file (`artix.cfg`) includes the following entries:

```
# Well known Services QName aliases
bus:qname_alias:container = "{http://ws.iona.com/container}ContainerService";
bus:qname_alias:locator = "{http://ws.iona.com/locator}LocatorService";
bus:qname_alias:peermanager = "{http://ws.iona.com/peer_manager}PeerManagerService";
bus:qname_alias:sessionmanager = "{http://ws.iona.com/sessionmanager}SessionManagerService";
bus:qname_alias:sessionendpointmanager =
   "{http://ws.iona.com/sessionmanager}SessionEndpointManagerService";
bus:qname_alias:uddi_inquire = "{http://www.iona.com/uddi_over_artix}UDDI_InquireService";
bus:qname_alias:uddi_publish = "{http://www.iona.com/uddi_over_artix}UDDI_PublishService";
bus:qname_alias:login_service = "{http://ws.iona.com/login_service}LoginService";

bus:initial_contract:url:container = "install_root/artix/3.0/wsdl/container.wsdl";
bus:initial_contract:url:locator = "install_root/artix/3.0/wsdl/locator.wsdl";
bus:initial_contract:url:peermanager = "install_root/artix/3.0/wsdl/peer-manager.wsdl";
bus:initial_contract:url:sessionmanager =
   "install_root/artix/3.0/wsdl/session-manager.wsdl";
bus:initial_contract:url:sessionendpointmanager =
   "install_root/artix/3.0wsdl/session-manager.wsdl";
bus:initial_contract:url:uddi_inquire = "install_root/artix/3.0/wsdl/uddi/uddi_v2.wsdl";
bus:initial_contract:url:uddi_publish = "install_root/artix/3.0/wsdl/uddi/uddi_v2.wsdl";
bus:initial_contract:url:login_service = "install_root/artix/3.0/wsdl/login_service.wsdl";
```

In your application, if you resolve the WSDL or a reference for any of these services, by default, the WSDL from these values is used. Most of these services are configured to use a port of zero. If you do not want to use the default WSDL for any of these services, you must override the default.

Further information

For more details on the configuration variables for bootstrapping Artix services, see the *Artix Configuration Guide*.

For more examples of bootstrapping in Artix applications, see the following demos:

- `..demos\basic\bootstrap`
- `..demos\advanced\container\deploy_plugin`
- `..demos\advanced\container\deploy_routes`
- `..demos\advanced\locator`
- `..demos\advanced\locator_list_endpoints`

Part III

Integrating Artix

In this part

This part contains the following chapters:

Further information

For more details on using Artix other middleware environments, see Artix for CORBA and Artix for Java.

CHAPTER 11

Embedding Artix in a BEA Tuxedo Container

Artix can be run and managed by BEA Tuxedo like a native Tuxedo application.

In this chapter

This chapter includes the following sections:

Introduction

Overview

To enable Artix to interact with native BEA Tuxedo applications, you must embed Artix in the Tuxedo container.

At a minimum, this involves adding information about Artix in your Tuxedo configuration file, and registering your Artix processes with the Tuxedo bulletin board.

In addition, you can also enable to Tuxedo bring up your Artix process as a Tuxedo server when running `tmboot`.

Note: BEA Tuxedo integration is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports Tuxedo integration.

This chapter explains these steps in detail.

Embedding an Artix Process in a Tuxedo Container

Note: A Tuxedo administrator is required to complete a Tuxedo distributed architecture. When deploying Artix in a distributed architecture with other middleware, please also see the documentation for those middleware products.

Procedure

To embed an Artix process in a Tuxedo container, complete the following steps:

1. Ensure that your environment is correctly configured for Tuxedo.
2. You can add the Tuxedo plug-in, `tuxedo`, to your Artix process's `orb_plugins` list.

   ```
   orb_plugins=[... "tuxedo"];
   ```

   However, the tuxedo plug-in is loaded transparently when the process parses the WSDL file.
3. Set `plugins:tuxedo:server` to `true` in your Artix configuration scope.
4. Ensure that the executable for your Artix process is placed in the directory specified in the `APPDIR` entry of your Tuxedo configuration.
5. Edit your Tuxedo configuration's `SERVERS` section to include an entry for your Artix process.

   For example, if the executable of your Artix process is `ringo`, add the following entry in the `SERVERS` section:

   ```
   ringo SVRGRP=BEATLES SVRID=1
   ```

   This associates `ringo` with the Tuxedo group called `BEATLES` in your configuration and assigns `ringo` a server ID of 1. You can modify the server's properties as needed.

6. Edit your Tuxedo configuration's `SERVICES` section to include an entry for your Artix process.

   While standard Tuxedo servers only require a `SERVICES` entry if you are setting optional runtime properties, Artix servers in the Tuxedo container require an entry, even if no optional runtime properties are being set. The name entered for the Artix process is the name specified in the `serviceName` attribute of the Tuxedo port defined in the Artix contract for the process.

   For example, given the port definition shown in Example 18, the `SERVICES` entry would be `personalInfoService`.

Example 18: *Sample Service Entry*

```
<service name="personalInfoService">
   <port name="tuxInfoPort" binding="tns:personalInfoBinding">
       <tuxedo:server>
           <tuxedo:service name="personalInfoService"/>
       </tuxedo:server>
  </port>
</service>
```

7. If you made the Tuxedo configuration changes in the ASCII version of the configuration, `UBBCONFIG`, reload the `TUXCONFIG` with `tmload`.

When you have configured Tuxedo, it manages your Artix process as if it were a regular Tuxedo server.

CHAPTER 12

Enterprise Performance Logging

IONA's performance logging plug-ins enable Artix to integrate effectively with third-party Enterprise Management Systems (EMS).

In this chapter

This chapter contains the following sections:

Enterprise Management Integration

Overview

IONA's performance logging plug-ins enable both Artix and Orbix to integrate effectively with *Enterprise Management Systems* (EMS), such as IBM Tivoli™, HP OpenView™, CA Unicenter™, or BMC Patrol™. The performance logging plug-ins can also be used in isolation or as part of a bespoke solution.

Enterprise Management Systems enable system administrators and production operators to monitor enterprise-critical applications from a single management console. This enables them to quickly recognize the root cause of problems that may occur, and take remedial action (for example, if a machine is running out of disk space).

Performance logging

When performance logging is configured, you can see how each Artix server is responding to load. The performance logging plug-ins log this data to file or `syslog`. Your EMS (for example, IBM Tivoli) can read the performance data from these logs, and use it to initiate appropriate actions, (for example, issue a restart to a server that has become unresponsive, or start a new replica for an overloaded cluster).

Example EMS integration

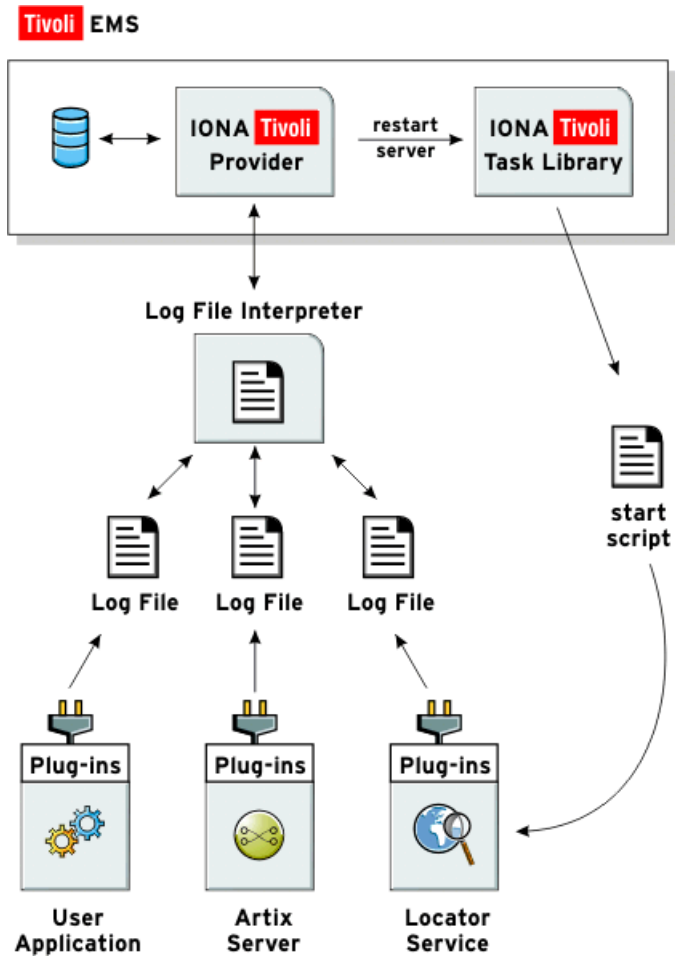
Figure 21 shows an overview of the IONA and IBM Tivoli integration at work. In this example, a restart command is issued to an unresponsive server.

In Figure 21, the performance log files indicate a problem. The IONA Tivoli Provider uses the log file interpreter to read the logs. The provider sees when a threshold is exceeded and fires an event. The event causes a task to be activated in the Tivoli Task Library. This task restarts the appropriate server.

This chapter explains how to manually configure the performance logging plug-ins. It also explains the format of the performance logging messages.

For details on how to integrate your EMS environment with Artix, see the IONA guide for your EMS. For example, the *IONA Tivoli Integration Guide* or *IONA BMC Patrol Integration Guide*.

Figure 21: *Overview of an Artix and IBM Tivoli Integration*



Configuring Performance Logging

Overview

This section explains how to manually configure performance logging. This section includes the following:

- "Performance logging plug-ins".
- "Monitoring Artix requests".
- "Logging to a file or syslog".
- "Logging to a syslog daemon".
- "Monitoring clusters".
- "Configuring a server ID".
- "Configuring a client ID".
- "Configuring with the GUI".

Note: You can also use the **Artix Designer** GUI tool to configure performance logging automatically. However, manual configuration gives you more fine-grained control.

Performance logging plug-ins

The performance logging component includes the following plug-ins:

Table 6: *Performance Logging Plug-ins*

Plug-in	Description
Response monitor	Monitors response times of requests as they pass through the Artix binding chains. Performs the same function for Artix as the response time logger does for Orbix.
Collector	Periodically collects data from the response monitor plug-in and logs the results.

Monitoring Artix requests

You can use performance logging to monitor Artix server and client requests.

To monitor both client and server requests, add the `bus_response_monitor` plug-in to the `orb_plugins` list in the global configuration scope. For example:

```
orb_plugins = ["xmlfile_log_stream", "soap", "at_http",
   "bus_response_monitor"];
```

To configure performance logging on the client side only, specify this setting in a client scope only.

Logging to a file or syslog

You can configure the collector plug-in to log data either to a file or to `syslog`. The configuration settings depends on whether your application is written in C++ or Java.

C++ configuration

The following example configuration for a C++ application results in performance data being logged to `/var/log/my_app/perf_logs/treasury_app.log` every 90 seconds:

```
plugins:it_response_time_collector:period = "90";
plugins:it_response_time_collector:filename =
"/var/log/my_app/perf_logs/treasury_app.log";
```

If you do not specify the response time period, it defaults to 60 seconds.

Java configuration

Configuring the Java collector plug-in is slightly different from the C++ collector) because the Java collector plug-in makes use of Apache Log4J. Instead of setting `plugins:it_response_time_collector:filename`, you set the `plugins:it_response_time_collector:log_properties` to use Log4J, for example:

```
plugins:it_response_time_collector:log_properties = ["log4j.rootCategory=INFO, A1",
"log4j.appender.A1=com.iona.management.logging.log4jappender.TimeBasedRollingFileAppender",
"log4j.appender.A1.File="/var/log/my_app/perf_logs/treasury_app.log",
"log4j.appender.A1.MaxFileSize=512KB",
"log4j.appender.A1.layout=org.apache.log4j.PatternLayout",
"log4j.appender.A1.layout.ConversionPattern=%d{ISO8601} %-80m %n"
];
```

Logging to a syslog daemon

You can configure the collector to log to a syslog daemon or Windows event log, as follows:

```
plugins:it_response_time_collector:system_logging_enabled = "true";
plugins:it_response_time_collector:syslog_appID = "treasury";
```

The `syslog_appid` enables you to specify your application name that is prepended to all syslog messages. If you do not specify this, it defaults to `iona`.

Monitoring clusters

You can configure your EMS to monitor a cluster of servers. You can do this by configuring multiple servers to log to the same file. If the servers are running on different hosts, the log file location must be on an NFS mounted or shared directory.

Alternatively, you can use `syslogd` as a mechanism for monitoring a cluster. You can do this by choosing one `syslogd` to act as the central logging server for the cluster. For example, say you decide to use a host named `teddy` as your central log server. You must edit the `/etc/syslog.conf` file on each host that is running a server replica, and add a line such as the following:

```
# Substitute the name of your log server
  user.info                                          @teddy
```

Some syslog daemons will not accept log messages from other hosts by default. In this case, it may be necessary to restart the `syslogd` on `teddy` with a special flag to allow remote log messages.

You should consult the `man` pages on your system to determine if this is necessary and what flags to use.

Configuring a server ID

You can configure a server ID that will be reported in your log messages. This server ID is particularly useful in the case where the server is a replica that forms part of a cluster.

In a cluster, the server ID enables management tools to recognize log messages from different replica instances. You can configure a server ID as follows:

```
plugins:it_response_time_collector:server-id = "Locator-1";
```

This setting is optional; and if omitted, the server ID defaults to the ORB name of the server. In a cluster, each replica must have this value set to a unique value to enable sensible analysis of the generated performance logs.

Configuring a client ID

You can also configure a client ID that will be reported in your log messages. Specify this using the `client-id` configuration variable, for example:

```
plugins:it_response_time_collector:client-id = "my_client_app";
```

This setting enables management tools to recognize log messages from client applications. This setting is optional; and if omitted, it is assumed that that a server is being monitored.

Configuration example

The following simple example configuration file is from the management demo supplied in your Artix installation:

```
include "../../../../../etc/domains/artix.cfg";

demos {

  management

  {

    orb_plugins = ["xmlfile_log_stream", "soap", "at_http",
                   "bus_response_monitor"];
```

```
    plugins:it_response_time_collector:period = "5";

   client {

    plugins:it_response_time_collector:client-id=
      "management-demo-client";

    plugins:it_response_time_collector:filename=
      "management_demo_client.log";
   };

   server {

     plugins:it_response_time_collector:server-id=
       "management-demo-server";

     plugins:it_response_time_collector:filename=
       "management_demo_server.log";
   };
  };
};
```

In this example, the `bus_response_monitor` plug-in and `plugins:it_response_time_collector:period` are set in the global scope. This specifies these settings for both the client and server applications.

Configuring with the GUI

The **Artix Designer** GUI tool automatically generates performance logging configuration for the Artix services. The generated `server-id` defaults to the following format:

`DomainName_ServiceName_Hostname` (for example, `artix_locator_myhost`)

For details on how to automatically generate performance logging, see the *IONA Tivoli Integration Guide* or *IONA BMC Patrol Integration Guide*.

Performance Logging Message Formats

Overview

This section describes the performance logging message formats used by IONA products. It includes the following:

- "Artix log message format".
- "Orbix log message format".
- "Simple life cycle message formats".

Artix log message format

Performance data is logged in a well-defined format. For Artix applications, this format is as follows:

```
YYYY-MM-DD HH:MM:SS server=ServerID [namespace=nnn service=sss
   port=ppp operation=name] count=n avg=n max=n min=n int=n oph=n
```

Table 7: *Artix log message arguments*

Argument	Description
`server`	The server ID of the process that is logging the message.
`namespace`	The Artix namespace.
`service`	The Artix service.
`port`	The Artix port.
`operation`	The name of the operation for CORBA invocations or the URI for requests on servlets.
`count`	The number of operations of invoked (IIOP). or The number of times this operation or URI was logged during the last interval (HTTP).
`avg`	The average response time (milliseconds) for this operation or URI during the last interval.

Table 7: *Artix log message arguments*

Argument	Description
`max`	The longest response time (milliseconds) for this operation or URI during the last interval.
`min`	The shortest response time (milliseconds) for this operation or URI during the last interval.
`int`	The number of milliseconds taken to gather the statistics in this log file.
`oph`	Operations per hour.

The combination of namespace, service and port above denote a unique Artix endpoint.

Orbix log message format

The format for Orbix log messages is as follows:

```
YYYY-MM-DD HH:MM:SS server=ServerID [operation=Name] count=n
   avg=n max=n min=n int=n oph=n
```

Table 8: *Orbix log message arguments*

Argument	Description
`server`	The server ID of the process that is logging the message.
`operation`	The name of the operation for CORBA invocations or the URI for requests on servlets.
`count`	The number of operations of invoked (IIOP). or The number of times this operation or URI was logged during the last interval (HTTP).
`avg`	The average response time (milliseconds) for this operation or URI during the last interval.
`max`	The longest response time (milliseconds) for this operation or URI during the last interval.

Table 8: *Orbix log message arguments*

Argument	Description
`min`	The shortest response time (milliseconds) for this operation or URI during the last interval.
`int`	The number of milliseconds taken to gather the statistics in this log file.
`oph`	Operations per hour.

Simple life cycle message formats

The server will also log simple life cycle messages. All servers share the following common format.

```
YYYY-MM-DD HH:MM:SS server=ServerID status=CurrentStatus
```

Table 9: *Simple life cycle message formats arguments*

Argument	Description
`server`	The server ID of the process that is logging the message.
`status`	A text string describing the last known status of the server (for example, `starting_up`, `running`, `shutting_down`).

CHAPTER 13

Artix CA-WSDM Integration

Artix provides support for integration with Computer Associates Web Services Distributed Management (CA-WSDM). This chapter provides an introduction, and shows how to configure CA-WSDM integration in Artix applications.

In this chapter

This chapter includes the following sections:

Artix CA WSDM Observer

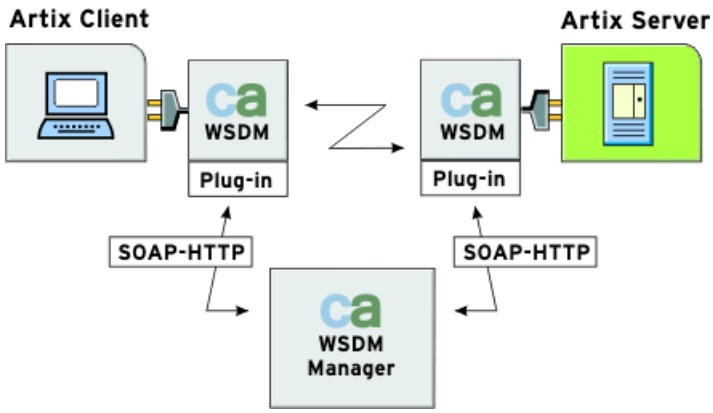
Overview

An Artix CA WSDM observer is a plug-in interceptor that integrates Artix with Computer Associates Web Services Distributed Management (WSDM) software. This section gives an architectural overview and lists the observed data.

Artix CA WSDM Observer

An Artix CA WSDM observer interceptor can sit on the client side or server side as shown in Figure 22.

Figure 22: *CA WSDM Observer Architecture*



A CA WSDM observer operates as follows:

- Collects information about messages sent to observed services over any supported transports at both the server and client request interceptor level. It asynchronously reports this information to a CA WSDM service using SOAP over HTTP.

- Automatically registers all services it observes with CA WSDM by sending a service QName and a list of QNames of operations supported by a corresponding `portType` interface. This enables a CA WSDM operator to create *service groups*.
- Periodically polls a CA WSDM service for configuration updates. For example, CA WSDM transaction monitors can be enabled, which enable an operator to look at the raw input and output messages. The operator can check if it took an operation longer to complete its request, or if its request or response size was larger than expected.

Observed data

An Artix CA WSDM observer reports the following data to a CA WSDM service about any service operation:

- Operation name and namespace.
- Request and response size.
- Operation duration.
- Operation timestamp.
- Operation transaction identifier.
- Service port endpoint address.
- Client address (where the request came from).
- Request and response messages (if duration, request/response size monitors are enabled).
- User fault exception details.

Note: Some data may not be reported if it is not available at a request interceptor level for a given transport. For example, request and response size, or raw messages for CORBA services.

Configuring a CA WSDM Observer

Overview

You can enable an Artix CA WSDM Observer by adding a plug-in to your in `orb_plug-ins` list in your server or client scopes. All other CA WSDM configuration variables are optional. This section explains how to set all available options.

Setting the orb_plugins list

The CA WSDM Observer plug-in name is `ca_wsdm_observer`. To enable a CA WSDM observer, add this plug-in to your `orb_plugins` list in your server or client scope. For example:

```
# Artix configuration file

my_client_scope {
    orb_plugins = [ ..., "ca_wsdm_observer"];
    ...
};
```

Both client and server use the same endpoint, so specifying both gives more coarse-grained data. Typically, you would use either client or server only.

Specifying a minimum queue size

The minimum queue size specifies how many service request records must be available in a queue before a report is sent to a WSDM service. For example:

```
plugins:ca_wsdm_observer:min_queue_size = "6";
```

The default is `5`. You should set this variable if your load is expected to be large. If this variable is too low, the observer may send reports too frequently, and if it is too high, the memory footprint may increase significantly.

Specifying a report wait time

The report wait time specifies how often reports should be sent in seconds. For example:

```
plugins:ca_wsdm_observer:report_wait_time = 10;
```

This variable is an alternative to `min_queue_size`, which instead specifies the frequency of reports on a time basis. This variable should be used with `max_queue_size`.

Specifying a maximum queue size

The maximum queue size specifies the maximum number of service request records that the observer queue can hold. For example:

```
plugins:ca_wsdm_observer:max_queue_size = "600";
```

The default is `500`. New records are dropped when the queue size reaches this value. If `report_wait_time` is not set, this variable is ignored. In this case, reports are sent as soon as the queue size is equal to `max_queue_size`.

Automatically registering services

You can also specify whether the observer automatically registers observed services with a WSDM service. The default is:

```
plugins:ca_wsdm_observer:auto_register = "true";
```

If you have a large number of observed services, the runtime performance might be decreased because of equally large register service requests sent to a WSDM service.

You can set this variable to `false` and manually import service details from WSDL definitions into a WSDM console. However, this only works for SOAP-HTTP non-transient services. This is because WSDM can not import non-SOAP services described in WSDL, while Artix does not publish WSDL for transient services.

Specifying a handler type

A handler type specifies a value that identifies an Artix observer to a WSDM service. It should be above `200`. The default is:

```
plugins:ca_wsdm_observer:handler_type = "217";
```

In addition, if you change the default, you must also update the following file with the new handler type:

```
WSDM-Install-Dir/server/default/conf/WsdmSOMMA_Basic.properties
```

Entries in this file take a format of `observertype.X=ArtixObserver`, where `X` is the handler type value. The default entry is:

```
observertype.217=ArtixObserver
```

Specifying a configuration updates

To specify how often, in seconds, the observer should poll a WSDM service for configuration updates, use the following variable:

```
plugins:ca_wsdm_observer:config_poll_time
```

The default is `180` seconds (3 minutes). Configuration updates tell the observer whether transaction monitors have been enabled. If so, the observer copies input/output raw messages, and reports them to a WSDM service if duration or request/response size thresholds have been exceeded.

Further information

For a detailed example, see the CA WSDM demo in the following directory:

```
InstallDir\artix\3.0\demos\integration\ca_wsdm
```

For more information on CA WSDM, see the Computer Associates website (http://www.ca.com).

CHAPTER 14

Locating Services with UDDI

Artix provides support for Universal Description, Discovery and Integration (UDDI). This chapter explains the basics, and shows how to configure UDDI proxy support in Artix applications. It also shows how to configure jUDDI repository settings.

In this chapter

This chapter includes the following sections:

Introduction to UDDI

Overview

A Universal Description, Discovery and Integration (UDDI) registry is a form of database that enables you to store and retrieve Web services endpoints. It is particularly useful as a means of making Web services available on the Internet.

Instead of making your WSDL contract available to clients in the form of a file, you can publish the WSDL contract to a UDDI registry. Clients can then query the UDDI registry and retrieve the WSDL contract at runtime.

Publishing WSDL to UDDI

You can publish your WSDL contract either to a local UDDI registry or to a public UDDI registry, such as http://uddi.ibm.com or http://uddi.microsoft.com.

To publish your WSDL contract, navigate to one of the public UDDI Web sites and follow the instructions there.

A list of public UDDI registries is available from WSINDEX (http://www.wsindex.org/UDDI/Registries/index.html)

Artix UDDI URL format

Artix uses UDDI query strings that take the form of a URL. The syntax for a UDDI URL is as follows:

uddi:*UDDIRegistryEndpointURL*?*QueryString*

The UDDI URL is built from the following components:

- *UDDIRegistryEndpointURL*—the endpoint address of a UDDI registry. This could either be a local UDDI registry (for example, `http://localhost:9000/services/uddi/inquiry`) or a public UDDI registry on the Internet (for example, `http://uddi.ibm.com/ubr/inquiryapi` for IBM's UDDI registry).

- *QueryString*—a combination of attributes used to query the UDDI database for the Web service endpoint data. Currently, Artix only supports the `tmodelname` attribute. An example of a query string is:

  ```
  tmodelname=helloworld
  ```

  Within a query component, the characters `;`, `/`, `?`, `:`, `@`, `&`, `=`, `+`, `,`, and `$` are reserved.

Examples of valid UDDI URLs

```
uddi:http://localhost:9000/services/uddi/inquiry?tmodelname=helloworld
uddi:http://uddi.ibm.com/ubr/inquiryapi?tmodelname=helloworld
```

Initializing a client proxy with UDDI

To initialize a client proxy with UDDI, simply pass a valid UDDI URL string to the proxy constructor.

For example, if you have a local UDDI registry, `http://localhost:9000/services/uddi/inquiry`, where you have registered the WSDL contract from the `HelloWorld` demonstration, you can initialize the `GreeterClient` proxy as follows:

C++

```
// C++
...
IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);

// Instantiate an instance of the proxy
GreeterClient hw("uddi:http://localhost:9000/services/uddi/inquiry?tmodelname=helloworld");

String string_out;

// Invoke sayHi operation
hw.sayHi(string_out);
```

Java

```
//Java
String wsdlPath = "uddi:http://localhost:9000/services/uddi/inquiry?tmodelname=helloworld";
...........
Bus bus = Bus.init((String[])orbArgs.toArray(new String[orbArgs.size()]));
QName name = new QName("http://www.iona.com/hello_world_soap_http","SOAPService");
QName portName = new QName("","SoapPort");
URL wsdlLocation = null;
try {
    wsdlLocation = new URL(wsdlPath);
} catch (java.net.MalformedURLException ex) {
    wsdlLocation = new File(wsdlPath).toURL();
}

ServiceFactory factory = ServiceFactory.newInstance();
Service service = factory.createService(wsdlLocation,name);
Soap impl = (Soap)service.getPort(portName,Soap.class);
```

Configuring UDDI Proxy

Overview

Artix UDDI proxy service can be used by applications to query endpoint information from a UDDI repository. This section explains how to configure UDDI proxy support for both C++ and Java client applications.

C++ configuration

To configure an Artix C++ application for UDDI proxy support, add `uddi_proxy` to the application's `orb_plugins` list. For example:

```
# Artix configuration file

my_application_scope {
    orb_plugins = [ ..., "uddi_proxy"];
    ...
};
```

Java configuration

To configure an Artix Java application for UDDI proxy support, perform the following steps:

1. Add `java` to the application's `orb_plugins` list.
2. Add `java_uddi_proxy` to the application's `java_plugins` list. For example:

```
# Artix Configuration File

my_application_scope {
    orb_plugins = [..., "java", ...];

     java_plugins=["java_uddi_proxy"];
    ...
};
```

Configuring a jUDDI Repository

Overview

The Artix demos use an open source UDDI repository implementation named jUDDI. These demos use the HSQLDB database to store UDDI information. For convenience, this is configured to run in file (embedded) mode by default.

Setting jUDDI properties

You can configure jUDDI properties, such as your database settings, in your `juddi.properties` file. This file is located in the following directory:

```
InstallDir\artix\3.0\demos\integration\juddi\artix_server\etc
```

For example, the HSQLDB database settings in the default `juddi.properties` file are as follows:

```
# hsqldb
juddi.useConnectionPool=true
juddi.jdbcDriver=org.hsqldb.jdbcDriver
juddi.jdbcURL=jdbc:hsqldb:etc/juddi_db
juddi.jdbcUser=sa
juddi.jdbcPassword=
juddi.jdbcMaxActive=10
juddi.jdbcMaxIdle=10
```

If you want change your database to MySQL, uncomment all the `mysql` settings, and use the following instead:

```
# mysql
juddi.useConnectionPool=true
juddi.jdbcDriver=com.mysql.jdbc.Driver
juddi.jdbcURL=jdbc:mysql://10.129.9.101:3306/juddi
juddi.jdbcUser=root
juddi.jdbcPassword=
juddi.jdbcMaxActive=10
juddi.jdbcMaxIdle=10
```

Further information

For more details, see: http://ws.apache.org/juddi/.

Glossary

A

Artix Designer

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

B

Binding

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

Bus

See Service Bus.

Bridge

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

C

Collection

A group of related WSDL contracts that can be deployed as one or more physical entities such as Java, C++, or CORBA based applications. It can also be deployed as a switch process.

Connection

An established communication link between any two Artix endpoints.

Contract

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

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

Contract Editor

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

D

Deployment Mode

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

E

Embedded Mode

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

Endpoint

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

H

Host

The network node on which a particular service resides.

M

Marshalling Format

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

P

Payload Format

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

Protocol

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

R

Routing

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

Router

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

S

Servant

An Artix servant is an object (Java or C++) that implements the service and port operations specified in a WSDL file

Server

An Artix server is a process in which one or more Artix servants may be created registered to process incoming operation requests through an Artix bus object.

Service

An Artix service includes a collection of ports, each of which implements a set of operations. Each port can be associated with a particular transport through a binding. A service has no compile-time dependencies. It can be dynamically configured by deploying one or more contracts on it.

Service Bus

Handles the interaction between clients and services. Enables services to activate. Enables clients to make invocations on services in a distributed environment.

The middleware used by the client or service is independent of the bus. The bus is a pluggable middleware-neutral service invocation framework. The middleware used is defined by WSDL.

Standalone Mode

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

Switch

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

System

A collection of services and transports.

T

Transport

An on-the-wire format for messages.

Transport Plug-in

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

U

UDDI

Universal Description, Discovery, and Integration.

Index

Q

R

S

T

U

V

W

X