Artix™

Locator Guide

Version 4.0, March 2006

Making Software Work Together™

Updated: 23-Mar-2006

Contents

Preface

What is Covered in this Book

This book describes how to use the Artix locator service.

Who Should Read this Book

This book is intended for administrators and developers who want to configure and deploy an Artix locator service.

The information in this book is at an intermediate to advanced level, and presumes the reader has a working knowledge of WSDL contracts, Java or C++, Artix configuration concepts, and the deployment of Artix plug-ins into an Artix container.

How to Use this Book

This book is organized into the following chapters:

- Chapter 1, Locator Introduction, which provides an overview of the Artix locator and its uses.
- Chapter 2, Configuring and Deploying the Locator Service, which describes how to edit your Artix configuration files to deploy one or more Artix locator services. This chapter includes information on using an Artix 4 locator from Artix 3 clients.
- Chapter 3, Using the Locator from an Artix Client, which describes how to write service consumer code in both C++ and Java that take advantage of a deployed Artix locator, and that queries an Artix locator. This chapter includes information on migrating from Artix 3 to Artix 4 locator clients.

The Artix Library

The Artix documentation library is organized in the following sections:

- Getting Started
- Designing and Developing Artix Solutions
- Configuring and Deploying Artix Solutions
- Using Artix Services
- Integrating Artix Solutions
- Integrating with Enterprise Management Systems
- Reference Documentation

Getting Started

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

- Release Notes contains release-specific information about Artix.
- Installation Guide describes the prerequisites for installing Artix and the procedures for installing Artix on supported systems.
- Getting Started with Artix describes basic Artix and WSDL concepts.
- Using Artix Designer describes how to use Artix Designer to build Artix solutions.
- Artix Technical Use Cases provides a number of step-by-step examples of building common Artix solutions.

Designing and Developing Artix Solutions

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

- Building Service-Oriented Architectures with Artix provides an overview of service-oriented architectures and describes how they can be implemented using Artix.
- Understanding Artix Contracts describes the components of an Artix contract. Special attention is paid to the WSDL extensions used to define Artix-specific payload formats and transports.
- Developing Artix Applications in C++ discusses the technical aspects of programming applications using the C++ API.

- Developing Advanced Artix Plug-ins in C++ discusses the technical aspects of implementing advanced plug-ins (for example, interceptors) using the C++ API.
- Developing Artix Applications in Java discusses the technical aspects of programming applications using the Java API.

Configuring and Deploying Artix Solutions

This section includes:

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

Using Artix Services

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

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

Integrating Artix Solutions

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

- Artix for CORBA provides information on using Artix in a CORBA environment.
- Artix for J2EE provides information on using Artix to integrate with J2EE applications.

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

Integrating with Enterprise Management Systems

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

- IBM Tivoli Integration Guide explains how to integrate Artix with IBM Tivoli.
- BMC Patrol Integration Guide explains how to integrate Artix with BMC Patrol.
- CA WSDM Integration Guide explains how to integrate Artix with CA WSDM.

Reference Documentation

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

- Artix Command Line Reference
- Artix Configuration Reference
- Artix WSDL Extension Reference
- Artix Java API Reference
- Artix C++ API Reference
- Artix .NET API Reference
- Artix Glossary

Getting the Latest Version

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

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

Searching the Artix Library

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

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

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

http://www.iona.com/support/docs/artix/4.0/index.xml

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

Artix Online Help

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

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

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

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

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

Artix Glossary

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

Additional Resources

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

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

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

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

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

`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**).

Locator Introduction

The Artix locator service provides a way for clients to connect to services that is independent of the service implementation. This chapter provides an overview of the Artix locator service, including its expected use cases, its operation, and its defining WSDL contract. This chapter also discusses migrating from earlier versions of the Artix locator service.

In this chapter

This chapter discusses the following topics:

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

What is the Locator Service?

Overview

The Artix locator is a Web service that provides Web service clients with a mechanism to discover service endpoints at runtime. The locator isolates client applications from knowledge of an endpoint's physical location.

Use cases

The Artix locator service supports the following use cases:

Service endpoint repository

You can use the Artix locator to isolate the consumers of Artix services from having to know the exact network location of each service. Client applications can query the locator for the current location of a service. This allows you to redeploy popular services onto different hardware or different transports without needing to recompile or reconfigure client applications in any way.

Service endpoint querying

You can query the locator for its list of currently managed services, and can filter the results list by service name, port name, portType, binding, or port extensor name. In addition, you can assign services to named groups either in the service WSDL contract or in the Artix configuration file, and can filter the queried service list by group name.

Service load balancing

If you register multiple instances of a service with an Artix locator using the same service name, the locator automatically employs a round-robin algorithm to select the service instance whose reference is returned to requesting clients. This provides you with a lightweight mechanism to distribute the load on popular services without the overhead of setting up a highly available system.

Service fault tolerance

The Artix locator has fault tolerance features on both client and server sides. The client-side plug-in monitors the connection state to a service, and retries the connection if is dropped. The service-side locator plug-in automatically deregisters the service from the locator in the event of a failure on the service side.

The Artix locator can also be configured in a high availability configuration with multiple slave locators coordinating with one master locator.

How the Locator Works

Overview

The Artix locator service is implemented by Artix plug-ins on the service side and by a plug-in plus code on the client side. A client application sends a service's QName to the locator, and the locator returns a reference for that service. The client then constructs a proxy to the target service and uses that proxy thereafter to communicate with the target service.

How the locator works

Services are made locator-aware by means of configuration statements in the Artix configuration files associated with those services. A locator-aware service automatically registers itself with the locator during service startup. The locator and its registered servers periodically confirm that their communication pathways are operational by pinging each other. This monitoring is performed by the peer manager plug-in, which is automatically loaded by the Artix runtime when the locator functionality is enabled.

Client applications contact the locator and provide the QName for the desired Web service endpoint. The locator returns a reference, which contains the connection details needed to invoke the target Web service. The client then instantiates a proxy to the target service based on the results from the locator, and uses that proxy thereafter to communicate with the service.

The locator and references

Starting with Artix 4.0, the Artix locator returns references using the WS-Addressing standard for Web service references. Previous Artix releases used the proprietary Artix Reference format.

WS-Addressing references are represented by an instance of a class[1] that includes methods to obtain the location of the WSDL file, the service name, and the collection of ports associated with the service. The client-side plug-in has constructors that use these pieces of information during initialization. Consequently, a reference contains sufficient information to allow the client-side plug-in to create a proxy to a Web service endpoint.

1. The WS-A Reference class is `IT_Bus::Reference` for C++, and `com.iona.schemas.wsaddressing.EndpointReferenceType` for Java.

In general, developers of Artix applications do not need to be concerned with the format of references returned by the locator. If you are migrating previous Artix applications to Artix 4.0 or later, see "Migrating from Previous Versions" on page 12.

Registering endpoints

An Artix service registers its endpoints with the locator in order to make them accessible to Artix clients. When a service registers an endpoint in the locator, it creates an entry in the locator that associates a service QName with a reference for that endpoint.

Looking up references

An Artix client looks up a reference in the locator in order to find an endpoint associated with a particular service. After retrieving the reference from the locator, the client can then establish a remote connection to the target service by instantiating a client proxy object. This procedure is independent of the type of binding or transport protocol.

Automatic load balancing

If multiple endpoints are registered against the same service QName in the locator, the locator employs a round-robin algorithm to pick one of the endpoints. The locator thereby effectively *load balances* a service over all of its associated endpoints.

For example, an AddNewCustomer service might be listed in an Artix locator with two endpoints registered against it:

- Service: AddNewCustomer
  WSDL location: http://mainhost:2900
- Service: AddNewCustomer
  WSDL location: http://backuphost:2900

When an Artix client looks up a reference for AddNewCustomer, it obtains a reference to whichever endpoint is next in the sequence.

Locator-related plug-ins

Most of the communication details between the locator, registered services, and clients are handled by Artix plug-ins. The locator-related plug-ins on the service side are:

Locator service plug-in (`service_locator`)	This is the main locator service plug-in. It accepts and tracks service registrations, and hands out service references to requesting clients.

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 communicates with the `locator_client` plug-in.

The locator-related plug-in on the client side is:

Locator client plug-in (`locator_client`)	This plug-in queries the locator and returns a reference to the target service. Thereafter, it maintains communication with the `locator_endpoint` plug-in, and tries to reconnect if it detects a communication drop.

When you load an instance of the `service_locator` plug-in into an Artix container, the container automatically loads the `peer_manager` plug-in.

The `service_locator` and `locator_endpoint` plug-ins are optionally used alongside the `wsdl_publish` plug-in.

How do the plug-ins interact?

In the examples in this book and in locator demonstration code, the **locator service** plug-in is deployed in an Artix container. Although it can be deployed in any Artix process, the recommended approach is to use the container. The Artix container and plug-in architecture is described in "Deploying Services in an Artix Container" in *Configuring and Deploying Artix Solutions.*

The locator service plug-in automatically loads the `peer_manager` and `wsdl_publish` services into the same Artix container. The container is started with command-line directives to publish its own URL on launch. The container's URL can be stored in a file on a shared directory, or published in some other way so that other processes can locate the container.

The container process selects a TCP port on which to place the locator service[2] (unless you specify an exact port in configuration). Client processes can use the published URL of the container to ask the container to send the ~~locator service's URL, its WSDL contra~~ct, or a reference to the locator.

2. This locator service is usually run on the same port as the container itself. Thus, for example, if you query the container at localhost:1300, chances are good the locator service will be found at localhost:1300 as well.

An Artix service process can be deployed in a standalone server or in another Artix container. The examples in this book and in the locator demonstration code show the service deployed in a standalone server.

The service process is configured to load the **locator_endpoint** plug-in. The service's server executable is started with a command-line directive that identifies the URL, WSDL, or reference of the locator service (as previously obtained from the container housing the locator). Thus, when the service process starts up, its associated `locator_endpoint` plug-in automatically contacts the locator and registers the service.

Thereafter, the `peer_manager` plug-in associated with the `locator_service` plug-in pings the service regularly to make sure the communication path is still active. If the service is taken offline without deregistering itself with the locator, the `peer_manager` service detects the absence of the service and deregisters it from the locator.

The **locator_client** plug-in is loaded into a client application by means of configuration. This plug-in handles the details of getting a reference to the target service, and then connecting to the `locator_endpoint` plug-in associated with the target service. Once connected, the `locator_client` plug-in monitors the connection between client and service processes, and attempts to reconnect to the service if it detects a dropped connection.

When used in conjunction with the locator's load balancing or high availability features, the `locator_client` plug-in can reconnect to an alternate instance of the same service registered with the locator.

Locator service groups

Starting with Artix 4.0, you can assign services to named groups so that a group of related services can be identified by group name when you query the locator. Group assignments can be made in the service's WSDL contract or in an Artix configuration file.

The use of locator service groups is described in "Service groups" on page 47.

Setting up a locator service

Configuring and running an Artix locator service does not require writing any code. You set up a locator service by means of configuration settings in your Artix configuration file. You start the locator by starting an instance of the Artix container executable that is directed to a locator-specific configuration scope (that is, to a particular ORBname) in that configuration file.

The configuration and setup of the Artix locator service is described in Chapter 2 on page 15.

Using the locator from clients

You can configure client applications to make use of the Artix locator with two steps:

- Add the client-side locator plug-in to the configuration of client applications, by editing your Artix configuration file.
- Write code that queries the locator, asking it to return a reference to the desired target service. You can also write client code that asks the locator for a filtered or unfiltered list of its registered services.

The configuration and coding of locator-enabled client applications is described in Chapter 3 on page 33.

Locator WSDL Contract

Overview

The Artix locator service is described in the `locator.wsdl` contract, which defines the public interface through which the service can be accessed either locally or remotely. The locator WSDL contract is installed by default to the following location in your Artix installation:

```
ArtixInstallDir/artix/version/wsdl/locator.wsdl
```

LocatorService portType

The locator WSDL contract defines a single portType, LocatorService. This portType includes public operations for use by Artix developers, as well as internal operations used to communicate with services as they register and deregister with the locator service.

Binding and protocol

The locator is accessed through the SOAP binding over the HTTP protocol.

Public operations

The public operations defined for the LocatorService portType are the following:

- **lookupEndpoint** A request-response operation used by a client process to look up an endpoint from the locator based on the target service's QName.
- **listEndpoints** A request-response operation used by a client process to list all endpoints registered with the locator.
- **queryEndpoints** A request-response operation used by a client process to list all endpoints registered with the locator based on selection filters.

Internal operations

The following operations defined in `locator.wsdl` are used internally by the locator-related plug-ins as they interact with the locator service:

- **registerPeerManager** Register a peer endpoint manager with the locator service. Once registered, the locator associates a peer ID with the peer endpoint manager.

- **deregisterPeerManager** Deregister a peer endpoint manager with the locator service. Deregistering a peer manager also deregisters all endpoints that were registered by it.
- **registerEndpoint** Register an endpoint to become available in the locator. Once registered, an endpoint is returned in the response to the `listEndpoints` and `queryEndpoints` operations.
- **deregisterEndpoint** Deregister an endpoint from the locator. Once deregistered, an endpoint is no longer returned in the response to the `listEndpoints` and `queryEndpoints` operations.

Locator Demonstration Code

Overview

Artix includes demonstration code samples that illustrate various Artix features. Five of these demos illustrate different aspects of the locator. The locator-related demos are installed in subdirectories of:

```
    ArtixInstallDir/artix/version/demos/advanced/
```

Read the Readme.txt file in each demo's directory for instructions on building and running that demo.

locator demo

The primary locator demo illustrates how the locator can isolate clients from knowledge about changes in a service's physical location. The examples in this manual are simplified versions of the locator demo. This demo is implemented in both C++ and Java.

locator_query demo

The locator_query demo illustrates how to use the `listEndpoints` operation to obtain a list of the services registered with a locator. The demo goes on to illustrate how you can filter the returned list of services with various query selection elements, using the `query_endpoints` operation. This demo is implemented in both C++ and Java.

located_router demo

The located_router demo illustrates how endpoints that are wrapped by an Artix router can still use the locator service for dynamic discovery of endpoint information. In this demo, the endpoints that the router creates are automatically registered with the locator when the router starts up. This demo is implemented in C++ only.

locator_load_balancing demo

The locator_load_balancing example demonstrates how the Locator can be used to provide load balancing across several server processes hosting the same Web service, without the overhead of setting up a highly available infrastructure. This demo is implemented in both C++ and Java.

high_availability_locator demo

The high_availability_locator demo illustrates how to run the Artix Locator in a replicated and highly-available mode. This demo is implemented in C++ only.

Migrating from Previous Versions

Overview

The Artix 4.0 locator service supports queries from unmodified Artix 3.x client code. This allows you to migrate at your own pace from an Artix 3-based installation to an Artix 4 installation. You can replace Artix 3 locators and services with Artix 4 locators and services without having to rewrite or change your client applications.

Backward compatibility

Starting with Artix 4.0, the Artix locator returns references in WS-Addressing format. Prior versions of the Artix locator returned references in the proprietary Artix Reference format.

To maintain backward compatibility, the Artix 4.0 locator service contains both Artix 3.0-compatible and Artix 4.0 locator services in one service. As illustrated in Figure 1, Artix 3.0 client applications can query an Artix 4.0 locator and get the expected results.

Figure 1: *Artix 4.0 locator backward compatibility*

Figure TBD

Locator service QNames

The QName for the Artix 4 locator service is:

```
{http://ws.iona.com/2005/11/locator}LocatorService
```

The QName for the Artix 3-compatible locator service that runs alongside the Artix 4 locator service is the same as it was for Artix 3, which is:

```
{http://ws.iona.com/locator}LocatorService
```

You can verify that both locator services are running in the Artix 4 locator by querying the locator with the `it_container_admin` command. For example:

1. Go to the Artix 4 locator demonstration in *`ArtixInstallDir`*`/artix/4.0/demos/advanced/locator`.
2. Load the Artix environment as described in the *Getting Started* manual.
3. Build the C++ demo as described in the demo's Readme.txt file.
4. From the demo's `bin` directory, start the locator with the `start_locator` command.
5. From the `bin` directory, run the following command:

```
it_containter_admin -container ../etc/ContainerService.url
   -listservices
```

6. The following list of service QNames is returned:

```
{http://ws.iona.com/peer_manager}PeerManagerService ACTIVATED
{http://ws.iona.com/2005/11/locator}LocatorService ACTIVATED
{http://ws.iona.com/locator}LocatorService ACTIVATED
```

Supported configurations

The following configurations are supported by the Artix 4 locator service:

- Artix 4 locator, Artix 4 services, Artix 4 clients
- Artix 4 locator, Artix 4 services, Artix 3 clients

Unsupported configurations

The following configurations are *not* supported by the Artix 4 locator service:

- Artix 4 locator, Artix 3 services, Artix 3 clients
- Artix 4 locator, Artix 3 services, Artix 4 clients

The following configurations are *not* supported by Artix 4 service or client applications:

- Artix 3 locator, Artix 4 services, Artix 3 clients
- Artix 3 locator, Artix 4 services, Artix 4 clients
- Artix 3 locator, Artix 3 services, Artix 4 clients

Migration strategies

The Artix 4 locator service is backward compatible by default. There are no configuration steps required to enable backward compatibility in the locator itself.

You can start your Artix 4 services in a way that supports both Artix 4 and Artix 3 client applications. See "Starting Services with Artix 3 Client Support" on page 28.

You can migrate your client applications one at a time to Artix 4 locator compatibility, using the steps described in "Migrating Client Application Code" on page 51.

When you have all Artix client applications migrated to Artix 4, the backward compatibility feature of the Artix 4 locator is no longer necessary for your site. There is no need to disable the backward compatibility feature, and the Artix 4 locator performance is not improved by disabling backward compatibility.

However, if you prefer to disable this feature, you can comment out two lines near the end of your site's Artix configuration file. The Artix configuration file is installed by default in:

```
ArtixInstallDir/artix/4.0/etc/domains/artix.cfg
```

The lines to comment out are in the "Well known Services QName aliases" section near the end of the file. Use the # symbol to comment out the following two lines:

```
bus:qname_alias:locator_oldversion = "{http://ws.iona.com/locator}LocatorService";
...
bus:initial_contract:url:locator_oldversion = "ArtixInstallDir/artix/version/wsdl/oldversion/
    locator.wsdl";
```

CHAPTER 2

Configuring and Deploying the Locator Service

This chapter discusses how to configure and deploy an Artix locator service by editing configuration files.

In this chapter

This chapter discusses the following topics:

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. The Artix container and plug-in architecture is discussed further in “Deploying Services in an Artix Container” in *Configuring and Deploying Artix Solutions.*

Artix configuration concepts

The information in this chapter presumes an understanding of Artix configuration concepts and practices, as described in *Configuring and Deploying Artix Solutions*. See the chapters “Artix Configuration” and “Finding Contracts and References.”

Configuring the locator to run in the container

To configure the locator to run in the Artix container, make sure the `service_locator` plug-in is included in the locator's configuration scope. For example, Example 1 shows the `locator.cfg` file used by the demo in `ArtixInstallDir/artix/version/demos/advanced/locator/etc/`:

Example 1: *Locator demo's locator.cfg file*

```
include "../../../../etc/domains/artix.cfg";

demo
{
  locator
  {
     client
     {
         orb_plugins = ["xmlfile_log_stream", "locator_client"];
     };
     server
     {
         orb_plugins = ["xmlfile_log_stream", "wsdl_publish", "locator_endpoint"];
     };
     service
     {
         orb_plugins = ["xmlfile_log_stream", "wsdl_publish", "service_locator"];
     };
  };
};
```

The portion of Example 1 in blue text shows a service in the scope `demo.locator.service` configured to load the `wsdl_publish` and `service_locator` plug-ins (as well as a logging service plug-in). 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` plug-ins 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 (installed by default in `ArtixInstallDir/artix/version/wsdl/locator.wsdl`), the addressing information is as follows:

Example 2: *Locator Service on Dynamic Port in default locator.cfg*

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

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

The locator service must itself be easily locatable by clients. Starting the locator on a dynamic port means it would start up on a different TCP port with every restart. This is not useful in a production environment.

Configuring a fixed port

There are several ways of deploying the locator on a well-known fixed port:

- You can edit the default `locator.wsdl` contract
- You can create a new `locator.wsdl` contract for your application.
- You can use features of the Artix container to determine the port on which the container deploys the locator.

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:

```
     ArtixInstallDir\artix\version\wsdl\locator.wsdl
```

2. Edit the `soap:address` attribute at the bottom of the contract to specify the desired port in the address. Example 3 shows a modified locator service contract entry. The portion shown in blue has been modified to point to port 9000 on the local computer.

Example 3: *Locator Service on Fixed Port*

```
<service name="LocatorService">
  <port name="LocatorServicePort" binding="ls:LocatorServiceBinding">
    <soap:address location="http://localhost:9000/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, as shown in Example 3.
3. In your Artix 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 = "/opt/app/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 recommended deployment for the locator is in an instance of the Artix container. 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. Query the container with the `it_container_admin` command (or with your own code). Ask the container to publish the live version of the locator WSDL after the container has assigned a port for the locator. For example:

```
it_container_admin -container ../../etc/ContainerService.url
   -publishwsdl -service {http://ws.iona.com/2005/11/
   locator}LocatorService -file ..\..\etc\locator-activated.wsdl
```

   This retrieves the locator's activated WSDL contract. This is the contract in which the default WSDL's port 0 has been dynamically updated with the actual port that the service is using. In this example, `it_container_admin` writes the contract to the `locator-activated.wsdl` file in the `..\..\etc` subdirectory.

3. Finally, you must make sure your clients use the activated WSDL file, now resident in the specified directory, when each client starts up at runtime.

Deploying the locator in the container on a fixed port

As an alternative, you can use the -port option when starting the container 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 now 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 20.

Shutting down the locator

To shut down a container-loaded locator, use the container's shutdown option. For example:

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

Registering a Service with the Locator

Overview

A service does not need to have its implementation changed to work with the Artix locator. All that is required is that the service 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 service that wishes to register itself with the locator must load the `locator_endpoint` plug-in. The `locator_endpoint` plug-in enables the service to register with the running locator. The following example shows the configuration scope of a service that registers with the locator service.

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

Another example is shown in Example 1 on page 17, where a service in the scope `demo.locator.service` is configured to load the `locator_endpoint` plug-in.

Using a copy of locator.wsdl

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

```
bus:initial_contract:url:locator="/opt/local/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 Reference*.

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 allows you to filter the endpoints that are registered by the locator endpoint manager. You can do this by explicitly including or excluding endpoints using configuration variables.

Configuration variables are discussed in detail in *Configuring and Deploying Artix Solutions*. The details of each variable are described in the *Artix Configuration Reference*.

Excluding endpoints to be registered

If there are a small number of endpoints that you want to be filtered out, you can explicitly exclude those endpoints from the locator by using the `exclude_endpoints` configuration variable.

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 an example of this configuration, 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 `include_endpoints` configuration variable.

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"];
```

Note: Combining the `exclude_endpoints` and `include_endpoints` configuration variables is ambiguous and unsupported. If you do this, the application fails to initialize.

Filtering endpoints using wildcards

You can use wildcarded service names with endpoint-filtering configuration variables. 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}*"];
```

Service registration

When a properly configured service 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 service with a locator. The locator generates a pool of references for the service type. When clients make a request for a service, the locator supplies references from this pool using a round-robin algorithm. For more information on load balancing see "Using Load Balancing" on page 25.

Using Load Balancing

Overview

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

Starting to load balance

When the locator is deployed and your services 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 a WSDL contract with a number of ports for the same service and have each service instance start up on a different port.
- Create a number of copies of the Artix contract defining the service, and change the port information so each copy has a separate port address. Then bring up each service 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 must be associated with the same binding and logical interface.

As each service starts up, it automatically registers with the locator. The locator recognizes that the services 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.

Using Fault Tolerance Features

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, the `locator_endpoint` plug-in associated with that endpoint 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 cannot notify the locator, and the locator can unknowingly give a client an invalid reference. This might cause the failure to cascade onto one or more clients.

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 further information on this configuration variable, see the *Artix Configuration Reference*.

Locator service failure

If the locator service itself 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 with the `peer_manager` service, 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 with which the endpoint pings the locator by setting the `plugins:session_endpoint_manager:peer_timeout` configuration variable. The default setting is `4` seconds. For further information on this configuration variable, see the *Artix Configuration Reference*.

Highly available locator cluster

You can configure three or more instances of the locator in a highly available locator cluster. This configuration is illustrated in the high_availability_locator demo.

The high availability features of Artix are discussed in “Deploying High Availability” in *Configuring and Deploying Artix Solutions*.

Starting Services with Artix 3 Client Support

Overview

This section describes how to start Artix 4 services in a way that supports both Artix 4 and Artix 3 client applications.

Building Artix 4 services

There are no required code changes between Artix 3 and Artix 4 for locator-aware Artix services. This means you can implement Artix 4 services from your Artix 3 service code by:

- Regenerating any code generated from WSDL using the Artix 4 code generators.
- Recompiling and linking your service application.
- Making sure the Artix 4 version of the `locator_endpoint` plug-in is loaded at runtime.

Specify the right QName

As described in "Locator service QNames" on page 12, the Artix 4 locator implements both Artix 4 and Artix 3-compatible locator services. If you access the Artix 4 locator using the QName of the Artix 3 locator, then the Artix 4 locator responds as an Artix 3 locator.

Supporting Artix 3 clients

To support Artix 3 client applications from your Artix 4 services, you must:

- Run the Artix 4 locator service.
- Run Artix 4 services.
- Make sure the Artix 3-compatible WSDL published from the Artix 4 locator is accessible to your Artix 3 clients and is in the location that they expect to find it.

Supporting the last bullet point depends on how you implemented the port on which the locator runs:

- By assigning a fixed port number in a copy of `locator.wsdl`
- By retrieving the activated WSDL from the locator and storing it in a location accessible to client applications

If you used locator on a fixed port

The locator demos located_router and locator_load_balancing use the fixed port method. Both demos use a copy of `locator.wsdl` that assigns port 9000. This was true in both Artix 3 and Artix 4 versions of the demo code. Similarly, the high_availability_locator demo uses fixed ports.

Clients of the Artix 3 demo should be able to locate and use the services of the Artix 4 demo without any changes. This is because the Artix 4 locator will run on port 9000, and the Artix 3 clients will look for the locator on port 9000. The Artix 3 clients will make requests using the Artix 3 QName of the locator service. This invokes the Artix 3 compatibility of the Artix 4 locator running at port 9000.

If your own client applications use a fixed port for the locator service, then Artix 3 clients should run without any changes against the Artix 4 locator and service running on the same port.

If you used activated WSDL

The batch file that starts the service in the locator demo queries the locator's container for the locator's WSDL contract, and then writes that activated WSDL to a file. The client batch file then reads the same activated WSDL from the same file.

If your applications use this technique, then you must modify the script that starts the services, and have the WSDL for both Artix 3 and Artix 4 locator services written to different network-accessible locations. Remember to write the Artix 3-compatible WSDL to the location your Artix 3 clients expect to find it.

For example, clients of the Artix 3 locator demo can be made to interoperate with the locator and services of the same-named Artix 4 demo by following these steps. This example uses the Windows version of Artix.

1. This example presumes two Artix installations on the same machine. Only for example purposes, let's say that:
   - Artix 4 is installed in C:\IONA
   - Artix 3 is installed in C:\IONA3
2. Copy `run_cxx_server.bat` to a new file. Let's call in `4-3_interop.bat`.
3. Add one extra line to `4-3_interop.bat`, as described and shown below.
4. Create a new `4-3_servers.bat` that calls `4-3_interop.bat` five times with five arguments, in the same way that `run_cxx_servers.bat` does.

5. Run the test batch files in the following sequence. In command prompt window 1:

   Run `start_locator.bat`
   Run `4-3_servers.bat`
   Run `run_cxx_client.bat`
   Run `run_dotnet_client.bat`
   Run `run_java_client.bat` five times with five arguments

6. Open command prompt window 2 and change to the Artix 3 locator demo's `bin` directory.

7. In command prompt window 2:

   Run `run_cxx_client.bat`
   Run `run_java_client.bat` five times with five arguments
   Run `run_dotnet_client.bat`

The line to add to `4-3_interop.bat` runs `it_container_admin` a second time, requesting WSDL using the old locator's QName:

```
-service {http://ws.iona.com/locator}LocatorService
```

Another argument writes the resulting WSDL to the location that the Artix 3 locator demo expects to find and use it:

```
-file /iona3/artix/3.0/demos/advanced/locator/etc/
   locator-activated.wsdl
```

The `4-3_interop.bat` file now looks like the following example. The newly added line is highlighted in boldface.

```
@echo off
@setlocal
call "../../../../bin/artix_env.bat";

IF "%1"=="blocking" (
SET DEMO_START=
SHIFT /1
) ELSE (
SET DEMO_START=start
)

IF "%1"=="corba" (GOTO runserver)
IF "%1"=="soaphttp" (GOTO runserver)
IF "%1"=="soaptunnel" (GOTO runserver)
IF "%1"=="fixedhttp" (GOTO runserver)
IF "%1"=="fixedtunnel" (GOTO runserver)

echo valid transports are corba soaphttp soaptunnel fixedhttp
   fixedtunnel
GOTO :end

:runserver
cd ..\cxx\server
it_container_admin -container ../../etc/ContainerService.url
   -publishwsdl -service {http://ws.iona.com/2005/11/
   locator}LocatorService -file ..\..\etc\locator-activated.wsdl

it_container_admin -container ../../etc/ContainerService.url
   -publishwsdl -service {http://ws.iona.com/
   locator}LocatorService -file /iona3/artix/3.0/demos/advanced/
   locator/etc/locator-activated.wsdl

%DEMO_START% server.exe %1 -ORBname demo.locator.server
   -ORBdomain_name locator -ORBconfig_domains_dir ../../etc
   -BUSservice_contract ../../etc/locator-activated.wsdl

GOTO end

:end
@endlocal
```

CHAPTER 3

Using the Locator from an Artix Client

This chapter describes the configuration and programming steps to enable an Artix client application to make use of a deployed Artix locator service.

In this chapter

This chapter discusses the following topics:

Implementing a C++ Client

Overview

This section shows how to configure client applications to load the client-side locator plug-in, and shows how to write client code in C++ that uses an Artix locator service to locate and connect to a target service of interest.

Artix configuration concepts

The information in this section presumes an understanding of Artix configuration concepts and practices, as described in *Configuring and Deploying Artix Solutions*. See the chapters “Artix Configuration” and “Finding Contracts and References.”

Configuring a client

To use a deployed locator service, and to make sure the connection between client and service is robust, configure client applications to load the `locator_client` plug-in.

An example is shown in Example 1 on page 17, where client applications in the scope `demo.locator.client` are configured to load the `locator_client` plug-in.

Note: Artix releases prior to 4.0 did not use the `locator_client` plug-in, or any plug-in, for clients of the locator.

Client plug-in features

The `locator_client` plug-in is responsible for implementing the initial connection to the locator service to obtain a reference to the target service. You must also write client code that makes a call to resolve the initial reference to the target service. However, it is the plug-in that implements the actions initiated by that call.

After the initial connection to the target service, the `locator_client` plug-in is responsible for maintaining a robust connection. When the `locator_client` plug-in detects a dropped connection to a service, it queries the locator for another instance of the same service. If there are no more service instances, the client-side plug-in continuously retries connecting to the original service.

There are no configuration variables for the `locator_client` plug-in.

Overview of C++ client code

In general, the steps each locator client applications must take are:

- Invoke `IT_Bus::Bus::resolve_initial_reference()` on the target service's QName.
- Using the returned reference, invoke `IT_Bus::ClientProxyBase()` to set up a proxy to the target service.

The `locator_client` plug-in does all the work behind the scenes of connecting to the locator service to obtain a reference to the target service.

Note: In Artix 3, locator client application code had to run `resolve_initial_reference()` against the locator service itself, and to set up a proxy to the locator. These steps are no longer necessary in Artix 4 locator client applications.

C++ Example

The locator client in Example 4 is a small, complete application designed to work in the context of the locator demonstration in `*ArtixInstallDir*/artix/*version*/demos/advanced/locator`.

See "Explanation of Example 4" on page 37 for notes on this example.

Example 4: *Locator client example in C++*

```
//
// C++ locator example client code
//
#include <it_bus/bus.h>
#include <it_bus/exception.h>
#include <it_cal/iostream.h>
#include "SimpleServiceClient.h"

IT_USING_NAMESPACE_STD
using namespace SimpleServiceNS;
using namespace IT_Bus;
using namespace WS_Addressing;

int main(int argc, char* argv[])
{
    cout << endl << "SimpleService C++ Client";

    // Initialize the Artix bus.
    IT_Bus::Bus_var bus;
```

Example 4: *Locator client example in C++*

```
     try
     {
         cout << endl << "Initializing the bus.";
1        bus = IT_Bus::init(argc, (char **)argv,
                               "demo.locator.client");
     }
     catch (IT_Bus::Exception& err)
     {
         cout << endl << "Caught unexpected exception while "
              << "initializing the bus: "
              << endl << err.message() << endl;
         return -1;
     }
     // Convert the target service name and namespace to a QName.
     // QName (namespace_prefix, local_part, namespace_uri);
2    QName service_qname("", "SOAPHTTPService",
                         "http://www.iona.com/FixedBinding");
     try
     {   // Get a WS-A reference to the target service.
3        EndpointReferenceType ep_ref;
         cout << endl << "Resolving "
              << service_qname.get_local_part()
              << " service in the locator.";
4        if (!bus->resolve_initial_reference(
                                service_qname, ep_ref))
         {
            cout << endl
                 << "Unable to resolve a reference using "
                 << "the locator resolver." << endl;
            return -1;
         }
         // Construct a new proxy to the target service
         // with the result from the locator.
         cout << endl << "Initializing a proxy with the "
                      << "results from the locator.";
         // SimpleServiceClient() is defined in
         // SimpleServiceClient.cxx from the Artix 4.0 demo in
         // <topDir>/artix/4.0/demos/advanced/locator/cxx/client
5        SimpleServiceClient simple_client(ep_ref);
```

Example 4: *Locator client example in C++*

```
        // Use the new proxy to invoke the say_hello operation on
        // the target service.
        cout << endl << "Invoking say_hello on the service "
             << service_qname.get_local_part() << ".";
        String my_greeting = String("Greetings from ") +
                             service_qname.get_local_part();
        String result;
6       simple_client.say_hello(my_greeting, result);

        cout << endl << "The say_hello operation returned: "
             << endl << "     " << result << "!";
    }

    catch (IT_Bus::Exception& err)
    {
        cout << endl
             << "Caught unexpected exception while invoking "
             << "on the endpoint: "
             << endl << err.message() << endl;
        return -1;
    }
    cout << endl << endl;
    return 0;
}
```

Explanation of Example 4

The following points refer to the number labels in Example 4.

1. This example hard codes an association with the `demo.locator.client` configuration scope by means of an argument to the `IT_Bus::init()` call. In a production application, you are more likely to specify the scope in an `-ORBname` parameter when invoking the client executable.

   The association with the configuration scope is what ensures that the `locator_client` plug-in is loaded at runtime. This example presumes a configuration file like the one shown in Example 1 on page 17.

2. This line constructs a QName for the target service to which this client application will connect at runtime. The components of the QName are defined in the target service's WSDL contract. In this case, the target service's contract is in *`ArtixInstallDir`*`/artix/`*`version`*`/demos/advanced/locator/etc/simple_service.wsdl`.

3. The reference is declared as an instance of the WS-Addressing standard's `EndpointReferenceType`.
4. This line invokes `resolve_initial_reference()`, passing the QName of the target service and an instance of the endpoint reference class.
5. The `SimpleServiceClient` class is defined in the locator demo in *`ArtixInstallDir`*`/artix/`*`version`*`/demos/advanced/locator/cxx/client`. This class is a wrapper for invoking `IT_Bus::ClientProxyBase()` to set up a client proxy. In this case, the proxy is set up for the target service defined in the QName set up in line 2.
6. Now that the client proxy to the target service is established, the code can invoke operations of the target service. The say_hello operation is defined in the target service's WSDL contract, `simple_service.wsdl`.

Compiling and running Example 4

The code in Example 4 can be saved to a file, then compiled and run in the context of the locator demo, as follows:

- Save the code to a file in *`ArtixInstallDir`*`/artix/`*`version`*`/demos/advanced/locator/cxx/client`.
- Create a separate make file based on the `Makefile` in that directory. Name the output executable something other than `client[.exe]`.
- Invoke `nmake -f` *`yourmakefile`*. (Windows) or `make -f` *`yourmakefile`* (UNIX).
- Create a batch file or shell script to run your executable, based on the `run_cxx_client[.bat]` in the demo's `bin` directory.
- Start the locator demo with `start_locator[.bat]`.
- Start the example services with `run_cxx_servers[.bat]`.
- Run the example's batch file or shell script.

When invoked as above, the example code produces output like the following:

```
SimpleService C++ Client
Initializing the bus.
Resolving SOAPHTTPService service in the locator.
Initializing a proxy with the results from the locator.
Invoking say_hello on the service SOAPHTTPService.
The say_hello operation returned:
     Greetings from SOAPHTTPService!
```

Implementing a Java Client

Overview

This section shows how to configure client applications to load the client-side locator plug-in, and shows how to write client code in Java that uses an Artix locator service to locate and connect to a target service of interest.

Artix configuration concepts

The information in this section presumes an understanding of Artix configuration concepts and practices, as described in *Configuring and Deploying Artix Solutions*. See the chapters “Artix Configuration” and “Finding Contracts and References.”

Configuring a client

To use the a deployed locator service, and to make sure the connection between client and service is robust, configure client applications to load the `locator_client` plug-in.

An example is shown in Example 1 on page 17, where client applications in the scope `demo.locator.client` are configured to load the `locator_client` plug-in.

Note: Artix releases prior to 4.0 did not use the `locator_client` plug-in, or any plug-in for clients of the locator.

Client plug-in features

The `locator_client` plug-in is responsible for implementing the initial connection to the locator service to obtain a reference to the target service. You must also write client code that makes a call to resolve the initial reference to the target service. However, it is the plug-in that implements the actions initiated by that call.

After the initial connection to the target service, the `locator_client` plug-in is responsible for maintaining a robust connection. When the `locator_client` plug-in detects a dropped connection to a service, it queries the locator for another instance of the same service. If there are no more service instances, the client-side plug-in continuously retries connecting to the original service.

Overview of Java client code

In general, the steps each locator client applications must take are:

- Invoke `com.iona.jbus.Bus.resolveInitialEndpointReference()` on the target service's QName.
- Using the returned reference, use the standard JAX-RPC or Artix API methods of setting up a proxy to the target service. See "Writing the Consumer Code" in *Developing Artix Applications in Java*.

The `locator_client` plug-in does all the work behind the scenes of connecting to the locator service to obtain a reference to the target service.

Note: In Artix 3, locator client application code had to build a proxy for the locator service itself, and then use that proxy to invoke operations from the locator. These steps are no longer necessary in Artix 4 locator client applications.

Java Example

The locator client in Example 5 is a small, complete application designed to work in the context of the locator demonstration in *`ArtixInstallDir`*`/artix/`*`version`*`/demos/advanced/locator`.

See "Explanation of Example 5" on page 42 for notes on this example.

Example 5: *Locator client example in Java*

```
//
// Java locator example client code
//
import java.util.*;
import java.io.*;
import java.net.*;
import java.rmi.*;

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

import com.iona.jbus.Bus;

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

Example 5: *Locator client example in Java*

```
       System.out.println("\nSimpleService Java Client");

       // Initialize the Artix bus.
       System.out.println("Initializing the bus.");
1      Bus bus = Bus.init(args);

       // Convert the target service name and namespace to a QName.
       QName service_qname = new
2                QName("http://www.iona.com/FixedBinding",
                                       "SOAPHTTPService");
       // Get a WS-A reference to the target service.
       System.out.println("Resolving " +
                          service_qname.getLocalPart() +
                          " service in the locator.");
3     com.iona.schemas.wsaddressing.EndpointReferenceType ep_ref =
4              bus.resolveInitialEndpointReference(service_qname);

       if (ep_ref == null) {
                  System.err.println("Error: Could not resolve " +
                                 "endpoint using locator resolver"
                                 + service_qname.getLocalPart()); }

       System.out.println("Initializing a proxy with the results "
                          + "from the locator.");
5      SimpleService simple_client = null;
       URL wsdlLocation = null;
       try {
6          wsdlLocation = new URL(ep_ref.getMetadata()
                                         .getWsdlLocation());
           }
       catch (java.net.MalformedURLException ex)
           {
           wsdlLocation =  new File(ep_ref.getMetadata()
                                           .getWsdlLocation())
                                           .toURL();
           }
       ServiceFactory factory = ServiceFactory.newInstance();
7      Service service = factory.createService(wsdlLocation,
                                                service_qname);
       QName portName = new QName("", "SOAPHTTPPort");
       simple_client =
                   (SimpleService)service.getPort(portName,
                                              SimpleService.class);
```

Example 5: *Locator client example in Java*

```
      if (simple_client == null) {
        System.err.println("Couldn't create SimpleService client "
                           + "proxy from locator");
        }

      // Use the new proxy to invoke the say_hello operation on
      // the target service.
      String result;
      String greeting = "Greetings from SOAPHTTPService!";
      System.out.println("Invoking say_hello on the service " +
                         service_qname.getLocalPart() +".");
8     result = simple_client.say_hello(greeting);
      System.out.println("The say_hello operation returned: \n"
                         + "     " + result);
   }
 }
```

Explanation of Example 5

The following points refer to the number labels in Example 5.

1. This example initializes the bus with whatever arguments are passed on the command line. The command line arguments must include `-ORBname demo.locator.client` to associate this client application with the configuration scope `demo.locator.client`.

   The association with the configuration scope is what ensures that the `locator_client` plug-in is loaded at runtime. This example presumes a configuration file like the one shown in Example 1 on page 17.
2. This line constructs a QName for the target service to which this client application will connect at runtime. The components of the QName are defined in the target service's WSDL contract. In this case, the contract is in *`ArtixInstallDir`*`/artix/`*`version`*`/demos/advanced/locator/etc/simple_service.wsdl`.
3. The reference is declared as an instance of the WS-Addressing standard's `EndpointReferenceType`.
4. This line invokes `com.iona.jbus.Bus.resolveInitialEndpointReference()`, passing the QName of the target service. The return value is an instance of the endpoint reference class.

5. The `SimpleService` class is defined in the locator demo in *`ArtixInstallDir`*`/artix/`*`version`*`/demos/advanced/locator/java/client`. This class is a wrapper for `java.rmi.Remote`.
6. The section of code at line 6 extracts the WSDL location for the target service from the endpoint reference for that service returned from the locator.
7. The code at line 7 invokes the JAX-RPC method of setting up a client proxy. In this case, the proxy is set up for the target service defined in the QName set up in line 2.
8. Now that the client proxy to the target service is established, the code can invoke operations of the target service. The say_hello operation is defined in the target service's WSDL contract, `simple_service.wsdl`.

Compiling and running Example 5

The code in Example 5 can be saved to a file, then compiled and run in the context of the locator demo, as follows:

- Save the code to a file in *`ArtixInstallDir`*`/artix/`*`version`*`/demos/advanced/locator/java/client`.
- Return to the top-level directory of the locator demo and invoke `ant`.
- Create a batch file or shell script to run your executable, based on the `run_java_client[.bat]` in the demo's `bin` directory.
- Start the locator demo with `start_locator[.bat]`.
- Start the example services with `run_cxx_servers[.bat]`.
- Run the example's batch file or shell script.

When invoked as above, the example code produces output like the following:

```
SimpleService Java Client
Initializing the bus.
Resolving SOAPHTTPService service in the locator.
Initializing a proxy with the results from the locator.
Invoking say_hello on the service SOAPHTTPService.
The say_hello operation returned:
     Greetings from SOAPHTTPService!
```

Querying a Locator Service

Overview

Starting with Artix 4.0, the locator has extended query functionality, compared to the basic `listEndpoints` operation offered in prior releases. The locator query capabilities are implemented as the `queryEndpoints` operation, which uses as its input parameter a select element defined in an extensible XML schema, `locator-query.xsd`.

Demonstration code

The querying functionality of the Artix 4 locator is illustrated in the `locator_query` demonstration example. See *`ArtixInstallDir`*`/artix/`*`version`*`/demos/advanced/locator_query`.

Filtered and unfiltered lists of services

To use the query functionality, follow these overall steps in your client application code:

1. Obtain a reference to the locator service and create a client proxy to the locator[3].
2. To obtain an unfiltered list of the services registered with that locator, invoke the locator's `listEndpoints` operation.
3. To obtain a filtered list of registered services, invoke the locator's `queryEndpoints` operation, passing it one or more query filters.

Extensible query language

The query language used by the `queryEndpoints` operation is governed by an XML Schema, which is installed by default in *`ArtixInstallDir`*`/artix/`*`version`*`/schemas/locator-query.xsd`.

The C++ data types used in the examples in this section are from code generated from this schema (or from `locator.wsdl`, which includes this schema). Artix does not ship with code generated from this schema or WSDL, so it is the Artix developer's responsibility to generate code from the schema or WSDL and make use of it.

3. When you use the locator in its usual role of returning a reference to a target service of interest, you do not need to create a proxy to the locator service itself. This is illustrated in "Implementing a C++ Client" on page 34 and "Implementing a Java Client" on page 39. It is only when using the query functionality of the locator that you need to create a proxy to the locator itself.

Because the query language is in a schema, you can extend the schema to add new query functionality.

The contents of the `locator_query.xsd` schema are shown in Example 6:

Example 6: *Contents of locator-query.xsd*

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
   targetNamespace="http://ws.iona.com/2005/11/locator/query"
      elementFormDefault="qualified"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:tns="http://ws.iona.com/2005/11/locator/query">
  <xs:simpleType name="FieldEnumeratedType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="GROUP"/>
      <xs:enumeration value="SERVICE"/>
      <xs:enumeration value="PORTNAME"/>
      <xs:enumeration value="INTERFACE"/>
      <xs:enumeration value="BINDING"/>
      <xs:enumeration value="EXTENSOR"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="FilterFieldType">
    <xs:union memberTypes="tns:FieldEnumeratedType xs:string"/>
  </xs:simpleType>
  <xs:complexType name="FilterType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="field" type="tns:FilterFieldType"
            use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="QuerySelectType">
    <xs:sequence>
      <xs:element name="filter" type="tns:FilterType" minOccurs="0"
          maxOccurs="unbounded"/>
      <xs:any namespace="##other" minOccurs="0" processContents="lax"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="select" type="tns:QuerySelectType"/>
</xs:schema>
```

Query functionality

The target namespace of the `locator-query.xsd` schema is `http://ws.iona.com/2005/11/locator/query`. The `query:select` element of type `query:QuerySelectType` is a sequence of filters. It is extensible insofar as it can support future `xs:any` elements without breaking compatibility. In the current implementation, the locator service ignores all `xs:any` elements that may be present within a `select` element.

A filter is a pair of *type* and *value*. The *value* is a string; some filters use QName values represented as strings in canonical form:

```
[{<namespace>}]<local-part>
```

The logic to convert QNames to and from canonical string representation is available from the `IT_Bus::QName` type (as shown in the example in this section).

The *type* of a filter is one of the `query:FieldEnumeratedType` values. The filter type is extensible by allowing any other field type. Extensibility was achieved by making the Filter *type* a union of the supported enumerated type and a string. Any value different from the ones present in the enumerated type is ignored by the current locator implementation.

The value of a filter could be either a string or a QName, depending on the filter type. When the value is a QName, you still needs to pass it as a string using its canonical value.

The matching rules for the supported filter types are shown in the following table. There is no wildcard support in these filter types, so the search text must be exact.

Filter type	Format	Filter the returned list of services by:
GROUP	`xs:string`	The case sensitive name of a group of services you are seeking. (Service group membership is defined in each service's WSDL contract or in an Artix configuration file as described in "Service groups" on page 47.)
SERVICE	`xs:QName`	The QName of the service you are seeking.
PORTNAME	`xs:string`	The case sensitive name of at least one of the ports in the service you are seeking.
INTERFACE	`xs:QName`	The QName of the `portType` associated with a binding, which is itself associated with at least one of the ports in the service you are seeking.

Filter type	Format	Filter the returned list of services by:
BINDING	`xs:string`	The QName of the binding associated with at least one of the ports in the service you are seeking.
EXTENSOR	`xs:string`	The QName of an extensor contained in at least one of the ports in the service you are seeking.

Service groups

Starting with Artix 4.0, you can assign arbitrary group membership to services. This feature is used in combination with the locator's query functionality. For example, you could query the locator to ascertain which services belong to which groups.

There is no restriction on assigning services to groups in different processes. It is valid to have services in the same process belong to different groups, or to no group at all. It is valid for services in different processes to belong to the same group. By default, a service belongs to no group.

A service can be assigned to a group by means of a WSDL extension or by means of configuration.

Assigning group membership with a configuration variable

The preferred method of assigning services to groups is performed in an Artix configuration file, using the `service_group` configuration variable.

Using the QName alias for a service in the configuration file, specify the `service_group` variable and assign an arbitrary string as the group name.

In the following example, the first line defines the QName alias `corba_svc`. The second line assigns the `corba_svc` service to the group named `CORBAGroup`.

```
bus:qname_alias:corba_svc = "{http://demo.iona.com/advanced/LocatorQuery}CORBAService";
...
plugins:locator:service_group:corba_svc = "CORBAGroup";
```

Note: Configuration-assigned group membership takes precedence over WSDL-assigned group membership.

You can define a global group for all services associated with the current bus. All services that do not have a group definition in their WSDL or configuration then belong to the global group by default.

```
plugins:locator:service_group = "<default-group-name>";
```

Assigning group membership in WSDL

You can use an Artix WSDL extension to assign a service to a group in the service's WSDL contract.

The WSDL extension is defined in a new schema under the `types` section in `locator.wsdl`:

```
<xs:schema targetNamespace="http://ws.iona.com/2005/11/locator/extensions">
    <xs:element name="group" type="xs:string"/>
</xs:schema>
```

This allows service WSDL contracts to use the `name=` attribute, as shown in this example taken from the locator_query demo.

```
xmlns:locx="http://ws.iona.com/2005/11/locator/extensions"
...
<service name="CORBAService">
    <locx:group>QUERY-DEMO</locx:group>
    <port binding="tns:SimpleServicePortType_CORBABinding" name="CORBAPort">
        <corba:address location="file:../../corba_server.ior"/>
        <corba:policy poaname="corbaport"/>
    </port>
</service>
```

Locator query example with single query

The following C++ code fragment illustrates the use of the locator's query functionality. This example uses a single query filter:

```
// Create a query
    QuerySelectType select;
    FilterType filter;
    FilterFieldType fld;

    fld.setFieldEnumeratedType(
        FieldEnumeratedType(FieldEnumeratedType::GROUP));
    filter.setfield(fld);
    filter.setvalue("SAMPLE-VALUE");
    select.getfilter().push_back(filter);

    // Create a proxy for the locator.
    // (This assumes that the bus already been initialized)
    Reference locator_ref;
    bus->resolve_initial_reference(LOCATOR_SERVICE_NAME,
                                   locator_ref);
    LocatorServiceClient locator_client(locator_ref);

    // Invoke
    ElementListT<endpoint> result;
    locator_client->queryEndpoints(select, result);

    // Use the result in some way ...
```

Locator query example with multiple queries

The locator supports queries based on multiple filters. The filters restrict the endpoints in the result set to those endpoints that match the value in each filter. They act as a composite filter with an implicit AND operator.

Filters have a type and a value. There are no restrictions on mixing different filters based on their type. It is valid to add filters of the same type.

The following C++ code fragment illustrates the use of the locator's query functionality with mutiple query filters.

```
QName sample_portType("", "MyPortType", "http://www.example.com/
    demo");

QuerySelectType select;
FilterType filter;
FilterFieldType fld;

fld.setFieldEnumeratedType(
    FieldEnumeratedType(FieldEnumeratedType::GROUP));
filter.setfield(fld);
filter.setvalue("SAMPLE-VALUE");
select.getfilter().push_back(filter);

fld.setFieldEnumeratedType(
    FieldEnumeratedType(FieldEnumeratedType::INTERFACE));
filter.setfield(fld);
filter.setvalue(sample_portType.get_as_canonical_string());
select.getfilter().push_back(filter);
```

Migrating Client Application Code

Overview

With the release of Artix 4.0, the following changes might affect any existing Artix client applications:

- Locator WSDL operation name changes were made in compliance with the wrapped doc-literal convention.
- Artix switched from using a proprietary reference format to using the standard WS_Addressing endpoint reference format.
- Locator client applications are now configured to load the `locator_client` plug-in. This plug-in takes over the tasks of creating a proxy to the target service and of monitoring and maintaining the connection to the target service. These tasks were formerly the responsibility of client application code.

For WS_Addressing migration information, see the chapters “Endpoint References” in *Developing Artix Application in C++*, or “Using Endpoint References” in *Developing Artix Applications in Java*.

Locator WSDL operation name changes

Artix 4.0 includes a new version of `locator.wsdl`. The operations contained in this new WSDL contract conform to the wrapped doc-literal convention. Specifically:

- The `lookup_endpoint()` operation, which returned an Artix Reference, has been replaced with `lookupEndpoint()`, which returns a WS-Addressing type `EndpointReferenceType`.
- The `list_endpoints()` operation has been replaced with `listEndpoints()`.
- A new `queryEndpoints()` operation has been added.

The new `locator.wsdl` file is located in the following directory of your Artix installation:

```
ArtixInstallDir/artix/version/wsdl
```

Old and new locator WSDL contracts supported

In Artix 4.0, by default the locator service resolves its service contract against the new `locator.wsdl` file and, therefore, supports the new operation names.

Artix 4.0 also includes a copy of the Artix 3.x `locator.wsdl` file in:

```
ArtixInstallDir/artix/version/wsdl/oldversion
```

The Artix 4.0 configuration file, `artix.cfg`, resolves which `locator.wsdl` contract to use by distinguishing the QName with which the locator service is called. The default `artix.cfg` file contains the following lines:

```
bus:qname_alias:locator_oldversion = "{http://ws.iona.com/locator}LocatorService";
bus:qname_alias:locator = "{http://ws.iona.com/2005/11/locator}LocatorService";
...
bus:initial_contract:url:locator_oldversion = "ArtixInstallDir/artix/version/wsdl/oldversion/
    locator.wsdl";
bus:initial_contract:url:locator = "ArtixInstallDir/artix/version/wsdl/locator.wsdl";
```

Thus, if client application code requests a reference using the QName `{http://ws.iona.com/locator}LocatorService`, then any request for the locator's initial contract is directed to the 3.x version of `locator.wsdl` in the `wsdl/oldversion` directory.

By using the Artix 4 version of the locator QName, `{http://ws.iona.com/2005/11/locator}LocatorService`, any request for the locator's initial contract is directed to the 4.x version of `locator.wsdl`.

Migrating client application code to Artix 4

As described in "Migrating from Previous Versions" on page 12, the Artix 4 locator supports the use of unmodified Artix 3 client applications. This allows you to put your first migration efforts into upgrading your locators and services to Artix 4. Once those tasks are complete, you can migrate your client applications as follows:

1. Edit your configuration files to make sure the `locator_client` plug-in is loaded in the configuration scope(s) used by your locator clients. See "Configuring a client" on page 34.
2. If your code directly invokes any operations of the `locator.wsdl` contract, update the operation names as described in "Locator WSDL operation name changes" on page 51.

3. For client applications in C++, simplify your client application code as described below.

   In Artix 3, the coding steps that every locator client application had to take were the following:

   i. Invoke `IT_Bus::Bus::resolve_initial_reference()` on the locator's QName.

   ii. Using the returned reference, invoke `IT_Bus::ClientProxyBase()` to set up a proxy to the locator.

   iii. Using the proxy, invoke the locator's `lookup_endpoint` operation to get a reference to the target service.

   iv. Using the reference returned by the locator, invoke `ClientProxyBase()` to set up a proxy to the target service.

   In Artix 4, because the `locator_client` plug-in is doing some of the work, the coding steps are shortened to the following[4]:

   i. Invoke `IT_Bus::Bus::resolve_initial_reference()` on the target service's QName.

   ii. Using the returned reference, invoke `IT_Bus::ClientProxyBase()` to set up a proxy to the target service.

4. For client applications in Java, change your client application code as described below.

   i. Remove code that resolves a reference to the locator and sets up a proxy to the locator service itself.

   ii. Instead of invoking the locator service's `lookup_endpoint` operation to get a reference, use `resolveInitialEndpointReference` to directly return a reference to the target service.

   iii. Use members of the endpoint reference class to extract from the returned reference the location of the WSDL for the target service.

   iv. Create a client proxy for the target service.

4. If your application invokes the `listEndpoints` or `queryEndpoints` operations of the locator service, then you must still create a proxy to the locator service. This is described in "Querying a Locator Service" on page 44.