IONA

Artix™ ESB

Locator Guide
Version 5.0, July 2007

Making Software Work Together™

Contents

Preface

What is Covered in This Book

This book describes the theory and operation of 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, Artix 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 consumers.
- Chapter 3, Using the Locator from an Artix Consumer, 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 consumers.

- Chapter 4, Using the Locator from a Non-Artix Client, describes how to use an Artix locator service from consumers generated by other SOA systems, using .NET and Axis consumers as examples.

The Artix Documentation Library

For information on the organization of the Artix library, the document conventions used, and finding additional resources, see Using the Artix Library.

Artix Locator Introduction

The Artix locator service provides a way for consumers to connect to services that is independent of the service location. 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:

What is the Locator Service?

Overview

The Artix locator is a Web service that provides Web service consumers with a mechanism to discover service endpoints at runtime. The locator isolates consumers from knowledge of a service endpoint's physical location. The locator allows service endpoints to advertise their availability to consumers.

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. Consumers 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 consumers in any way.

Service endpoint grouping and organizing

You can query the locator for its list of currently registered services, and you can filter and organize 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. This allows you to add structure to the way consumers view the locator service data.

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 or random algorithm to select the service instance whose reference is returned to requesting consumers. 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 between the service endpoints and the locator. The service-side locator plug-in is tolerant of restarts of the locator service, and automatically reregisters its endpoints when the locator

restarts. The locator service is tolerant of badly behaved services that do not register their endpoints on shutdown: the locator automatically deregisters the service from the locator in the event of a failure on the service side.

High availability locator

The Artix locator can be configured in a high availability configuration with two or more slave locators coordinating with one master locator. This allows you to distribute many instances of the locator across your service network, all of which share the same reliable repository.

How the Locator Works

Overview

The Artix locator service is a standalone service that holds a repository of active service endpoints on your service network. The Artix locator service functionality is implemented as a number of Artix plug-ins. In order to use this functionality, you need participation from consumers, service endpoints, and the central locator service. Therefore, to implement locator functionality on your network, you must enable Artix plug-ins for each of these points on your network, and configure them accordingly. You do not have to write any code to use this functionality. Since the locator service is described by a WSDL contract, you can also choose to use the locator functionality directly in your applications.

How the locator works

You can make your applications locator-aware simply by running an instance of the locator service and changing the configuration of your applications. This is true whether the configuration is the ART-based configuration of the Artix C++/JAX-RPC runtime, or the Spring-based configuration of the Artix Java JAX-WS runtime introduced in Artix 5.0.

Services are made locator-aware by means of configuration statements in the Artix configuration files associated with those services, or in the Spring configuration files of Artix Java JAX-WS services. A locator-aware service automatically registers itself with the locator during service startup. The locator and its registered services 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.

Consumers are also made locator-aware by means of configuration statements in their associated configuration files. Consumers are required to initialize their proxies in a certain way to take advantage of this functionality. A locator-aware consumer automatically contacts the locator service when it is setting up its proxies. The consumer-side locator plug-in contacts the locator service and provides the QName for the desired service endpoint. The locator returns a reference, which contains the addressing details needed to invoke the target service. This is passed to the consumer, which then instantiates a proxy to the target service.

The consumer-side locator plug-in performs a simple lookup of service endpoints based on the target service's QName. If a consumer needs to use the more advanced querying operations of the locator service, then such queries needs to be coded directly into the consumer.

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 that represents the addressing of the target service endpoint. All proxies in Artix have constructors that take a WS-Addressing reference as a parameter. The reference contains sufficient information to allow the consumer to create a functional proxy to a service endpoint.

Registering endpoints

An Artix service registers its endpoints with the locator in order to make them accessible to Artix consumers. When a service registers an endpoint in the locator, it creates an entry in the locator's list of services. The entry associates a service QName with a reference for that endpoint.

Looking up references

An Artix consumer 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 consumer can then establish a remote connection to the target service by instantiating a consumer proxy object. This procedure is independent of the type of binding or transport protocol.

Looking up references can be performed on behalf of the consumer by configuring it to use the locator consumer plug-in. As an alternative, you can write lookup code directly using the WSDL interface that the locator service exposes.

Automatic load balancing

If multiple endpoints are registered against the same service QName in the locator, the locator employs a random or 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/newcustomer

- Service: AddNewCustomer
  WSDL location: http://backuphost:2900/service/newcustomer

When an Artix consumer 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 consumers are handled by Artix plug-ins. The locator-related plug-ins 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 consumers.

This plug-in is normally deployed as a standalone service typically using the Artix container. All consumer and service endpoints need to be aware of this shared service on startup.

This plug-in is also responsible for making sure its data is reliable. It removes service endpoints from its repository if it believes they are inactive.

Locator endpoint manager plug-in (`locator_endpoint`)

This is the portion of the locator that resides with the service endpoints you want registered in the locator service. It registers its service endpoint with the locator service when they become active, and it deregisters them when they are shut down.

This plug-in is also responsible for registering its endpoints with the locator if the locator service is restarted.

Locator consumer plug-in (`locator_client`)

This plug-in queries the locator service and returns a reference to the target service.

When you load an instance of the `service_locator` or `locator_endpoint` plug-in into an Artix container, the container automatically loads the `peer_manager` plug-in. The `peer_manager` plug-in is responsible for the fault tolerant behavior of the locator service.

The `service_locator` and `locator_endpoint` plug-ins are optionally used alongside the `wsdl_publish` plug-in. IONA strongly recommends using the `wsdl_publish` plug-in when working with the locator.

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, C++ Runtime.

The locator service plug-in automatically loads the `peer_manager` service, and if specified, the `wsdl_publish` service, into the same Artix container. The container's URL is published in some way so that other processes can locate the container.

The container process selects a TCP port on which to place the locator service[1] (unless you specify an exact port in configuration). Consumer processes can use the published URL of the container to ask the container to send the locator service's URL, its WSDL contract, or a reference to the locator.

An Artix service process can be deployed in a standalone server or in another Artix container. For clarity, the examples in this book and in the locator demonstration code show the service deployed in a standalone server. The recommended approach is to use a container when developing your services.

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.

1. This locator service is usually run on the same port as the container itself. Thus, for example, if you query the container at localhost:9300, chances are good the locator service will be found at localhost:9300 as well.

Thereafter, the `peer_manager` plug-ins associated with both the `locator_service` and `locator_endpoint` plug-ins periodically ping each other to make sure both parties are still active. If either party detects that the other is inactive, action is taken to remedy the situation. For the `locator_service` side, the inactive endpoints are removed; for the `locator_endpoint` side, the plug-in attempts to re-establish communication in case the locator service is restarted.

The **locator_client** plug-in is loaded into a consumer by means of configuration. This plug-in handles the details of getting a reference to the target service. The consumer uses this reference to create a proxy to the target service.

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 68.

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 in that configuration file.

The configuration and setup of the Artix locator service is described in Chapter 2 on page 27.

Using the locator from consumers

You can configure consumers to make use of the Artix locator with two different approaches:

1. Add the consumer-side locator plug-in to the configuration of consumers, by editing your Artix configuration file. Your consumer-side code will take advantage of this plug-in as long as it uses the standard Artix methods of resolving the initial reference to its target service.

2. Write code that queries the locator directly using the WSDL contract that defines the locator service. Using this method, you can perform simple endpoint lookups as well as advanced querying.

The configuration and coding of locator-enabled consumers is described in .

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 consumer process to look up an endpoint from the locator based on the target service's QName.
- **listEndpoints** A request-response operation used by a consumer process to list all endpoints registered with the locator.
- **queryEndpoints** A request-response operation used by a consumer 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_endpoint` plug-in in communicating with the `locator_service` plug-in:

- **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 Sample Code

Overview

Artix includes code samples that illustrate various Artix features. Read the `Readme.txt` file in each sample's directory for instructions on building and running that sample.

C++/JAX-RPC locator samples

Five demos illustrate different aspects of the locator as used with the C++/JAX-RPC runtime. The locator-related demos for this runtime are installed in subdirectories of:

    ArtixInstallDir/artix_version/cxx_java/samples/advanced/

These samples are described in "Locator Samples for the C++/Java JNI Runtime" on page 19.

Java JAX-WS locator samples

The Java JAX-WS runtime introduced in Artix 5.0 includes one locator-related sample in the following directory:

    ArtixInstallDir/artix_version/java/samples/locator

This sample is described in "Locator Sample for the Java JAX-WS Runtime" on page 20.

Locator Samples for the C++/Java JNI Runtime

locator sample

The primary locator sample illustrates how the locator can isolate consumers from knowledge about changes in a service's physical location. Most examples in this manual are simplified versions of this locator sample. This sample shows how you can locator-enable an application simply using configuration. The consumer and server code are not aware that the locator is being used for discovery of endpoints.

This sample is implemented in both C++ and Java JNI.

locator_query sample

The locator_query sample illustrates more advanced uses of the locator lookup that the consumer-side plug-in does not implement. It illustrates how to use the `listEndpoints` operation to obtain a list of the services registered with a locator. The sample goes on to illustrate how you can filter the returned list of services with various query selection elements, using the `query_endpoints` operation.

This sample is implemented in both C++ and Java JNI.

located_router sample

The located_router sample illustrates how endpoints that are wrapped by an Artix router can still use the locator service for dynamic discovery of endpoint information. In this sample, the endpoints that the router creates are automatically registered with the locator when the router starts up.

This sample is implemented only in C++, but the illustrated functionality is available to Java JNI users as well.

locator_load_balancing sample

The locator_load_balancing sample 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 sample is implemented in both C++ and Java JNI.

high_availability_locator sample

The high_availability_locator sample illustrates how to run the Artix locator in a replicated and highly-available mode.

This sample is implemented only in C++, but the illustrated functionality is available to Java JNI users as well.

Locator Sample for the Java JAX-WS Runtime

locator discovery sample

This sample illustrates the basic reference discovery mechanism supported by the Artix locator service when used with the Java JAX-WS runtime.

The server is configured to register an endpoint reference with the locator when the corresponding JAX-WS endpoint is published. This registration occurs transparently to the server-side application code. The configured `registerOnPublish` feature that triggers this registration is also used to enable liveness monitoring via a heartbeat conversation with the peer manager instance embedded in the locator.

Migrating from Previous Versions

Overview

The Artix 4.x locator service supports queries from unmodified Artix 3.x consumer 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 consumers.

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.x locator service functionality is a combination of two distinct services combined in a single plug-in. The plug-in enables an Artix 3-compatible service that supports the locator WSDL from Artix 3.0. It also enables a more advanced Artix 4.x locator service. As shipped, both services are active, but you can disable the Artix 3-compatible service if your network does not have Artix 3 service endpoints or consumers.

As illustrated in Figure 1, Artix 3.0 consumers can query an Artix 4.x locator and get the expected results.

Figure 1: *Artix 4 locator backward compatibility*
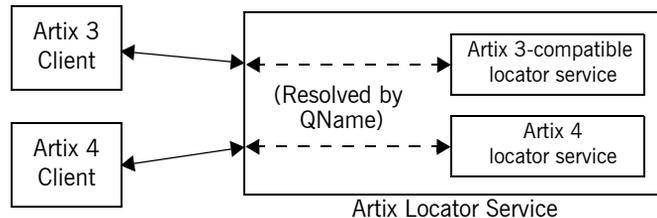
Locator service QNames

The QName for the Artix 4.x 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.x 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.x locator by querying the container with the `it_container_admin` command. For example:

1. Go to the Artix 4.x locator demonstration in *ArtixInstallDir*/ `artix_version`/cxx_java/samples/advanced/locator.

2. Load the Artix C++/Java JNI environment by invoking the `artix_env[.bat]` command.

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 locator service shipped with Artix 4.x, and the locator shipped with the C++/Java JNI runtime of Artix 5.x, supports the following combinations of services and consumers:

- Artix 4 services, Artix 4 consumers
- Artix 4 services, Artix 3 consumers
- Artix 3 services, Artix 4 consumers
- Artix 3 services, Artix 3 consumers

The terms used here have the following meanings:

Artix 4 service	Uses the Artix 4 or Artix 5/C++ `locator_endpoint` plug-in.
Artix 4 consumer	Uses the Artix 4 or Artix 5/C++ `locator_client` plug-in, or is coded to create a proxy to the locator using the new locator QName.
Artix 3 service	Uses the Artix 3 `locator_endpoint` plug-in.
Artix 3 consumer	Uses no plug-in; is coded to create a proxy to the locator using the old locator QName.

Unsupported configurations

Neither Artix 4.x services nor Artix 4.x consumers (as defined above) work in any combination with a locator service that uses the `locator_service` plug-in shipped with Artix 3.

Migration strategies

The Artix 4.0 locator service is backward compatible by default. There are no configuration steps required to enable backward compatibility in the locator service itself.

Locator services for Artix 4.1, 4.2, and the C++/Java JNI runtime of Artix 5.x require a one-line addition to their Artix configuration files, as described in "Artix 4.1/4.2/5.0 locator setup for backward compatibility".

You can start your Artix 4.x services in a way that supports both Artix 4 and Artix 3 consumers. See .

You can migrate your consumers one at a time to Artix 4.x locator compatibility, using the steps described in .

Artix 4.1/4.2/5.0 locator setup for backward compatibility

The `artix.cfg` file shipped with Artix 4.1 and 4.2, and with the C++/JAX-RPC runtime of Artix 5.x, all have a configuration entry, `bus:non_compliant_epr_format`. The shipped `artix.cfg` sets this entry by default to "`false`". This setting allows for greater interoperability between Artix and Web services software from other vendors.

If your site uses a locator service, locator-enabled services, and locator-enabled consumers all built with Artix 4.1, 4.2, or the C++/JAX-RPC runtime of Artix 5.x, then no further configuration is necessary.

If your site uses a locator service built with Artix 4.1, 4.2, or 5.0/C++ with services and consumers from Artix 4.0 and 3.0.x, then you must add one configuration entry in your Artix configuration. Add the line to the `locator.servce` scope of the configuration file that controls your instance of the locator service. The line to add is:

```
bus:non_compliant_epr_format = "true";
```

Note: The locator-related demos that ship with Artix 4.1, 4.2, and 5.0/C++ do not have this line added to their `locator.cfg` files.

For example, the following example shows an edited `locator.cfg` file for the primary locator demo that allows Artix 3.x and 4.0 consumers to connect to and use the Artix 4.1/4.2/5.0-C++ locator service:

```
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"];
      bus:non_compliant_epr_format = "true";
    };
  };
};
```

Disabling locator support for Artix 3

When you have all Artix consumers migrated to Artix 4.x or 5.x/C++, the backward compatibility feature of the Artix 4.x/5.x-C++ locator is no longer necessary for your site. However, there is no need to disable the backward compatibility feature, and the Artix 4.x/5.x-C++ locator performance is not improved by disabling backward compatibility.

If you prefer to disable this feature anyway, you can use a local configuration scope to override the Artix root configuration. In your local scope, set the WSDL path equal to an empty string for the Artix 3-compatible version of the locator service, using a line like the following:

```
bus:qname_alias:locator_oldversion = "";
```

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 Service

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, C++ Runtime.

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, C++ Runtime. See the chapters "Artix Configuration" and "Accessing Contracts and References."

Configuring the locator to run in a container

To configure the locator to run in an 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*/cxx_java/samples/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 boldface 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 plug-in). The service_locator plug-in implements the locator service functionality.

The locator service uses SOAP over HTTP, so the soap and at_http plug-ins are loaded automatically when the process parses the locator's WSDL contract.

Dynamic port used by default

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 shown in Example 2:

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 consumers. 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 because you need to make sure that all consumers and services on your network can access your locator service. Contacting the locator may be difficult if it starts on a different port every time.

Configuring a fixed port

There are several ways to deploy the locator on a well-known fixed port:

- You can edit the default locator.wsdl contract (this is not recommended)
- You can create a copy of locator.wsdl contract for your application and deploy it in a separate configuration scope.
- 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. By default, this contract 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 boldface 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 = "/myartix/etc/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 locator service must be started before any applications that need to use 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 /myartix/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 /myartix/etc/ContainerService.url
   -publishwsdl
   -service {http://ws.iona.com/2005/11/locator}LocatorService
   -file /myartix/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 `/myartix/etc` subdirectory.

3. Finally, you must make sure your consumers use the activated WSDL file, now resident in the specified directory, when each consumer 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 /myartix/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 consumer to 9000, or you can publish the WSDL from the container using it_container_admin with the -publishwsdl option, shown in .

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
   /myartix/etc -container /myartix/etc/ContainerService.url
   -shutdown
```

or if you deploy the locator and container on a fixed port:

```
it_container_admin -ORBdomain_name locator -host artixserver
   -port 9000 -shutdown
```

Registering Services with the Locator

Overview

A service does not need to have its implementation changed to work with the Artix locator. The only requirements are that the service is configured to load the correct plug-ins, and to reference the correct locator contract.

If you require more fine-grained control, you can control the service endpoints that are registered. You may want to do this if you have some services that you do not want to be visible to consumers.

Whichever Artix runtime you use, you locator-enable services by means of configuration, not coding. The configuration steps are different for services using the C++ runtime and for services using the Java runtime:

Configuring a Locator-Enabled Service, C++ Runtime

Configuring a locator-enabled service

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

Another example is shown in Example 1 on page 29, where a service in the scope `demo.locator.service` is configured to load the `locator_endpoint` plug-in.

> **Note:** Service developed using the JAX-RPC APIs also run on top of the C++ runtime using a JNI layer.

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_service/
    locator.wsdl";
```

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

For more information, see the Artix Configuration Reference, C++ Runtime.

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, C++ Runtime. The details of each variable are described in the Artix Configuration Reference, C++ Runtime.

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.example.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.example.com/finance}*"];
```

Service registration

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

You can register multiple instances of the same service with a locator. These service instances must be running in different applications (buses). When the locator receives multiple registrations of the same service implemented in different server applications, the locator generates a pool of references for the service type. When consumers make a request for a service, the locator supplies references from this pool using a load-balancing algorithm. For more information on load balancing see "Using Load Balancing" on page 39.

Configuring a Locator-Enabled Service, Java Runtime

Overview

This section describes the steps to configure your services for the Java runtime to automatically register with the Artix locator.

Spring-based configuration

Artix services written for the Java runtime use XML-based configuration files based on the Spring framework. Spring-based configuration for the Java runtime is described in Artix Configuration Reference, Java Runtime.

The following example service configuration file shows the bean element that allows the service to register with and use the Artix locator.

Example 4: *Service configuration for the Java JAX-WS runtime*

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:locatorEndpoint="http://cxf.iona.com/locator/endpoint"
       xsi:schemaLocation="
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/
   spring-beans.xsd">

 <!-- Config for Locator runtime support -->
    <bean id="LocatorSupport" class="com.iona.cxf.locator.LocatorSupport">
        <property name="bus" ref="cxf"/>
        <property name="contract">
            <value>./wsdl/locator.wsdl</value>
        </property>
    </bean>


<!-- Config for JAX-WS Endpoint published by the server -->
    <jaxws:endpoint name="{http://apache.org/hello_world_soap_http}SoapPort"
                    createdFromAPI="true">
        <jaxws:features>
          <locatorEndpoint:registerOnPublish monitorLiveness="true"
                                        heartbeatInterval="10001"/>
        </jaxws:features>
    </jaxws:endpoint>

</beans>
```

Using Load Balancing

Overview

The Artix locator provides a lightweight mechanism for balancing workloads among a group of services. When several services with the same service name register with the Artix locator, it automatically creates a list of references to each instance of this service. The locator hands out references to consumers using a round-robin or random algorithm. This process is automatic and invisible to both consumers 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:

- Start multiple services using the same WSDL contract but without hard coding the addressing for that service. For example, if the service uses HTTP, use a location such as `location=http://servicehost:0/servicename`. If the service uses CORBA, use the address `location="IOR:"`.

- Create a number of copies of the WSDL contract defining the service, and change the addressing information so that each copy has a unique address. Then bring up each service instance using a different copy of the contract.

> **Note:** The locator determines whether a service is part of a group by using the name specified in the `service` element of the service's contract. If you are using the Artix locator to load balance, each instance of your service should be associated with the same binding and logical interface. Otherwise, consumers might end up using a different binding, transport, or portType, depending on the endpoint reference obtained from the locator service.

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 service instances.

As consumers 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 consumer a reference to a dead endpoint.

However, when an endpoint fails unexpectedly, it cannot notify the locator, and the locator can unknowingly give a consumer an invalid reference. This might cause the failure to cascade onto one or more consumers if consumers try to invoke on a dead endpoint.

To decrease the risk of passing invalid references to consumers, the locator service monitors the health of the endpoints that have been registered with it. If it determines that an endpoint is no longer available, it removes that endpoint from its database. The locator service can determine the availability of its registered endpoints because it expects those endpoints to send periodic ping messages to the locator service. If these messages stop arriving, the locator service determines that the endpoint is dead.

You can adjust the interval between locator service pings by setting the `plugins:locator:peer_timeout` configuration variable. The default and minimum setting is 10,000 milliseconds (10 seconds). For further information on this configuration variable, see the Artix Configuration Reference, C++ Runtime.

Locator service failure

If the locator service itself fails, and it is not running in high availability mode, all the references to the registered endpoints are lost, and the active endpoints are no longer registered with the locator. The endpoints detect when the locator service fails, because they are expecting periodic messages from the locator using the peer manager service. Once an endpoint determines that the locator has failed, it attempts to reconnect to the locator

and reregister its endpoints. This behavior lets you stop and restart a deployed locator service without interruption to the consumers and services on the network.

You can adjust the interval with which the locator pings the endpoints by setting the `plugins:locator:peer_timeout` configuration variable. The default and minimum setting is 10,000 milliseconds (10 seconds). For further information on this configuration variable, see the Artix Configuration Reference, C++ Runtime.

Highly available locator cluster

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

The setup and configuration of a high availability locator cluster is discussed in the "Deploying High Availability" chapter of Configuring and Deploying Artix Solutions, C++ Runtime. See especially the "Configuring Locator High Availability" section of this chapter.

Starting Services with Artix 3 Consumer Support

Overview

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

Migrating Artix 4 services

There are no required changes to your application code between Artix 3 and Artix 4 for locator-aware Artix services. You can migrate your service and application code from Artix 3 to Artix 4 with these steps:

- Regenerate stub code generated from WSDL using the Artix 4 code generators.
- Recompile and link your service application.
- Make sure the Artix 4 version of the `locator_endpoint` plug-in is loaded at runtime and is configured correctly.
- To allow Artix 3.x and 4.0 consumers to connect to an Artix 4.1/4.2 locator service, add one configuration line to the locator service configuration scope, as described in "Artix 4.1/4.2/5.0 locator setup for backward compatibility" on page 23.

Specify the right QName

As described in "Locator service QNames" on page 21, the Artix 4.x locator implements both Artix 4 and Artix 3-compatible locator services. If you access an Artix 4.x locator using the QName of the Artix 3 locator, then the Artix 4.x locator responds as an Artix 3 locator.

Supporting Artix 3 consumers

To support Artix 3 consumers from your Artix 4.x services, you must:

- Run your locator service using the Artix 4.x version of the `locator_service` plug-in.
- Make sure the Artix 3-compatible WSDL published from the Artix 4.x locator is accessible to your Artix 3 consumers 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
• By retrieving the activated WSDL from the locator and storing it in a location accessible to consumers

Artix 3 interoperability if the locator is 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.

Consumers 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 consumers will look for the locator on port 9000. The Artix 3 consumers 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 consumers use a fixed port for the locator service, then Artix 3 consumers should run without any changes against the Artix 4 locator service running on the same port.

Artix 3 interoperability if you used activated WSDL

The locator demo named `locator` has a script that starts a service. This script queries the locator's container for the locator's WSDL contract, and then writes that activated WSDL to a file. The consumer startup script then reads the activated WSDL from the same file.

You do not need to write the activated locator WSDL to a file at the same time the service starts up, as is done in the `locator` demo. This could occur in a separate script, and only needs to done once.

If your applications uses the technique of writing activated WSDL to a file, then you must modify the script that writes the WSDL. In your modified script, 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 consumers expect to find it.

For example, consumers 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. For example purposes only, let's say that:

   ♦ Artix 4 is installed in C:\IONA

- ◆ Artix 3 is installed in C:\IONA3

2. In the `bin` directory of the Artix 4 version of the demo, copy `run_cxx_server.bat` to a new file. Let's call it `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. Open a command prompt window and run the test batch files in the following sequence:

    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 a second command prompt window 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 you must 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.

Example 5: *Example 4-3_interop.bat file*

```
@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
```

Adding SOAP 1.2 Support

Overview

The default `locator.wsdl` file shipped with Artix contains a SOAP 1.1 binding and a SOAP 1.1 service. Starting with release 4.1, Artix supports SOAP 1.2 bindings as well.

If your site requires the use of SOAP 1.2 bindings for communication with the locator service, follow these steps:

- Make a copy of the default `locator.wsdl` file.
- Edit your copy to include a SOAP 1.2 binding. See the SOAP 1.2 chapter of Writing Artix Contracts for guidelines on adding a SOAP 1.2 binding.
- Use the `bus:initial_contract:url` configuration variable to point to the location of your edited `locator.wsdl` file, or use one of several WSDL publishing methods described in "Accessing WSDL Contracts" in Configuring and Deploying Artix Solutions, C++ Runtime.

SOAP 1.2 considerations

The SOAP 1.2 binding in Artix 4.1, 4.2, and 5.0-C++ supports endpoint references (EPRs) only in the format defined by the WS-Addressing standard, and no longer supports the deprecated proprietary Artix references. Artix's SOAP 1.1 binding supports both EPRs and the Artix references used by Artix 3.0 and earlier.

This means that an Artix 4.1/4.2/5.0-C++ locator that uses the SOAP 1.2 binding cannot support connections from Artix 4.0 and 3.0 consumers, because those Artix versions did not support SOAP 1.2. Thus, when defining your Artix 4.1/4.2/5.0-C++ locator service, if your site intends to maintain backward compatibility with Artix 4.0 and Artix 3.0 consumers, do not also use a SOAP 1.2 binding. The configuration step described in "Artix 4.1/4.2/5.0 locator setup for backward compatibility" on page 23 is not compatible with a SOAP 1.2 binding.

Using the Locator from an Artix Consumer

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

In this chapter

This chapter discusses the following topics:

Configuring Artix Consumers to Use the Locator Service

Overview

Before a consumer can use the Artix locator service, it must be configured to load the required plug-ins. The plug-ins provide native access to the locator service and eliminates the need for creating a proxy to obtain service references.

The consumers written using the C++ APIs and consumers written using the JAX-RPC APIs run on top of the C++ runtime and require one set of configuration. Consumers written using the JAX-WS APIs run on top of the Java runtime and require a different set of configuration. However, both runtimes provide the same set of features to the consumer.

In this section

This section discusses the following topics:

Configuring C++ and JAX-RPC Consumers

Overview

This section describes how to configure consumers to use the locator_client plug-in, and describes the features of this plug-in. This section applies to all consumers written for C++ or Java JNI.

Note: JAX-RPC consumers are deployed into the C++ runtime using a JNI layer.

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, C++ Runtime. See the chapters "Artix Configuration" and "Accessing Contracts and References."

Configuring a consumer

To use a deployed locator service, configure consumers to load the locator_client plug-in.

An example is shown in Example 1 on page 29, where consumers in the scope demo.locator.client are configured to load the locator_client plug-in. The relevant portion of that example is shown here:

```
demo
{
  locator
  {
    client
    {
        orb_plugins = ["xmlfile_log_stream", "locator_client"];
    };
  };
};
```

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

Note: The locator_client plug-in is only supported for interacting with a locator service that uses the Artix 4 version of the locator_service plug-in. It does *not* query a locator service that uses the Artix 3 version of the locator_service plug-in.

Consumer plug-in features

The `locator_client` plug-in is responsible for helping consumers to resolve their target service endpoints using the locator service, without having any code that explicitly does so. With the plug-in configured to be used, when the consumer's code attempts to resolve its target service's endpoint, the plug-in connects to the locator service to obtain a reference to the target service. This interaction is triggered by the call to resolve the initial reference to the target service. However, it is the plug-in that implements the actions initiated by that call.

In order to function, the `locator_client` plug-in requires addressing information for the locator service. This can be specified using various techniques outlined in the "Accessing Contracts and References" chapter of Configuring and Deploying Artix Solutions, C++ Runtime. For example, you can pass in the location of the WSDL through the command line, or you can configure the location in the consumer's configuration domain.

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

Configuring JAX-WS Consumers

Overview

This section describes the steps to configure JAX-WS consumers to automatically use the Artix locator to find the Artix service of interest.

Spring-based configuration

JAX-WS consumers use XML-based configuration files based on the Spring framework. Spring-based configuration for the Java runtime is described in Artix Configuration Reference, Java Runtime.

The following example consumer configuration file shows the `bean` element that allows a consumer to use the Artix locator to find the Artix service of interest.

Example 6: *Consumer configuration for the Java JAX-WS runtime*

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/
   spring-beans.xsd">

    <!-- Config for Locator runtime support -->
    <bean id="LocatorSupport" class="com.iona.cxf.locator.LocatorSupport">
        <property name="bus" ref="cxf"/>
        <property name="contract">
            <value>./wsdl/locator.wsdl</value>
        </property>
    </bean>
</beans>
```

Obtaining Service References from the Locator Service

Overview

Once a consumer is configured to load the locator service plug-ins, it requires some additional coding to use the locator service to obtain references to the services in which it is interested. Each of the programming models supported by Artix has a slightly different way of enabling a consumer's use of the locator service. However, all of the models make it extremely simple.

In this section

This section discusses the following topics:

Implementing a C++ Consumer

Overview

This section shows how to write consumer code in C++ that uses an Artix locator service to locate and connect to a target service of interest.

C++ consumer code

The steps each locator consumer must take are:

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

2. 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 consumer code had to directly interact with the locator service using the WSDL contract defining the locator. These steps are no longer necessary in Artix 4 and 5 locator consumers.

C++ Example

The locator consumer in Example 7 is a small, complete application designed to work in the context of the locator demonstration in *ArtixInstallDir*/artix/*version*/cxx_java/samples/advanced/locator.

See "Explanation of Example 7" on page 57 for notes on this example.

Example 7: *Locator consumer example in C++*

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

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

Example 7: *Locator consumer example in C++*

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

    // Initialize the Artix bus.
    IT_Bus::Bus_var bus;
    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;
    }
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.";

5       SimpleServiceClient simple_client(ep_ref);
```

Example 7:   *Locator consumer 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 7

The following points refer to the number labels in Example 7.

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
    consumer 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 29.

2. This line constructs a QName for the target service to which this
    consumer 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*/cxx_java/
    samples/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*/cxx_java/samples/advanced/ locator/cxx/client. This class is derived from `IT_Bus::ClientProxyBase()`, which is the base class for all Artix C++ proxies. In this case, the proxy is set up for the target service defined in the QName set up as described in paragraph **2** above.

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 7   The code in Example 7 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*/cxx_java/ samples/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 JAX-RPC Consumer

Overview

This section shows how to write a JAX-RPC consumer that uses an Artix locator service to locate and connect to a target service of interest.

JAX-RPC consumer code

Consumer's wishing to use the Artix locator service can simply use the `Bus.createClient` shown in Example 8.

Example 8: *createClient() Method*

```
abstract  Remote createClient(QName serviceName,
                              Class interfaceClass)
```

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. Using the returned reference the `Bus` creates a proxy. It also handles failover if the service instance originally contacted dies and other service instances are available.

JAX-RPC Example

Example 9 shows code for getting a proxy using the `createClient()` method. The example assumes that you have properly configured the consumer to load the `locator_client` plug-in.

See "Explanation of Example 9" on page 60 for notes on this example.

Example 9: *Locator consumer example with JAX-RPC*

```
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 9: *Locator consumer example with JAX-RPC*

```
1     Bus bus = Bus.init(args);

2      QName name = new QName("http://www.iona.com/FixedBinding",
                             "SOAPHTTPService");
3      client = (SimpleService)bus.createClient(name,
                                          SimpleService.class);
       if (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 " +
                          name.getLocalPart() +".");

4      result = client.say_hello(greeting);
       System.out.println("The say_hello operation returned: \n"
                          + "      " + result);
    }
  }
```

Explanation of Example 9

The following points refer to the number labels in Example 9.

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 consumer 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 29.

2. This line constructs a QName for the target service to which this consumer will connect at runtime. The components of the QName are defined in the target service's WSDL contract.

3. The `createClient()` method uses the supplied QName to perform a look up using the locator service and obtain a reference to an instance of the desired service. It uses the EPR returned from the locator to create a proxy to the service instance.

4. 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`.

Implementing a JAX-WS Consumer

Overview

This section shows how to write a JAX-WS consumer that uses an Artix locator service to locate and connect to a target service of interest.

JAX-WS Code

The steps each JAX-WS locator consumer must take are:

1. Mint a locator mediated EPR using the `EndpointReferenceUtils.mint()` method.

    Note: When the consumer is not configured to use the locator service, the `EndpointReferenceUtils.mint()` method requires that you have provided an implementation of the `EndpointResolver` interface and properly registered it.

    The mint() method takes the QName of the service in which the consumer is interested and the consumer's `Bus` instance.

2. Get the `org.apache.cxf.jaxws.ServiceImple` instance of the service class using the `org.apache.cxf.jaxws.support.ServiceDelegateAccessor.get()` method.

    This method takes an instance of the JAX-WS 2.0 `Service` object your consumer would normally use and returns an Artix specific implementation of the object. The Artix specific `ServiceImpl` object provides additional methods including a JAX-WS 2.1 style `getPort()` method that allows you to create proxies from an EPR.

3. Use the returned Artix specific `ServiceImpl` object's `getPort()` method to create a proxy for the desired service.

    The `getPort()` method takes the EPR minted from the locator and the `Class` object for the SEI of the desired service. It returns a proxy for the desired service.

JAX-WS example

Example 10 shows code for using the locator service from a JAX-WS consumer. You can find a sample application in *ArtixInstallDir*/artix/ *version*/java/samples/locator/discovery.

Example 10: *JAX-WS Consumer Using the Artix Locator*

```
import org.apache.cxf.jaxws.ServiceImpl;
import org.apache.cxf.jaxws.support.ServiceDelegateAccessor;
import org.apache.cxf.ws.addressing.EndpointReferenceType;
import org.apache.cxf.wsdl.EndpointReferenceUtils;

...

public final class Client
{
  private static final QName SERVICE_NAME
        = new QName("http://apache.org/hello_world_soap_http",
   "SOAPService");

  ...

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

    EndpointReferenceType epr =
      EndpointReferenceUtils.mint(SERVICE_NAME, bus);

    ServiceImpl serviceImpl =
      ServiceDelegateAccessor.get(new SOAPService());

    Greeter greeter = serviceImpl.getPort(epr, Greeter.class);

  ...
  }

  ...
}
```

The markers in the code:
1 `Bus bus = BusFactory.getDefaultBus();`
2 `EndpointReferenceType epr = EndpointReferenceUtils.mint(SERVICE_NAME, bus);`
3 `ServiceImpl serviceImpl = ServiceDelegateAccessor.get(new SOAPService());`
4 `Greeter greeter = serviceImpl.getPort(epr, Greeter.class);`

The code in Example 10 does the following:

1. Gets the consumer's bus instance.
2. Mints an EPR for the service whose QName is stored in SERVICE_NAME.

3. Gets the Artix specific `ServiceImpl` object for the JAX-WS 2.0 `Service` object that would normally be used to get a proxy for the remote service.

4. Gets a proxy for the remote service using the Artix specific `ServiceImpl` and the EPR minted from the locator.

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`.

> **Note:** JAX-WS consumers do not have native access to the locator service's query functionality. You can access this functionality by generating the proper stub code using the JAX-WS code generators.

Demonstration code

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

Filtered and unfiltered lists of services

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

1. Obtain a reference to the locator service and create a client proxy to the locator.

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.

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 11:

Example 11: *Contents of locator-query.xsd*

```xml
<?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 68.)
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 multiple 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 Consumer Code

Overview

The following differences between Artix 4.x/5.0-C++ and Artix 3.0 might affect any existing Artix consumers:

- 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 consumers 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. These tasks were formerly the responsibility of consumer code.

For WS-Addressing migration information, see the chapters "Endpoint References" in Developing Artix Applications in C++, or "Using Endpoint References" in Developing Artix Applications with JAX_RPC.

Old and new locator WSDL contracts supported

As described in "Backward compatibility" on page 21, the Artix 4 locator service incorporates both Artix 4 and Artix 3-compatible locator services.

Artix 4 includes a newer version of `locator.wsdl`. The new `locator.wsdl` file is located by default in the following directory of your Artix installation:

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

In a production environment, the `locator.wsdl` can be in any location.

Artix 4 also includes a copy of the Artix 3.x `locator.wsdl` file. By default, this file is installed in:

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

The Artix 4/5-C++ 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 consumer 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 4/5-C++ 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/5.x-C++ version of `locator.wsdl`.

Configuration for Artix 4.1/4.2 locator service

To allow Artix 3.x and 4.0 consumers to connect to an Artix 4.1, 4.2, or 5.0-C++ locator service, add one new configuration entry to the locator service configuration scope, as described in "Artix 4.1/4.2/5.0 locator setup for backward compatibility" on page 23.

Locator WSDL operation names

The names of public operations in the Artix 4 and 5-C++ version of `locator.wsdl` have been changed, as described in Table 1.

Table 1: *Operation names in Artix 3 and Artix 4/5 locator.wsdl*

Artix 3 locator.wsdl	Artix 4/5-C++ locator.wsdl	Notes
lookup_endpoint	lookupEndpoint	The Artix 3 version returns an Artix Reference. The Artix 4/5-C++ version returns a WS-Addressing type reference.
list_endpoints	listEndpoints	

Table 1: *Operation names in Artix 3 and Artix 4/5 locator.wsdl*

Artix 3 locator.wsdl	Artix 4/5-C++ locator.wsdl	Notes
	queryEndpoints	There is no Artix 3 equivalent operation.

Migrating consumer code to Artix 4

As described in "Migrating from Previous Versions" on page 21, the Artix 4/5-C++ locator supports the use of unmodified Artix 3 consumers. This allows you to put your first migration efforts into upgrading your locators and services to Artix 4 or 5-C++. Once those tasks are complete, you can migrate your consumers 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 consumers. See "Configuring C++ and JAX-RPC Consumers" on page 51.

2. If your code directly invokes any operations of the locator.wsdl contract, update the operation names as described in "Locator WSDL operation names" on page 73.

3. For consumers in C++, simplify your consumer code as described below.

   In Artix 3, the coding steps that every locator consumer 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 most of the work, the coding steps are shortened to the following[2]:

    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 consumers in Java for the C++/JAX-RPC runtime, change your consumer 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.

---

2. 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 65.

Using the Locator from a Non-Artix Client

This chapter describes how to use an Artix locator service from consumers generated by other SOA systems.

In this chapter

This chapter discusses the following topics:

Implementing a .NET Client

Overview

.NET clients can use the locator to discover services, using the `Bus.Services.dll` library. The locator provides a number of methods for looking up services and managing service registration. The Artix .Net plug-in is Web Services Enhancements 2.0 (WSE 2.0) compliant. The helper classes included in the `Bus.Services` library simplify working with the locator by providing native .Net calls to access the locator and the endpoint references it produces.

What you need before starting

Before starting to develop a client that uses the Artix locator you need:

- A means for contacting a deployed Artix locator. This can be one of the following:
  - An endpoint reference
  - An HTTP address
  - A local copy of the locator's contract
- A locally accessible copy of the WSDL contract that defines the service that you want the client to invoke upon.
- To install WSE 2.0 SP3 before starting an Artix .NET locator client.

Demonstration code

The code examples in this section are taken from the locator demo's .NET client code. The .NET client makes a request on a service instance for which it does not have a current endpoint reference. The .NET client accesses the locator to get a reference to an active instance of the service on which it wants to make requests. The complete client code can be found in the following directory of your Artix installation:

```
InstallDir\artix\Version\cxx_java\samples\advanced\locator\dotne
    t\client
```

To develop a .Net client that uses the Artix locator, do the following:

1. Create a new project in Visual Studio.

2. Right-click the folder for your new project and select **Add Reference** from the pop-up menu.

3. Click **Browse** on **Add Reference** window.

4. In the file selection window, browse to your Artix installation and select the `Bus.Services.dll` from the `InstallDir\artix\Version\utils\.NET` directory.

5. Click **OK** to return to the Visual Studio editing area.

6. Right-click the folder for your new project and select **Add Web Reference** from the pop-up menu.

7. In the **Address:** field of the browser, enter the full path name of the contract for the service on which you are going to make requests.

8. Click **Add Reference** to return to the Visual Studio editing area.

9. Open the `.cs` file generated for the contract you imported.

10. Locate the class declaration for the service on which you intend to make requests. The class declaration looks similar to that shown in Example 12.

Example 12: *.Net Service Proxy Class Declaration*

```
public class SOAPService :
   System.Web.Services.Protocols.SoapHttpClientProtocol {
```

11. Add a new C# class to your project.

12. Add the statement `using Bus.Services;` after the statement `using System;`.

13. Create a service proxy for the Artix locator by instantiating an instance of the `Bus.Services.Locator` class as shown in Example 13.

Example 13: *Instantiating a Locator Proxy in .NET*

```
Locator l = new Locator("http://localhost:8080");
```

The constructor's parameter is the HTTP address of a deployed locator. The `Locator` class also has two constructors that take an Artix reference or a WSDL contract for use with the Artix locator.

14. Create a QName representing the name of the service you wish to locate using an instance of the `System.Xml.XmlQualifiedName` class as shown in Example 14.

Example 14: *Creating a .NET QName*

```
XmlQualifiedName service = new XmlQualifiedName(
  "HelloWorldService",
  "http://www.iona.com/hello_world_soap_http"
);
```

15. Invoke the `lookup_endpoint()` method on the locator proxy as shown in Example 15.

Example 15: *Looking-up an Endpoint Reference.*

```
Reference ref = l.lookupEndpoint(service);
```

`lookupEndpoint()` takes the QName of the desired service as a parameter and returns an endpoint reference if an instance of the specified service is registered with the locator instance. Endpoint references are implemented in the .NET `Bus.Services.EndpointReferenceType` class.

16. Create a .NET proxy for the service on which you are going to make requests as you normally would.

17. Change the value of the proxy's `.Url` member to the SOAP address of the endpoint reference returned from the locator as shown in Example 16.

Example 16: *Changing the URL of a .Net Service Proxy to Use an Endpoint Reference*

```
simpleService.Url = endpoint.Address.Value;
```

18. Make requests on the service as you would normally.

Implementing an Axis Client

Overview

Because the Artix locator is a SOAP over HTTP service whose interface is defined by a WSDL contract, an Axis client can use it to locate deployed instances of a service. Using the Artix locator from an Axis client involves generating a proxy for the locator service and interpreting the returned endpoint reference.

Axis version

The examples in this chapter were developed using Axis 1.3. They should work with Axis 1.4 as well.

Procedure

To develop an Axis client that uses Artix locator, do the following:

1.   Generate a WSDL file from a running locator instance.

2.   Generate Axis stub code from the generated locator WSDL file as shown in Example 17:

Example 17:   *Generating Axis Stub Code for Locator*

```
Java  org.apache.axis.wsdl.WSDL2Java locator.wsdl
```

3.   Generate Axis stub code from the WSDL document for the service on which you want your client to invoke, as shown in Example 18:

Example 18:   *Generating Axis Stub Code for the Target Web Service*

```
Java  org.apache.axis.wsdl.WSDL2Java simple_service.wsdl
```

Note: Axis only understands services that use SOAP over HTTP. If you are starting from the WSDL for an Artix service that supports other bindings and transports, you must make a copy of its WSDL document and remove references to any namespaces, bindings, and transports other than SOAP over HTTP. Then run the Axis WSDL2Java generator on your simplified copy of the WSDL document.

4. Retrieve a locator service endpoint as shown in Example 19:

Example 19: *Retrieving a Locator Service Endpoint*

```
String tns = "http://ws.iona.com/2005/11/locator";
QName service = new QName(tns, "LocatorService");
String port = "LocatorServicePort";
java.lang.String url = get_soap_address("locator.wsdl",
    service, port);
java.net.URL endpoint = new java.net.URL(url);
```

5. Instantiate a locator proxy as shown in Example 20:

Example 20: *Instantiating a Locator Proxy*

```
LocatorService_Service lssl = new
    LocatorService_ServiceLocator();
LocatorServiceBindingStub locProxy =
    (LocatorServiceBindingStub)lssl.getLocatorServicePort(end
    point);
```

6. Get a reference to a service using the locator proxy as shown in Example 21.

Example 21: *Getting an Endpoint Reference*

```
QName servName = new QName("http://www.iona.com/
    FixedBinding", "SOAPHTTPService");
EndpointReferenceType serviceEpr =
    locProxy.lookupEndpoint(servName);
```

7. Get the address of the service from the returned endpoint reference as shown in Example 22.

Example 22: *Getting the Service Address*

```
String servURL =
    serviceEpr.getAddress().get_value().toString();
serv_endpoint = new java.net.URL(servURL);
```

8. Create a proxy for the service and invoke on it as you normally would.

Index