Artix™

Managing Artix Solutions with JMX

Version 4.1, September 2006

Making Software Work Together™

Updated: September 22, 2006

Contents

List of Figures

List of Tables

Preface

What is Covered in this Book

Managing Artix Solutions with JMX explains how to monitor and manage Artix services in a runtime environment using Java Management Extensions.

This book does not discuss the specifics of the different middleware and messaging products that Artix interacts with. It is assumed that you have a working knowledge of the specific middleware products and transports you are using.

Who Should Read this Book

The main audience of *Managing Artix Solutions with JMX* is Artix system administrators. However, anyone involved in designing a large scale Artix solution will find this book useful.

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

How to Use this Book

This book includes the following:

- Chapter 1 introduces the Artix JMX architecture and describes the Artix components that can be managed using JMX.
- Chapter 2 explains how to configure an Artix runtime for JMX.
- Chapter 3 explains how to manage and monitor Artix services using JMX consoles.

The Artix Library

The Artix documentation library is organized in the following sections:

- Getting Started
- Designing Artix Solutions
- Configuring and Managing Artix Solutions
- Using Artix Services
- Integrating Artix Solutions
- Integrating with Management Systems
- Reference
- Artix Orchestration

Getting Started

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

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

Designing Artix Solutions

The books in this section go into greater depth about using Artix to solve real-world problems. They describe how to build service-oriented architectures with Artix and how Artix uses WSDL to define services:

- Building Service-Oriented Infrastructures with Artix provides an overview of service-oriented architectures and describes how they can be implemented using Artix.
- Writing Artix Contracts describes the components of an Artix contract. Special attention is paid to the WSDL extensions used to define Artix-specific payload formats and transports.

Developing Artix Solutions

The books in this section how to use the Artix APIs to build new services:

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

Configuring and Managing Artix Solutions

This section includes:

- Configuring and Deploying Artix Solutions explains how to set up your Artix environment and how to configure and deploy Artix services.
- Managing Artix Solutions with JMX explains how to monitor and manage an Artix runtime using Java Management Extensions.

Using Artix Services

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

- Artix Router Guide explains how to integrate services using the Artix router.
- Artix Locator Guide explains how clients can find services using the Artix locator.
- Artix Session Manager Guide explains how to manage client sessions using the Artix session manager.
- Artix Transactions Guide, C++ explains how to enable Artix C++ applications to participate in transacted operations.
- Artix Transactions Guide, Java explains how to enable Artix Java applications to participate in transacted operations.
- Artix Security Guide explains how to use the security features in Artix.

Integrating Artix Solutions

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

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

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

Integrating with Management Systems

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

- IBM Tivoli Integration Guide explains how to integrate Artix with the IBM Tivoli enterprise management system.
- BMC Patrol Integration Guide explains how to integrate Artix with the BMC Patrol enterprise management system.
- CA-WSDM Integration Guide explains how to integrate Artix with the CA-WSDM SOA management system.
- AmberPoint Integration Guide explains how to integrate Artix with the AmberPoint SOA management system.

Reference

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

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

Artix Orchestration

These books describe the Artix support for Business Process Execution Language (BPEL), which is available as an add-on to Artix. These books include:

- Artix Orchestration Release Notes
- Artix Orchestration Installation Guide
- Understanding Artix Orchestration
- Artix Orchestration Administration Console Help.

Getting the Latest Version

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

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

Searching the Artix Library

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

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

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

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

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

Artix Online Help

Artix Designer and Artix Orchestration Designer include comprehensive online help, providing:

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

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

- Select **Help**|**Help Contents** from the menu bar. The help appears in the contents panel of the Eclipse help browser.
- Press **F1** for context-sensitive help.

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

Artix Glossary

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

Additional Resources

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

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

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

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

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

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

Bold	Bold words in normal text represent graphical user interface components such as menu commands and dialog boxes. For example: the **User Preferences** dialog.

Keying Conventions

This book uses the following keying conventions:

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

CHAPTER 1

Monitoring and Managing an Artix Runtime with JMX

This chapter explains how to monitor and manage an Artix runtime using Java Management Extensions (JMX).

In this chapter

This chapter discusses the following topics:

Introduction

Overview

You can use Java Management Extensions (JMX) to monitor and manage key Artix runtime components both locally and remotely. For example, using any JMX-compliant client, you can perform the following tasks:

- View bus status.
- Stop or start a service.
- Change bus logging levels dynamically.
- Monitor service performance details.
- View the interceptors for a selected port.

How it works

Artix has been instrumented to allow runtime components to be exposed as JMX Managed Beans (MBeans). This enables an Artix runtime to be monitored and managed either in process or remotely with the help of the JMX Remote API.

Artix runtime components can be exposed as JMX MBeans, out-of-the-box, for both Java and C++ Artix servers. All leading vendor application servers and containers can be managed using JMX. However, what is unique about the Artix instrumentation is that its core runtime can also be managed. This contrasts with the JVM 1.5 management capabilities where you can observe garbage collection and thread activities using JMX.

In addition, support for registering custom MBeans is also available in Artix since version 3.0. Java developers can create their own MBeans and register them either with their MBeanServer of choice, or with a default MBeanServer created by Artix (see "Relationship between runtime and custom MBeans" on page 20).

Figure 1 shows an overview of how the various components interact. The Java custom MBeans are optional components.

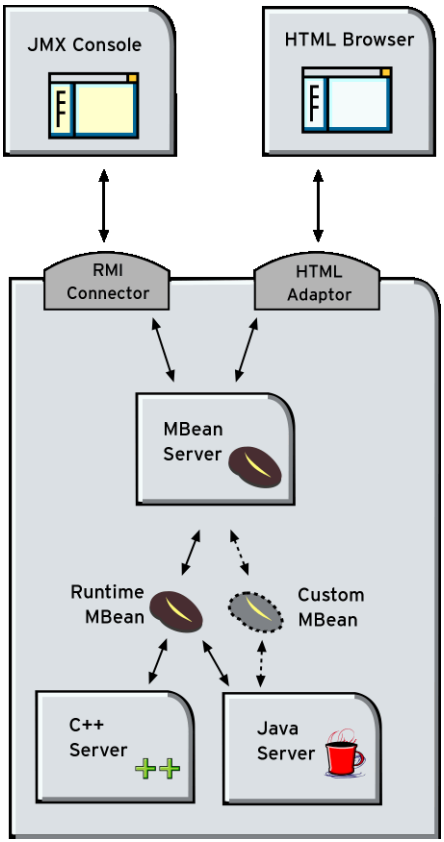

Figure 1: *Artix JMX Architecture*

What can be managed

Both Java and C++ Artix servers can have their runtime components exposed as JMX MBeans. The following components can be managed:

- Bus
- Service
- Port

All runtime components are registered with an MBeanServer as Open Dynamic MBeans. This ensures that they can be viewed by third-party management consoles without any additional client-side support libraries.

All MBeans for Artix runtime components conform with Sun's JMX Best Practices document on how to name MBeans (see http://java.sun.com/products/JavaManagement/best-practices.html). Artix runtime MBeans use `com.iona.instrumentation` as their domain name when creating ObjectNames.

Note: An MBeanServerConnection, which is an interface implemented by the MBeanServer is used in the examples in this chapter. This ensures that the examples are correct for both local and remote access.

See also "Further information" on page 22 for details of how to access MBean Server hosting runtime MBeans either locally and remotely.

Relationship between runtime and custom MBeans

The Artix runtime instrumentation provides an out-of-the-box JMX view of C++ and Java services. Java developers can also create custom JMX MBeans to manage Artix Java components such as services.

You may choose to write custom Java MBeans to manage a service because the Artix runtime is not aware of the current service's application semantics. For example, the Artix runtime can check service status and update performance counters, while a custom MBean can provide details on the status of a business loan request processing.

It is recommended that custom MBeans are created to manage application-specific aspects of a given service. Ideally, such MBeans should not duplicate what the runtime is doing already (for example, calculating service performance counters).

It is also recommended that custom MBeans use the same naming convention as Artix runtime MBeans. Specifically, runtime MBeans are named so that containment relationships can be easily established. For example:

```
// Bus :
com.iona.instrumentation:type=Bus,name=demos.jmx_runtime

Service :
com.iona.instrumentation:type=Bus.Service,name="{http://ws.iona.
   com}SOAPService",Bus=demos.jmx_runtime

// Port :
com.iona.instrumentation:type=Bus.Service.Port,name=SoapPort,Bus
   .Service="{http://ws.iona.com}SOAPService",Bus=demos.jmx_runt
   ime
```

Using these names, you can infer the relationships between ports, services and buses, and display or process a complete tree in the correct order. For example, assuming that you write a custom MBean for a loan approval Java service, you could name this MBean as follows:

```
com.iona.instrumentation:type=Bus.Service.LoanApprovalManager,na
   me=LoanApprovalManager,Bus.Service="{http://ws.iona.com}SOAPS
   ervice",Bus=demos.jmx_runtime
```

For details on how to write custom MBeans, see Developing Artix Applications in Java.

Accessing the MBeanServer programmatically

Artix runtime support for JMX is enabled using configuration settings only. You do not need to write any additional Artix code. When configured, you can use any third party console that supports JMX Remote to monitor and manage Artix servers.

If you wish to write your own JMX client application, this is also supported. To access Artix runtime MBeans in a JMX client, you must first get a handle to the MBeanServer. The following code extract shows how to access the MBeanServer locally:

```
Bus bus = Bus.init(args);
MBeanServer mbeanServer =
    (MBeanServer)bus.getRegistry().getEntry(ManagementConstants.M
    BEAN_SERVER_INTERFACE_NAME);
```

The following shows how to access the MBeanServer remotely:

```
// The address of the connector server
String url = "service:jmx:rmi://host:1099/jndi/artix";
JMXServiceURL address = new JMXServiceURL(url);

// Create the JMXConnectorServer
JMXConnector cntor = JMXConnectorFactory.connect(address, null);

// Obtain a "stub" for the remote MBeanServer
MBeanServerConnection mbsc = cntor.getMBeanServerConnection();
```

Please see the `advanced/management/jmx_runtime` demo for a complete example on how to access, monitor and manage Artix runtime MBeans remotely.

Further information

For further information, see the following URLs:

JMX

http://java.sun.com/products/JavaManagement/index.jsp

JMX Remote

http://www.jcp.org/aboutJava/communityprocess/final/jsr160/

Open Dynamic MBeans

http://java.sun.com/j2se/1.5.0/docs/api/javax/management/openmbean/package-summary.html

ObjectName

http://java.sun.com/j2se/1.5.0/docs/api/javax/management/ObjectName.html

MBeanServerConnection

http://java.sun.com/j2se/1.5.0/docs/api/javax/management/MBeanServerConnection.html

MBeanServer

http://java.sun.com/j2se/1.5.0/docs/api/javax/management/MBeanServer.html

Managed Bus Components

Overview

This section describes the attributes and methods that you can use to manage JMX MBeans representing Artix bus components. For example, you can use any JMX client to perform the following tasks:

- View bus attributes.
- Enable monitoring of bus services.
- Dynamically change logging levels for known subsystems.

If you wish to write your own JMX client, this section describes methods that you can use to access Artix logging levels and subsystems, and provides a JMX code example.

Bus MBean registration

When an Artix bus is initialized, a corresponding JMX MBean is created and registered for that bus with an MBeanServer.

Java

For example, in an Artix Java application, this occurs after the following call:

```
String[] args = ...;
Bus serverBus = Bus.init(args);
```

C++

For example, in an Artix C++ application, this occurs after the following call:

```
Bus_var server_bus = Bus.init(argc, argv);
```

When a bus is shutdown, a corresponding MBean is unregistered from the MBeanServer.

Bus naming convention

An Artix bus `ObjectName` uses the following convention:

```
com.iona.instrumentation:type=Bus,name=busIdentifier
```

Bus attributes

The following bus component attributes can be managed by any JMX client:

Table 1: *Managed Bus Attributes*

Name	Description	Type	Read/Write
`scope`	Bus scope used to initialize a bus.	`String`	No
`identifier`	Bus identifier, typically the same as its scope.	`String`	No
`arguments`	Bus arguments, including the executable name.	`String[]`	No
`servicesMonitoring`	Used to enable/disable services performance monitoring.	`Boolean`	Yes
`services`	A list of object names representing services on this bus.	`ObjectName[]`	No

`servicesMonitoring` is a global attribute which applies to all services and can be used to change a performance monitoring status.

Note: By default, service performance monitoring is enabled when JMX management is enabled in a standalone server, and disabled in an `it_container` process.

When using a JMX console to manage a `it_container` server, you can enable performance monitoring by setting the `serviceMonitoring` attribute to `true`.

`services` is a list of object names that can be used by JMX clients to build a tree of components. Given this list, you can find all other registered service MBeans that belong to this bus.

For examples of bus attributes displayed in a JMX console, see "Using JMX Consoles with Artix" on page 45.

Bus methods

If you wish to write your own JMX client, you can use the following bus methods to access logging levels and subsystems:

Table 2: *Managed Bus Methods*

Name	Description	Parameters	Return Type
`getLoggingLevel`	Returns a logging level for a subsystem.	`subsystem` (`String`)	`String`
`setLoggingLevel`	Sets a logging level for a subsystem.	`subsystem` (`String`), `level` (`String`)	`Boolean`
`setLoggingLevelPropagate`	Sets a logging level for a subsystem with propagation.	`subsystem` (`String`), `level` (`String`), `propagate` (`Boolean`)	`Boolean`

All the attributes and methods described in this section can be determined by introspecting `MBeanInfo` for the `Bus` component (see http://java.sun.com/j2se/1.5.0/docs/api/javax/management/MBeanInfo.html).

Example JMX client

The following code extract from an example JMX client application shows how to access bus attributes and logging levels:

```
MBeanServerConnection mbsc = ...;
String busScope = ...;
ObjectName busName = new ObjectName("com.iona.instrumentation:type=Bus,name=" + busScope);

if (mbsc.isRegistered(busName)) {
    throw new MBeanException("Bus mbean is not registered");
}

// MBeanInfo can be used to check for all known attributes and methods
MBeanInfo info = mbsc.getMBeanInfo(busName);

// bus scope
String scope = (String)mbsc.getAttribute(busName, "scope");
// bus identifier
String identifier = (String)mbsc.getAttribute(busName, "identifier");
// bus arguments
String[] busArgs = (String[])mbsc.getAttribute(busName, "arguments");
```

```
// check servicesMonitoring attribute, then disable and reenable it
Boolean status = (Boolean)mbsc.getAttribute(busName, "servicesMonitoring");
if (!status.equals(Boolean.TRUE)) {
     throw new MBeanException("Service monitoring should be enabled by default");
}

mbsc.setAttribute(busName, new Attribute("servicesMonitoring", Boolean.FALSE));
status = (Boolean)mbsc.getAttribute(busName, "servicesMonitoring");
if (!status.equals(Boolean.FALSE)) {
    throw new MBeanException("Service monitoring should be disabled now");
}

mbsc.setAttribute(busName, new Attribute("servicesMonitoring", Boolean.TRUE));
status = (Boolean)mbsc.getAttribute(busName, "servicesMonitoring");
if (!status.equals(Boolean.TRUE)) {
    throw new MBeanException("Service monitoring should be reenabled now");
}

// list of service MBeans
ObjectName[] serviceNames = (ObjectName[])mbsc.getAttribute(busName, "services");

// logging
String level = (String)mbsc.invoke(
                              busName,
                              "getLoggingLevel",
                              new Object[] {"IT_BUS"},
                              new String[] {"subsystem"});
if (!level.equals("LOG_ERROR")) {
    throw new MBeanException("Wrong IT_BUS logging level");
}

level = (String)mbsc.invoke(
                         busName,
                         "getLoggingLevel",
                         new Object[] {"IT_BUS.INITIAL_REFERENCE"},
                         new String[] {"subsystem"});
if (!level.equals("LOG_ERROR")) {
    throw new MBeanException("Wrong IT_BUS.INITIAL_REFERENCE logging level");
}
level = (String)mbsc.invoke(
                         busName,
                         "getLoggingLevel",
                         new Object[] {"IT_BUS.CORE"},
                         new String[] {"subsystem"});
if (!level.equals("LOG_INFO_LOW")) {
    throw new MBeanException("Wrong IT_BUS.CORE logging level");
}
```

```
Boolean result = (Boolean)mbsc.invoke(
                          busName,
                          "setLoggingLevel",
                          new Object[] {"IT_BUS", "LOG_WARN"},
                          new String[] {"subsystem", "level"});

level = (String)mbsc.invoke(
                          busName,
                          "getLoggingLevel",
                          new Object[] {"IT_BUS"},
                          new String[] {"subsystem"});
if (!level.equals("LOG_WARN")) {
    throw new MBeanException("IT_BUS logging level has not been set properly");
}

level = (String)mbsc.invoke(
                         busName,
                         "getLoggingLevel",
                         new Object[] {"IT_BUS.INITIAL_REFERENCE"},
                         new String[] {"subsystem"});
if (!level.equals("LOG_WARN")) {
    throw new MBeanException("IT_BUS.INITIAL_REFERENCE logging level has not been set
   properly");
}

level = (String)mbsc.invoke(
                         busName,
                         "getLoggingLevel",
                         new Object[] {"IT_BUS.CORE"},
                         new String[] {"subsystem"});
if (!level.equals("LOG_INFO_LOW")) {
    throw new MBeanException("IT_BUS.CORE logging level should not be changed");
}

// propagate
result =  (Boolean)mbsc.invoke(
                             busName,
                             "setLoggingLevelPropagate",
                             new Object[] {"IT_BUS", "LOG_SILENT", Boolean.TRUE},
                             new String[] {"subsystem", "level", "propagate"});

level = (String)mbsc.invoke(
                         busName,
                         "getLoggingLevel",
                         new Object[] {"IT_BUS"},
                         new String[] {"subsystem"});
```

```
if (!level.equals("LOG_SILENT")) {
    throw new MBeanException("IT_BUS logging level has not been set properly");
}

level = (String)mbsc.invoke(
                        busName,
                        "getLoggingLevel",
                        new Object[] {"IT_BUS.INITIAL_REFERENCE"},
                        new String[] {"subsystem"});
if (!level.equals("LOG_SILENT")) {
    throw new Exception("IT_BUS.INITIAL_REFERENCE logging level has not been set
   properly");
}
level = (String)mbsc.invoke(
                        busName,
                        "getLoggingLevel",
                        new Object[] {"IT_BUS.CORE"},
                        new String[] {"subsystem"});
if (!level.equals("LOG_SILENT")) {
    throw new MBeanException("IT_BUS.CORE logging level shouldve been set to LOG_SILENT");
}
```

Further information

For information on Artix logging levels and subsystems, see Configuring and Deploying Artix Solutions.

Managed Service Components

Overview

This section describes the attributes and methods that you can use to manage JMX MBeans representing Artix service components. For example, you can use any JMX client to perform the following tasks:

- View managed services.
- Dynamically change a service status.
- Monitor service performance data.
- Manage service ports.

The Artix locator and session manager services have also been instrumented. These provide an additional set of attributes on top of those common to all services.

If you wish to write your own JMX client, this section describes methods that you can use and provides a JMX code example.

Service MBean registration

When an Artix servant is registered for a service, a JMX Service MBean is created and registered with an MBeanServer.

Java

For example, in an Artix Java application, this occurs after the following call:

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

QName bankServiceName = new
   QName("http://www.iona.com/bus/tests", "BankService");
Servant servant = new SingleInstanceServant(new BankImpl(),
   serviceWsdlURL, bus);

bus.registerServant(servant, bankServiceName, "BankPort");
```

C++

For example, in an Artix C++ application, this happens after the following call:

```
Bus_var server_bus = Bus.init(argc, argv);

BankServiceImpl servant;
bus->register_servant(
    servant,
    wsdl_location,
    QName("http://www.iona.com/bus/tests", "BankService")
);
```

When a service is removed, a corresponding MBean is unregistered from the MBeanServer.

Service naming convention

An Artix service `ObjectName` uses the following convention:

```
com.iona.instrumentation:type=Bus.Service,name="{namespace}local
   name",Bus=busIdentifier
```

In this format, a `name` has an expanded service QName as its value. This value includes double quotes to permit for characters that otherwise would not be allowed.

Service attributes

The following service component attributes can be managed by any JMX client:

Table 3: *Managed Service Attributes*

Name	Description	Type	Read/Write
`name`	Service QName in expanded form.	`String`	No
`state`	Service state.	`String`	No
`serviceCounters`	Service performance data.	`CompositeData`	No
`ports`	A list of ObjectNames representing ports for this service.	`ObjectName[]`	No

`name` is an expanded QName, such as `{http://www.iona.com/bus/tests}BankService`.

`state` represents a current service state that can be manipulated by stop and start methods.

`ports` is a list of ObjectNames that can be used by JMX clients to build a tree of components. Given this list, you can find all other registered Port MBeans which happen to belong to this Service.

serviceCounters attributes

The following service performance attributes can be retrieved from the `serviceCounters` attribute:

Table 4: *serviceCounters Attributes*

Name	Description	Type
`averageResponseTime`	Average response time in milliseconds.	`Float`
`requestsOneway`	Total number of oneway requests to this service.	`Long`
`requestsSinceLastCheck`	Number of requests happened since last check.	`Long`
`requestsTotal`	Total number of requests (including oneway) to this service.	`Long`
`timeSinceLastCheck`	Number of seconds elapsed since last check.	`Long`
`totalErrors`	Total number of request-processing errors.	`Long`

For examples of service attributes displayed in a JMX console, see "Using JMX Consoles with Artix" on page 45

Service methods

If you wish to write your own JMX client, you can use the following service methods to manage a specific service:

Table 5: *Managed Service Attributes*

Name	Description	Parameters	Return Type
`name`	Start (activate) a service.	None	`Void`
`state`	Stop (deactivate) a service.	None	`Void`

All the attributes and methods described in this section can be accessed by introspecting `MBeanInfo` for the Service component.

Example JMX client

The following code extract from an example JMX client application shows how to access service attributes and methods:

```
MBeanServerConnection mbsc = ...;

String busScope = ...;
ObjectName serviceName = new ObjectName("com.iona.instrumentation:type=Bus.Service" +
                    ",name=\"{http://www.iona.com/hello_world_soap_http}SOAPService\""
   +",Bus=" + busScope);

if (!mbsc.isRegistered(serviceName)) {
    throw new MBeanException("Service MBean should be registered");
}

// MBeanInfo can be used to check for all known attributes and methods
MBeanInfo info = mbsc.getMBeanInfo(serviceName);

// service name
String name = (String)mbsc.getAttribute(serviceName, "name");

// check service state attribute then reset it by invoking stop and start methods

String state = (String)mbsc.getAttribute(serviceName, "state");
if (!state.equals("ACTIVATED")) {
    throw new MBeanException("Service should be activated");
}

mbsc.invoke(serviceName, "stop", null, null);
```

```
state = (String)mbsc.getAttribute(serviceName, "state");
if (!state.equals("DEACTIVATED")) {
    throw new MBeanException("Service should be deactivated now");
}

mbsc.invoke(serviceName, "start", null, null);

state = (String)mbsc.getAttribute(serviceName, "state");
if (!state.equals("ACTIVATED")) {
    throw new MBeanException("Service should be activated again");
}

// check service counters

CompositeData counters = (CompositeData)mbsc.getAttribute(serviceName, "serviceCounters");
Long requestsTotal = (Long)counters.get("requestsTotal");
Long requestsOneway = (Long)counters.get("requestsOneway");
Long totalErrors = (Long)counters.get("totalErrors");
Float averageResponseTime = (Float)counters.get("averageResponseTime");
Long requestsSinceLastCheck = (Long)counters.get("requestsSinceLastCheck");
Long timeSinceLastCheck = (Long)counters.get("timeSinceLastCheck");

// ports
ObjectName[] portNames = (ObjectName[])mbsc.getAttribute(serviceName, "ports");
```

Further information

MBeanInfo

http://java.sun.com/j2se/1.5.0/docs/api/javax/management/MBeanInfo.html

CompositeData

http://java.sun.com/j2se/1.5.0/docs/api/javax/management/openmbean/CompositeData.html

Artix Locator Service

Overview

The Artix locator can also be exposed as a JMX MBean. A locator managed component is a service managed component that can be managed like any other bus service with the same set of attributes and methods. The Artix locator also exposes it own specifc set of attributes.

Locator attributes

An Artix locator MBean exposes the following locator-specific attributes:

Table 6: *Locator MBean Attributes*

Name	Description	Type
`registeredEndpoints`	Number of registered endpoints.	`Integer`
`registeredServices`	Number of registered services, less or equal to number of endpoints.	`Integer`
`serviceLookups`	Number of service lookup requests.	`Integer`
`serviceLookupErrors`	Number of service lookup failures.	`Integer`
`registeredNodeErrors`	Number of node (peer ping) failures.	`Integer`

Example JMX client

The following code extract from an example JMX client application shows how to access locator attributes and methods:

```
MBeanServerConnection mbsc = ...;
String busScope = ...;
ObjectName serviceName = new ObjectName("com.iona.instrumentation:type=Bus.Service" +
                    ",name=\"{http://ws.iona.com/2005/11/locator}LocatorService\""
   +",Bus=" + busScope);

// use common attributes and methods, see an example above

// Locator specific attributes
Integer regServices = (Integer)mbsc.getAttribute(serviceName, "registeredServices");
Integer endpoints = (Integer)mbsc.getAttribute(serviceName, "registeredEndpoints");
Integer nodeErrors = (Integer)mbsc.getAttribute(servicetName, "registeredNodeErrors");
Integer lookupErrors = (Integer)mbsc.getAttribute(serviceName, "serviceLookupErrors");
Integer lookups = (Integer)mbsc.getAttribute(serviceName, "serviceLookups");
```

Artix Session Manager Service

Overview

The Artix session manager can also be exposed as a JMX MBean. A session manager component is a service managed component that can be managed like any other bus service with the same set of attributes and methods. The Artix session manager also exposes it own specifc set of attributes.

Session manager attributes

An Artix session manager MBean exposes the following session manager-specific attributes:

Table 7: *Session Manager MBean Attributes*

Name	Description	Type
`registeredEndpoints`	Number of registered endpoints.	`Integer`
`registeredServices`	Number of registered services, less or equal to number of endpoints.	`Integer`
`serviceGroups`	Number of service groups.	`Integer`
`serviceSessions`	Number of service sessions	`Integer`

Example JMX client

The following code extract from an example JMX client application shows how to access session manager attributes and methods:

```
MBeanServerConnection mbsc = ...;
String busScope = ...;
ObjectName serviceName = new ObjectName("com.iona.instrumentation:type=Bus.Service" +

   ",name=\"{http://ws.iona.com/sessionmanager}SessionManagerService\"" +",Bus=" +
   busScope);
// use common attributes and methods, see an example above

// SessionManager specific attributes
Integer regServices = (Integer)mbsc.getAttribute(serviceName, "registeredServices");
Integer endpoints = (Integer)mbsc.getAttribute(serviceName, "registeredEndpoints");
Integer serviceGroups = (Integer)mbsc.getAttribute(serviceName, "serviceGroups");
Integer serviceSessions = (Integer)mbsc.getAttribute(serviceName, "serviceSessions");
```

Managed Port Components

Overview

This section describes the attributes that you can use to manage JMX MBeans representing Artix port components. For example, you can use any JMX client to perform the following tasks:

- Monitor managed ports.
- View message and request interceptors.

If you wish to write your own JMX client, this section also shows an example of accessing these attributes in JMX code.

Port MBean registration

Port managed components are typically created as part of a service servant registration. When service is activated, all supported ports will also be registered as MBeans.

When a service is removed, a corresponding Service MBean, as well as all its child Port MBeans are unregistered from the MBeanServer.

Naming convention

An Artix port `ObjectName` uses the following convention:

```
com.iona.instrumentation:type=Bus.Service.Port,name=portName,Bus
    .Service="{namespace}localname",Bus=busIdentifier
```

Port attributes

The following bus component attributes can be managed by any JMX client:

Table 8: *Supported Service Attributes*

Name	Description	Type	Read/Write
`name`	Port name.	`String`	No
`address`	Transport specific address representing an endpoint.	`String`	No
`interceptors`	List of interceptors for this port.	`String[]`	No

Table 8: *Supported Service Attributes*

Name	Description	Type	Read/Write
`transport`	An optional attribute representing a transport for this port.	`ObjectName[]`	No

interceptors

The `interceptors` attribute is a list of interceptors for a given port. Internally, `interceptors` is an instance of `TabularData` that can be considered an array/table of `CompositeData`. However, due to a current limitation of `CompositeData`, (no insertion order is maintained, which makes it impossible to show interceptors in the correct order), the interceptors are currently returned as a list of strings, where each `String` has the following format:

```
[name]: name [type]: type [level]: level [description]: optional
   description
```

In this format, *`type`* can be `CPP` or `Java`; *`level`* can be `Message` or `Request`.

It is most likely that this limitation will be fixed in a future JDK release, probably JDK 1.7 because the enhancement request has been accepted by Sun. In the meantime, interceptors details can be retrieved by parsing a returned `String` array.

For examples of port attributes displayed in a JMX console, see "Using JMX Consoles with Artix" on page 45

Example JMX client

The following code extract from an example JMX client application shows how to access port attributes and methods:

```
MBeanServerConnection mbsc = ...;

String busScope = ...;
ObjectName portName = new ObjectName("com.iona.instrumentation:type=Bus.Service.Port" +
                     ",name=SoapPort" +

   ",Bus.Service=\"{http://www.iona.com/hello_world_soap_http}SOAPService\"" +",Bus=" +
   busScope);

if (!mbsc.isRegistered(portName)) {
    throw new MBeanException("Port MBean should be registered");
}

// MBeanInfo can be used to check for all known attributes and methods
MBeanInfo info = mbsc.getMBeanInfo(portName);

// port name
String name = (String)mbsc.getAttribute(portName, "name");

// port address
String address = (String)mbsc.getAttribute(portName, "address");

// check interceptors

String[] interceptors = (String[])mbsc.getAttribute(portName, "interceptors");
if (interceptors.length != 6) {
    throw new MBeanException("Number of port interceptors is wrong");
}

handleInterceptor(interceptors[0],
                  "MessageSnoop",
                  "Message",
                  "CPP");
handleInterceptor(interceptors[1],
                  "MessagingPort",
                  "Request",
                  "CPP");
handleInterceptor(interceptors[2],
                  "http://schemas.xmlsoap.org/wsdl/soap/binding",
                  "Request",
                  "CPP");
```

```
handleInterceptor(interceptors[3],
                  "TestInterceptor",
                  "Request",
                  "Java");
handleInterceptor(interceptors[4],
                  "bus_response_monitor_interceptor",
                  "Request",
                  "CPP");
handleInterceptor(interceptors[5],
                  "ServantInterceptor",
                  "Request",
                  "CPP");
```

For example, the `handleInterceptor()` function may be defined as follows:

```
private void handleInterceptor(String interceptor,
                               String name,
                               String level,
                               String type) throws Exception {
  if (interceptor.indexOf("[name]: " + name) == -1 ||
      interceptor.indexOf("[type]: " + type) == -1 ||
      interceptor.indexOf("[level]: " + level) == -1) {

      throw new MBeanException("Wrong interceptor details");
  }
  // analyze this interceptor further
}
```

CHAPTER 2

Configuring JMX in an Artix Runtime

This chapter explains how to configure an Artix runtime to be managed with Java Management Extensions (JMX).

In this chapter

This chapter discusses the following topic:

Artix JMX Configuration

Overview

This section explains the Artix configuration variable settings that you must configure to enable JMX monitoring of the Artix runtime, and access for remote JMX clients.

Enabling the management plugin

To expose the Artix runtime using JMX MBeans, you must enable a `bus_management` plug-in as follows:

```
jmx_local
{
   plugins:bus_management:enabled="true";
};
```

This setting enables local access to JMX runtime MBeans. The `bus_management` plug-in wraps runtime components into Open Dynamic MBeans and registers them with a local MBeanServer.

Configuring remote JMX clients

To enable remote JMX clients to access runtime MBeans, use the following configuration settings:

```
jmx_remote
{
   plugins:bus_management:enabled="true";
   plugins:bus_management:connector:enabled="true";
};
```

These settings allow for both local and remote access.

Specifying a remote access URL

Remote access is performed through JMX Remote, using an RMI Connector on a default port of `1099`. Using this configuration, you can use the following JNDI-based JMXServiceURL to connect remotely:

```
service:jmx:rmi:///jndi/rmi://host:1099/artix
```

Configuring a remote access port

To specify a different port for remote access, use the following configuration variable:

```
plugins:bus_management:connector:port="2000";
```

You can then use the following JMXServiceURL:

```
service:jmx:rmi:///jndi/rmi://host:2000/artix
```

Configuring a stub-based JMXServiceURL

You can also configure the connector to use a stub-based JMXServiceURL as follows:

```
jmx_remote_stub
{
   plugins:bus_management:enabled="true";
   plugins:bus_management:connector:enabled="true";
   plugins:bus_management:connector:registry:required="false";
};
```

See the javax.management.remote.rmi package for more details on remote JMX.

Publishing the JMXServiceURL to a local file

You can also request that the connector publishes its JMXServiceURL to a local file:

```
plugins:bus_management:connector:url:publish="true";
```

The following entry can be used to override the default file name:

```
plugins:bus_management:connector:url:file="../../service.url";
```

Further information

For further information, see the following:

RMI Connector

http://java.sun.com/j2se/1.5.0/docs/api/javax/management/remote/rmi/RMIConnector.html

JMXServiceURL

http://java.sun.com/j2se/1.5.0/docs/api/javax/management/remote/JMXServiceURL.html

http://java.sun.com/j2se/1.5.0/docs/api/javax/management/remote/rmi/package-summary.html

CHAPTER 3

Using JMX Consoles with Artix

You can use third-party management consoles that support JMX Remote to monitor and manage Artix servers (for example, JConsole and MC4J). You can view the status of a bus instance, stop or start a service, change bus logging levels, or view interceptor chains. For convenience, Artix installs the MC4J management console, which you can run out-of-the-box with the JMX demo.

In this chapter

This chapter discusses the following topics:

Managing Artix Services with MC4J

Overview

You can use the open source MC4J management console to view service attributes and operations, stop or start a service, view interceptor chains, and change bus logging levels dynamically.

Artix installs MC4J into the *InstallDir*\artix\4.1\mc4j directory. This section uses the jmx_runtime Artix demo to show a detailed walk-through example of how to use MC4J to monitor and manage an Artix server.

Starting the MC4J console

To start the MC4J management console, perform the following steps:

1. Change directory to *InstallDir*\artix\4.1\bin.
2. Run the following command:

Windows
```
> start_mc4j.bat
```

UNIX
```
% ./start_mc4j
```

Running the JMX demo

Before creating a new server connection in the MC4J console, perform the following steps:

1. Change to the demo directory:

   ```
   cd InstallDir\artix\4.1\demos\advanced\management\jmx_runtime
   ```

2. Build the C++ or Java demo:

C++
```
nmake
```

Java
```
ant
```

3. Run the C++ or Java server:

C++
```
run_cxx_server.bat
```

Java
```
run_java_server.bat
```

Creating a new server connection

To create a new server connection in the MC4J console, perform the following steps:

1. Select **MC4J Connections,** and right click, as shown in Figure 2.

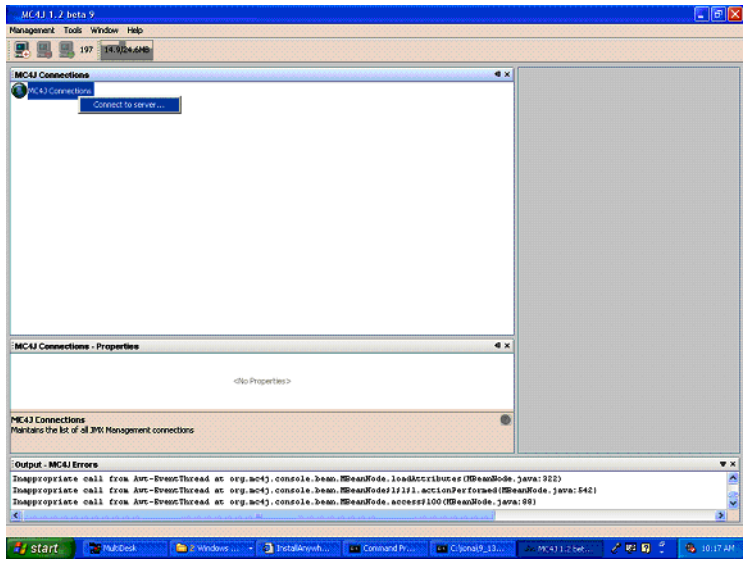

Figure 2: *Connecting to a Server*

2. Click **Connection server...** to launch the **My wizard** dialog, as shown in Figure 3.

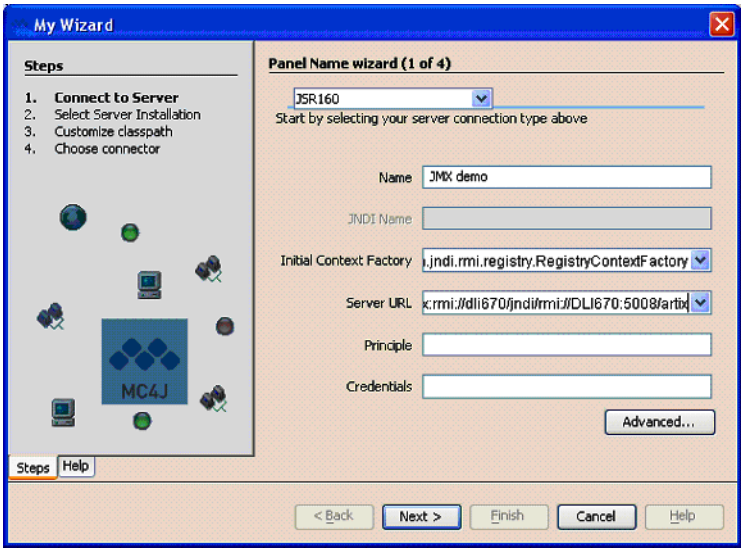

Figure 3: *Server Connection Details*

3. In the **My Wizard** dialog, select `JSR160` as your server connection type.
4. Enter `JMX demo` as your connection **Name**.
5. Enter the contents of the following file as the **Server URL**:
   `demos/advanced/management/jmx_runtime/etc/connector.url`

6. Click **Next** to go to next screen, as shown in Figure 4.

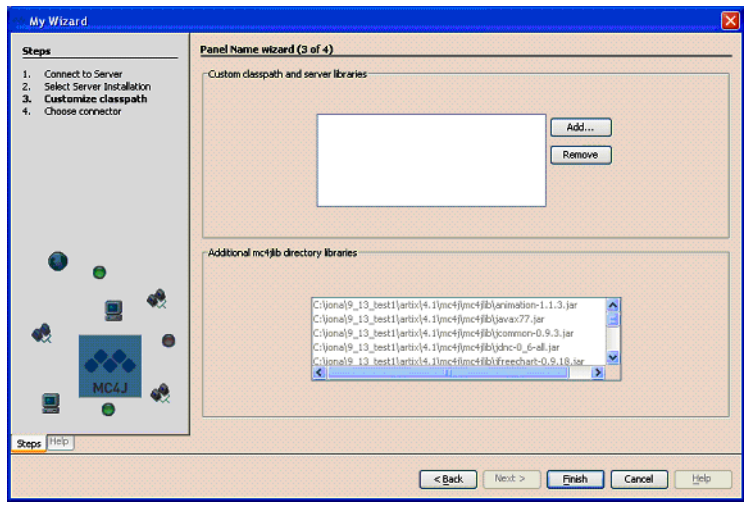

Figure 4: *Creation of Server Connection*

7. Click **Finish** to finish the creation of a new server connection.

8. In the left panel of the MC4J console, a new server connection named `JMX demo` is created, as shown in Figure 5:

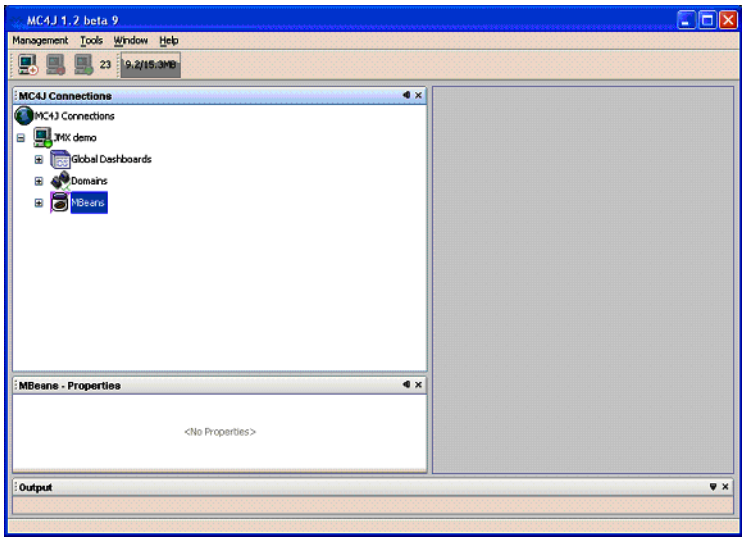

Figure 5: *New Server Connection*

Monitoring and managing a service

To monitor and manage an example service in the Mc4J console, perform the following steps:

1. Expand the **MBeans** tree node in the left panel of MC4J.
2. Double click on the following tree node, as shown in Figure 6:

   ```
   Name='{http://www.iona.com/jmx_runtime}SOAPService',type=Bus.
   Service
   ```

This displays the attributes and operations of the `SOAPService` in the service properties dialog.

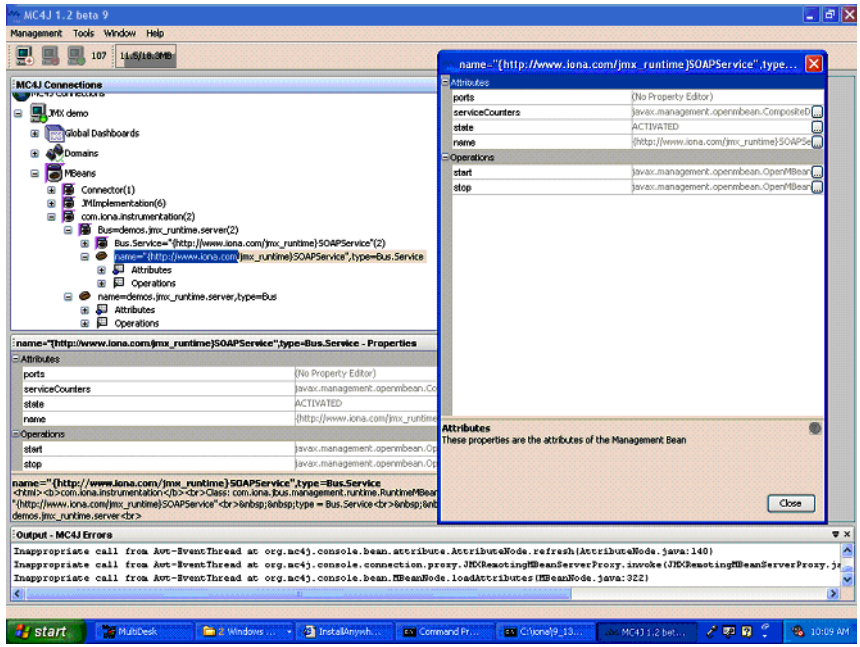

Figure 6: *Viewing Service Properties*

3. Click the **...** button at the right of the `serviceCounters` attribute in the service properties dialog. This displays the details for the `serviceCounters` attribute, as shown in Figure 7.

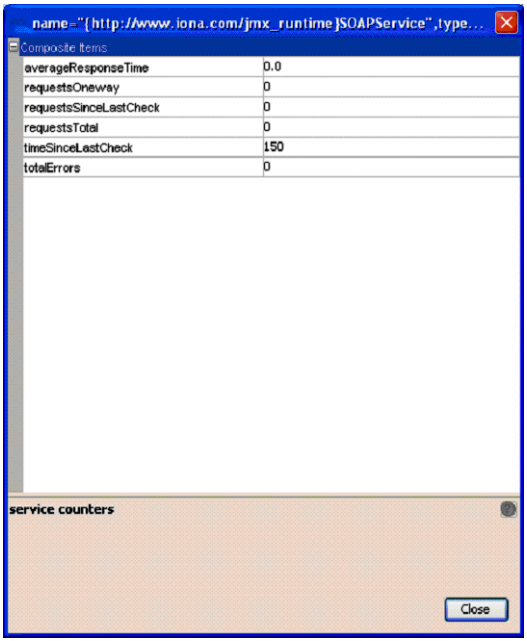

Figure 7: *Viewing Service Counters Properties*

4. Click the **...** button at right of the **stop** operation on the service properties dialog. This displays a dialog for the **stop** operation, as shown in Figure 8.

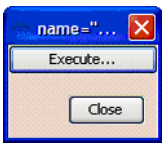

Figure 8: *Stopping a Service*

5. Click **Execute...** to stop the service. In the `SOAPservice` properties dialog, the state attribute of the service becomes `DEACTIVATED`, as shown in Figure 9.

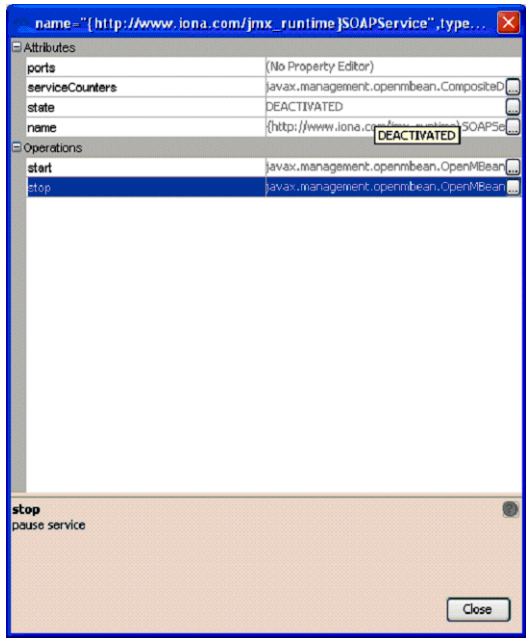

Figure 9: *Deactivated Service*

6. Click the **...** button at the right of **start** operation on SOAP service properties. This displays a dialog for the **start** operation, which is the same as the one shown in Figure 8.

7. Click **Execute...** to restart the service. In the service properties dialog, the state of the `SOAPService` becomes `ACTIVATED`, as shown in Figure 10.

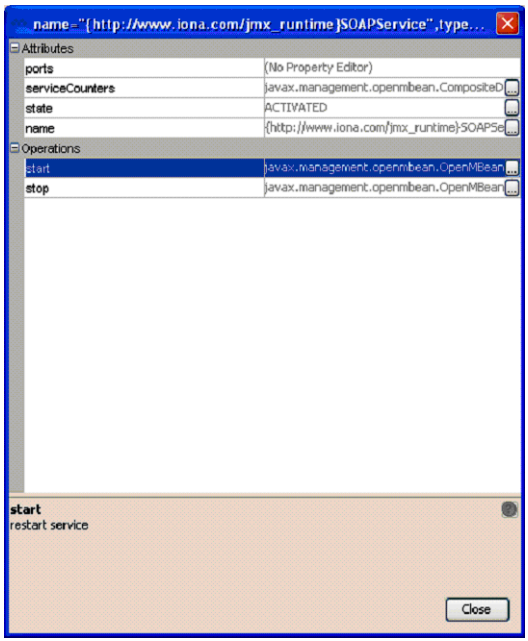

Figure 10: *Activated a Service*

Monitoring a service port

To monitor an example service port in the Mc4J console, perform the following steps:

1. Click the following node in the left panel of the MC4J console:

   `name=SoapPort,tyoe=Bus.Service.Port`

   This displays the attributes for `SoapPort`, as shown in Figure 11.

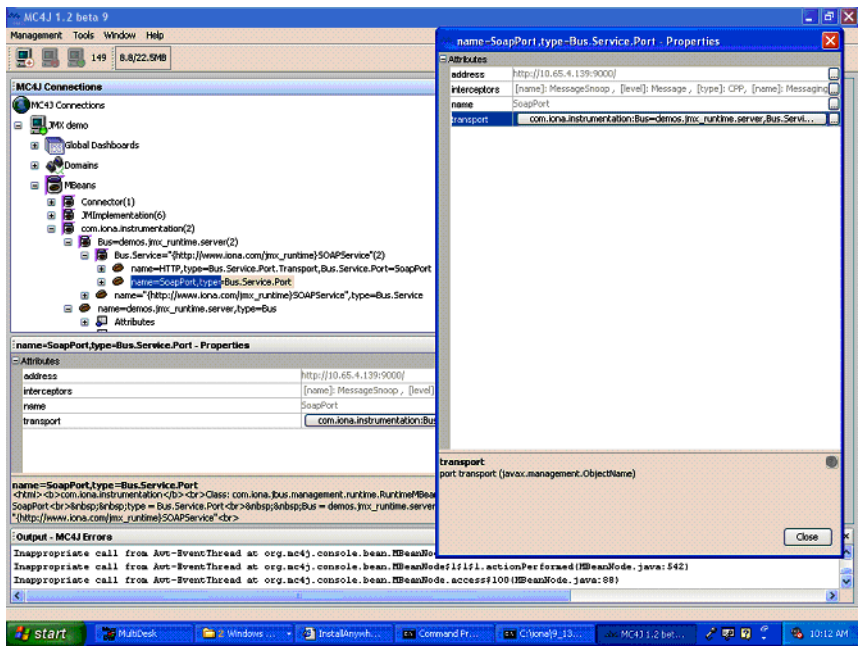

Figure 11: *Viewing Port Properties*

2. Click the **...** button at the right of the `interceptors` attribute in Figure 11. This displays the `interceptors` properties for the selected bus, as shown in Figure 12.

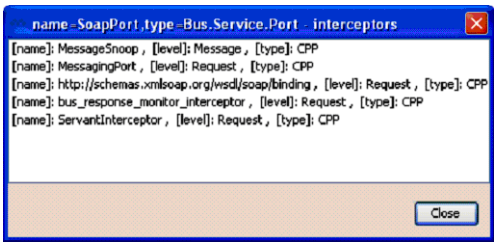

Figure 12: *Viewing Interceptor Properties*

Further information

For full details on using the MC4J management console, see the MC4J documentation:

http://mc4j.org/confluence/display/MC4J/User+Guide

Managing Logging Levels with MC4J

Overview

This section uses the `jmx_runtime` Artix demo to show a detailed walk-through example of how to use the MC4J console to manage Artix bus logging levels dynamically at runtime.

Defined demo logging configuration

The following logging configuration is defined in the `demos.jmx_runtime` configuration scope:

Logging Subsystem	Logging Level
`IT_BUS`	`LOG_ERROR`
`IT_BUS.CORE`	`LOG_INFO_LOW`

This means that the logging level for `IT_BUS`, and all of its child subsystems, is `LOG_ERROR`. The only exception is `IT_BUS.CORE`, which has a logging level of `LOG_INFO_LOW`.

Viewing logging levels for a subsystem

To view logging levels for a specified Artix logging subsystem in MC4J, perform the following steps:

1. Expand the following tree node in the left panel of MC4J:

   `name=demos.jmx_runtime.server,type=Bus`
2. Expand the `Operations` node.
3. Double click `getLoggingLevel`. This displays the **My Wizard** screen, as shown in Figure 13.

You can use this wizard to view the logging level of a specified subsystem.

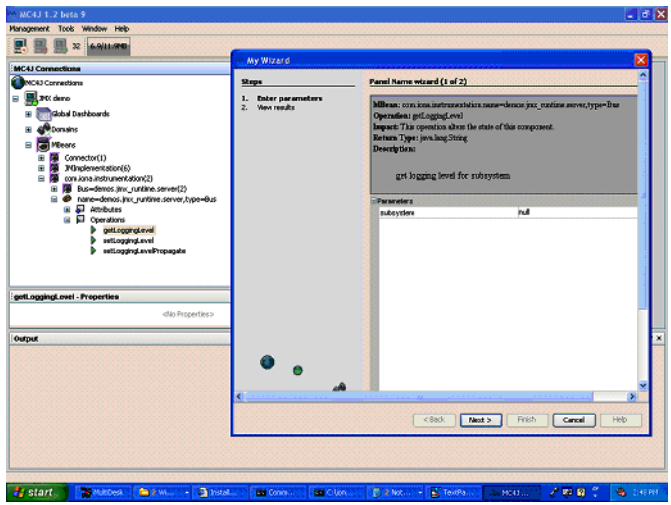

Figure 13: *Logging Viewing Wizard*

4. Enter the `IT_BUS` subsystem, as shown in Figure 14.

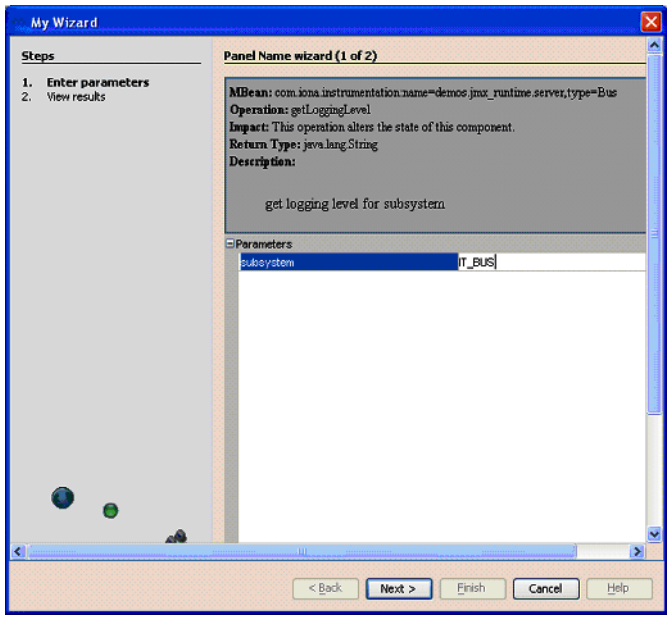

Figure 14: *Entering a Logging Subsystem*

5. Click **Next**. This displays the logging level of `IT_BUS` as `LOG_ERROR`, as shown in Figure 15.
6. Click **Finish**.

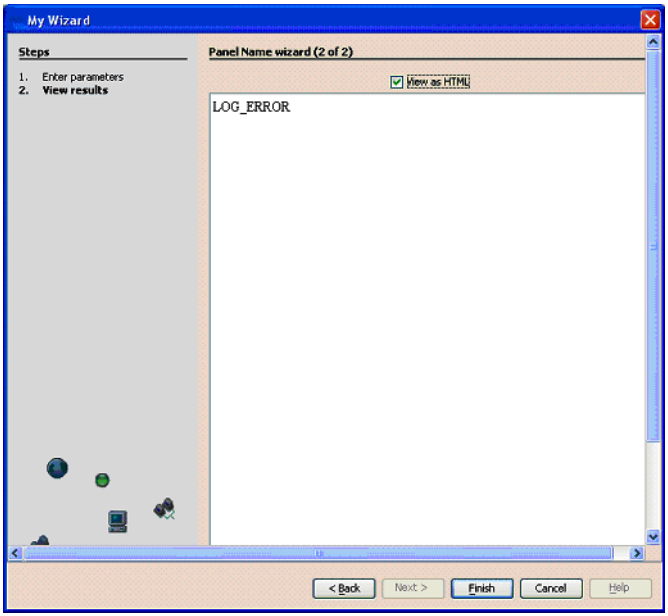

Figure 15: *Displayed Logging Level*

7. Similarly, use the **My Wizard** screen to enter a logging subsystem of `IT_BUS.INITIAL_REFERENCE`.
8. Click **Next**. The logging level for the `IT_BUS.INITIAL_REFERENCE` subsystem is also displayed as `LOG_ERROR`. The `IT_BUS.INITIAL_REFERENCE` subsystem inherits the same logging level from its `IT_BUS` parent.
9. Finally, use the **My Wizard** screen to enter a logging subsystem of `IT_BUS.CORE`.
10. Click **Next.** The logging level for `IT_BUS.CORE` is displayed as `LOG_INFO_LOW`. The logging level for `IT_BUS.CORE` has been configured differently from its `IT_BUS` parent (see "Defined demo logging configuration" on page 57).

Setting the logging level for a subsystem

To set the logging level for a specified logging subsystem, perform the following steps:

1. Double click the `setLoggingLevel` node in the left panel of the MC4J console. This displays the **My Wizard** screen, as show in Figure 16.
2. Enter `IT_BUS` for the `subsystem`, and `LOG_WARN` for the logging `level`, as as show in Figure 16.

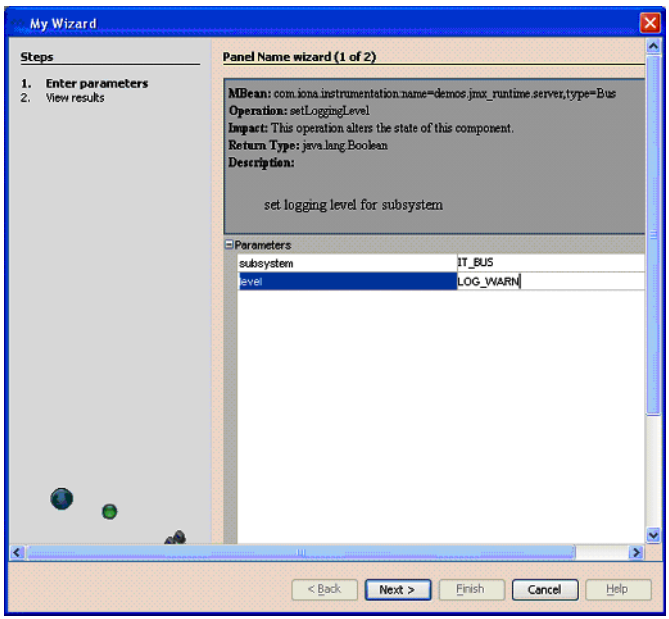

Figure 16: *Setting a Logging Level*

3. Click **Next**. This displays `true`, as shown in Figure 17, which means that the logging level is set successfully.

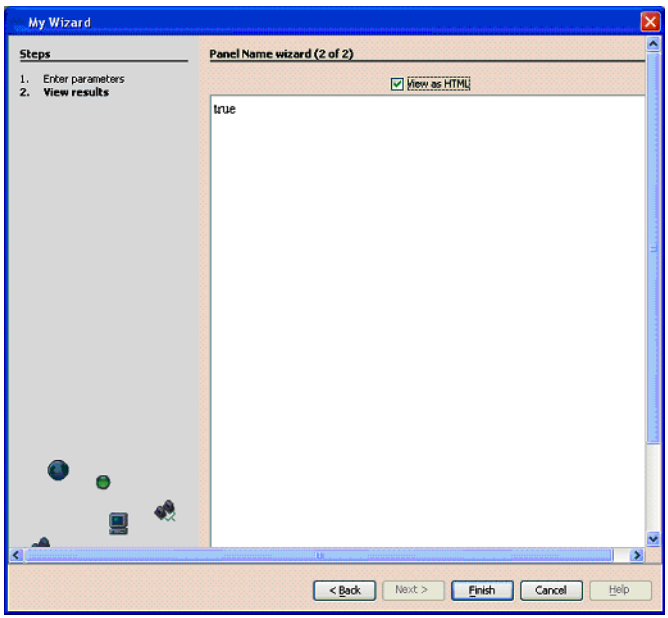

Figure 17: *Logging Level Set Successfully*

4. View the logging level of the `IT_BUS` subsystem to verify your setting (as described in "Viewing logging levels for a subsystem" on page 57). The logging level for `IT_BUS` is now `LOG_WARN`.
5. View the logging level for the `IT_BUS.INITIAL_REFERENCE` subsystem. The logging level for `IT_BUS.INITIAL_REFERENCE` is also `LOG_WARN`.
6. View the logging level for `IT_BUS.CORE`. The logging level of `IT_BUS.CORE` is still `LOG_INFO_LOW`. It does not inherit the `LOG_WARN` level from its parent because its logging level has been configured separately (see "Defined demo logging configuration" on page 57).

Setting the logging level for a subsystem with propagation

To set a logging level to override a child subsystem with a separately configured logging level, perform the following steps:

1. Double click the `setLoggingLevelPropagate` tree node in left panel of MC4J. This displays the **My Wizard** screen, as shown in Figure 17.

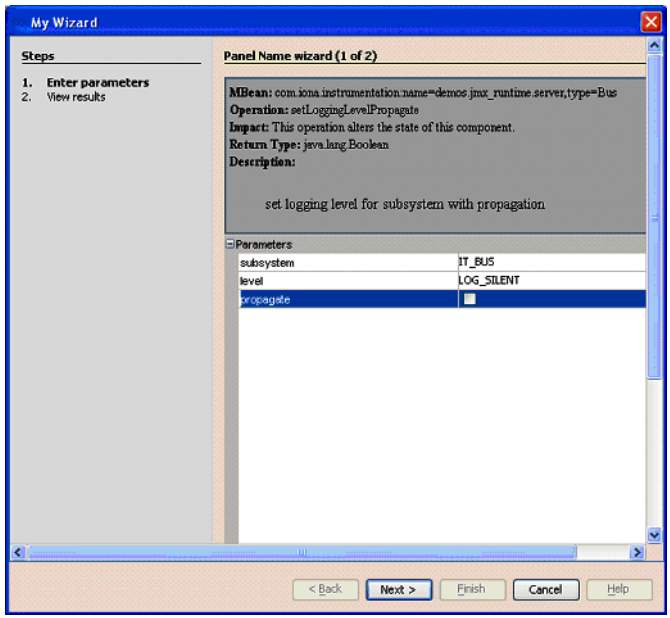

Figure 18: *Propagating a Logging Level*

2. Enter `IT_BUS` as the `subsystem`, and `LOG_SILENT` as the logging level.
3. Click **Next**. The returned value is `true`, which means that the logging level is set successfully.
4. View the logging level for `IT_BUS` (as described in "Viewing logging levels for a subsystem" on page 57). The logging level for `IT_BUS` is `LOG_SILENT`.

5. View the logging level for `IT_BUS.INITIAL_REFERENCE`. The logging level for `IT_BUS.INITIAL_REFERENCE` is also `LOG_SILENT`.
6. View the logging level for `IT_BUS.CORE`. The logging level for `IT_BUS.CORE` is also `LOG_SILENT`. Specifying propagation overrides log levels for all child logging subsystems.

Further information

For detailed information on Artix logging, see Configuring and Deploying Artix Solutions.

Managing Artix Services with JConsole

Overview

You can also use JConsole, which is provided with JDK 1.5, to monitor and manage Artix applications. JConsole displays Artix runtime managed components in a hierarchical tree, as shown in Figure 19.

Using JConsole

To use JConsole, perform the following steps:

1. Start up JConsole using the following command:
   `JDK_HOME/bin/jconsole`
2. Select the **Advanced** tab.
3. Enter or paste a JMXServiceURL (either the default URL, or one copied from a published `connector.url` file).

Managing services

Figure 19 shows the attributes displayed for a managed service component (for example, the `serviceCounters` performance metrics displayed in the right pane). For detailed information on these attributes, see "Service attributes" on page 30.

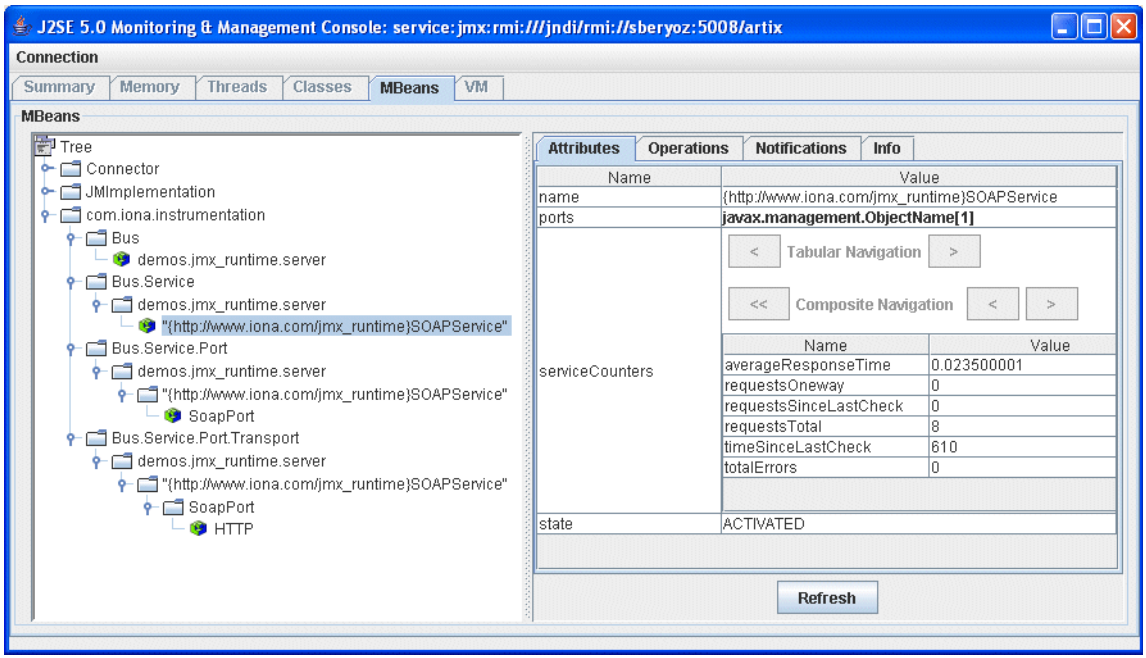

Figure 19: *Managed Service in JConsole*

Managing ports

Figure 20 shows the attributes displayed for a managed port component (for example, the `interceptors` list displayed in the right pane). For detailed information on these attributes, see "Port attributes" on page 37.

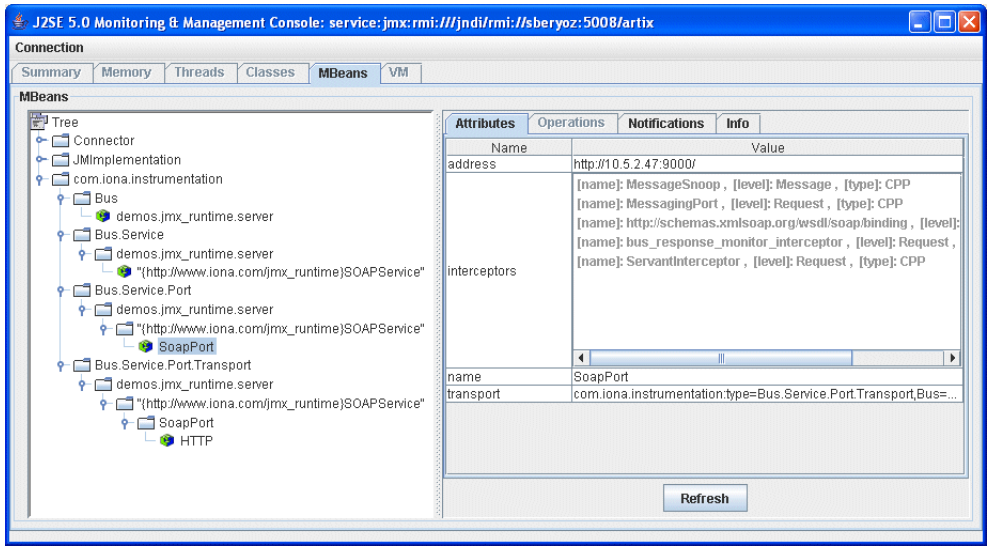

Figure 20: *Managed Port in JConsole*

Managing containers

Figure 21 shows an example of a locator service deployed into an Artix container. For more information, see "Locator attributes" on page 34.

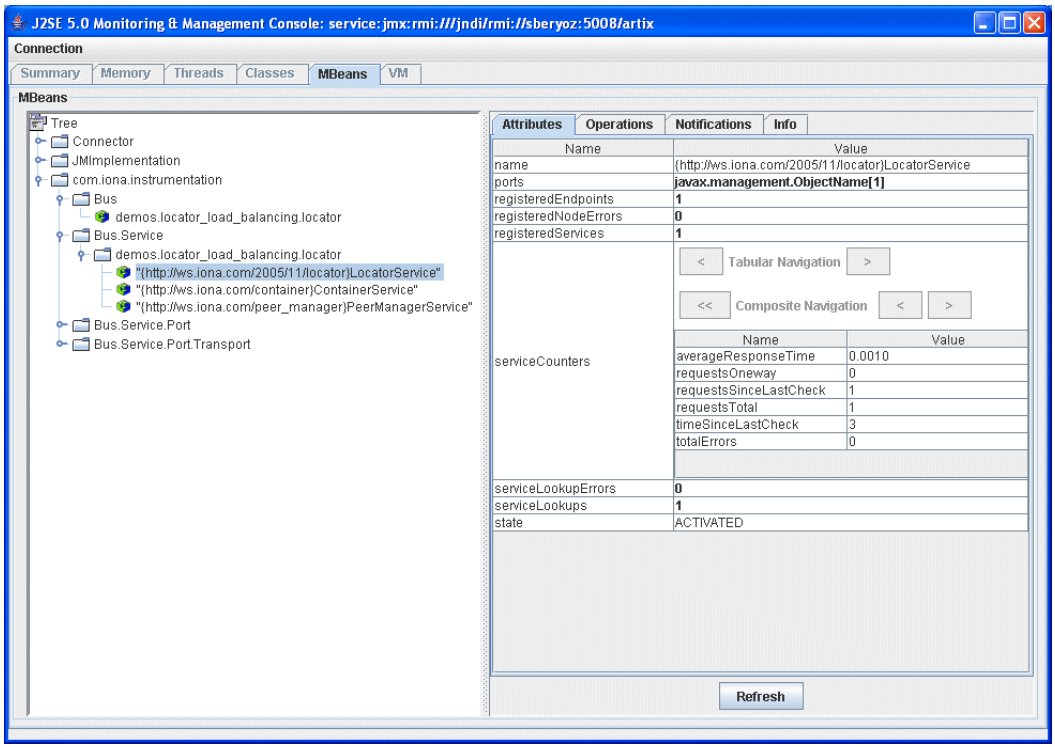

Figure 21: *Managed Locator in JConsole*

Note: When using a JMX console to manage a service running in an Artix container, set the `serviceMonitoring` attribute to `true` to enable service performance monitoring (see "Bus attributes" on page 24).

Further information

For more information on using JConsole, see the following:

http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html

Managing Artix Services with the JMX HTTP adaptor

Overview

You can also manage Artix services using the default HTTP adaptor console that is provided with the JMX reference implementation. This console is browser-based, as shown in Figure 22.

Using the JMX HTTP adaptor

To use the JMX HTTP adaptor, perform the following steps:

1. Specify following configuration settings:

```
plugins:bus_management:http_adaptor:enabled="true";
plugins:bus_management:http_adaptor:port="7659";
```

2. Enter the following URL in your browse:

   `http://localhost:7659`

   This displays the main HTTP adaptor management view, as shown in Figure 22.

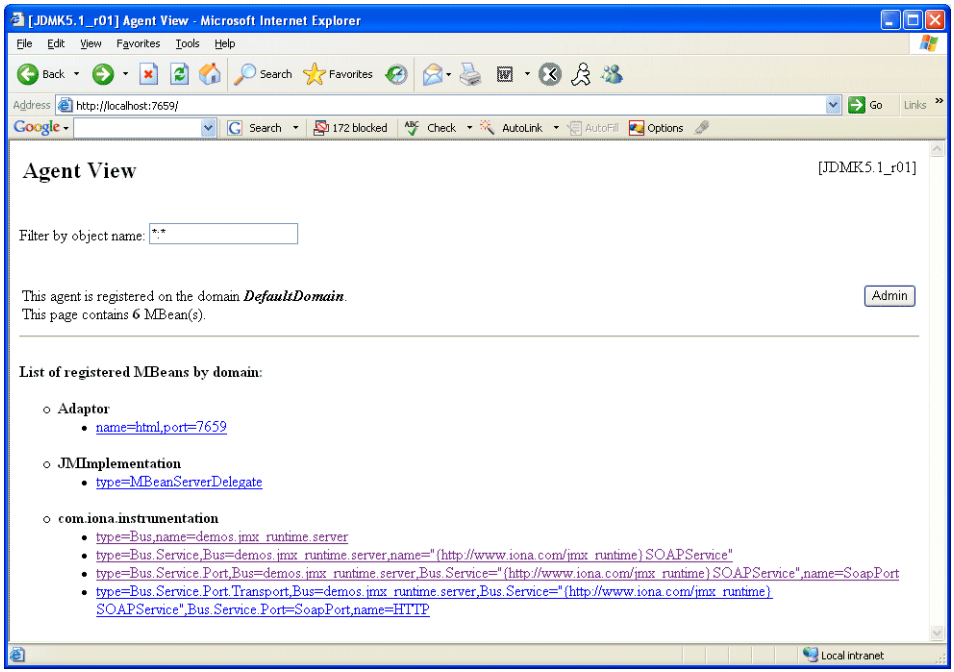

Figure 22: *HTTP Adaptor Main View*

Figure 23 shows the attributes displayed for a managed bus component (for example, the `services` that it includes). For detailed information on these attributes, see "Bus attributes" on page 24.

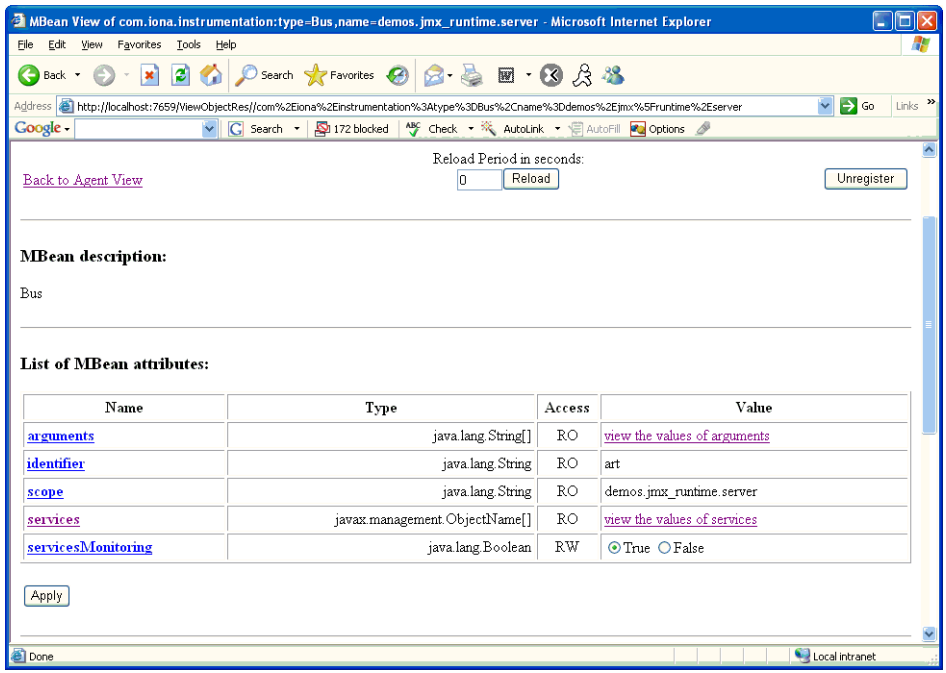

Figure 23: *HTTP Adaptor Bus View*

Further information

For further information on using the HTTP JMX adaptor, see the following:

http://java.sun.com/developer/technicalArticles/J2SE/jmx.html

Index

T