



# Artix™

---

Managing Artix Solutions with  
JMX

Version 4.2, March 2007

IONA Technologies PLC and/or its subsidiaries may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this publication. Except as expressly provided in any written license agreement from IONA Technologies PLC, the furnishing of this publication does not give you any license to these patents, trademarks, copyrights, or other intellectual property. Any rights not expressly granted herein are reserved.

IONA, IONA Technologies, the IONA logos, Orbix, Artix, Making Software Work Together, Adaptive Runtime Technology, Orbacus, IONA University, and IONA XMLBus are trademarks or registered trademarks of IONA Technologies PLC and/or its subsidiaries.

Java and J2EE are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. CORBA is a trademark or registered trademark of the Object Management Group, Inc. in the United States and other countries. All other trademarks that appear herein are the property of their respective owners.

While the information in this publication is believed to be accurate, IONA Technologies PLC makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IONA shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

---

#### COPYRIGHT NOTICE

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of IONA Technologies PLC. No third-party intellectual property right liability is assumed with respect to the use of the information contained herein. IONA Technologies PLC assumes no responsibility for errors or omissions contained in this publication. This publication and features described herein are subject to change without notice.

Copyright © 1999-2007 IONA Technologies PLC. All rights reserved.

All products or services mentioned in this publication are covered by the trademarks, service marks, or product names as designated by the companies that market those products.

Updated: March 12, 2007

# Contents

<b>List of Figures</b>	<b>5</b>
<b>List of Tables</b>	<b>7</b>
<b>Preface</b>	<b>9</b>
<b>What is Covered in this Book</b>	<b>9</b>
<b>Who Should Read this Book</b>	<b>9</b>
<b>How to Use this Book</b>	<b>9</b>
<b>The Artix Documentation Library</b>	<b>10</b>
<b>Chapter 1 Monitoring and Managing an Artix Runtime with JMX</b>	<b>11</b>
<b>Introduction</b>	<b>12</b>
<b>Managed Bus Components</b>	<b>17</b>
<b>Managed Service Components</b>	<b>23</b>
Artix Locator Service	<b>28</b>
Artix Session Manager Service	<b>30</b>
<b>Managed Port Components</b>	<b>31</b>
<b>Chapter 2 Configuring JMX in an Artix Runtime</b>	<b>35</b>
<b>Artix JMX Configuration</b>	<b>36</b>
<b>Chapter 3 Managing Artix Services with JMX Consoles</b>	<b>39</b>
<b>Managing Artix Services with MC4J</b>	<b>40</b>
Managing Logging Levels with MC4J	<b>51</b>
<b>Managing Artix Services with JConsole</b>	<b>59</b>
<b>Managing Artix Services with the JMX HTTP adaptor</b>	<b>63</b>
<b>Chapter 4 Managing WS-RM Persistence with JMX</b>	<b>67</b>
<b>WS-RM Persistence Management</b>	<b>68</b>
<b>Viewing Messages in the WS-RM Persistence Database</b>	<b>70</b>
<b>Index</b>	<b>77</b>

## CONTENTS

# List of Figures

Figure 1: Artix JMX Architecture	13
Figure 2: Connecting to a Server	41
Figure 3: Server Connection Details	42
Figure 4: Creation of Server Connection	43
Figure 5: New Server Connection	44
Figure 6: Viewing Service Properties	45
Figure 7: Viewing Service Counters Properties	46
Figure 8: Stopping a Service	46
Figure 9: Deactivated Service	47
Figure 10: Activated a Service	48
Figure 11: Viewing Port Properties	49
Figure 12: Viewing Interceptor Properties	50
Figure 13: Logging Viewing Wizard	52
Figure 14: Entering a Logging Subsystem	53
Figure 15: Displayed Logging Level	54
Figure 16: Setting a Logging Level	55
Figure 17: Logging Level Set Successfully	56
Figure 18: Propagating a Logging Level	57
Figure 19: Managed Service in JConsole	60
Figure 20: Managed Port in JConsole	61
Figure 21: Managed Locator in JConsole	62
Figure 22: HTTP Adaptor Main View	64
Figure 23: HTTP Adaptor Bus View	65
Figure 24: Connecting to a JMX Agent	71
Figure 25: WS-RM Persistence Enabled Endpoint	72
Figure 26: WS-RM Persistence Enabled Client	74

Figure 27: WS-RM Persistence Enabled Messages

75

# List of Tables

Table 1: Managed Bus Attributes	18
Table 2: Managed Bus Methods	19
Table 3: Managed Service Attributes	24
Table 4: serviceCounters Attributes	25
Table 5: Managed Service Attributes	26
Table 6: Locator MBean Attributes	28
Table 7: Session Manager MBean Attributes	30
Table 8: Supported Service Attributes	31

## 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.

- [Chapter 4](#) explains how to manage Web services Reliable Messaging persistence in Artix using JMX.

### **The Artix Documentation Library**

For information on the organization of the Artix library, the document conventions used, and where to find additional resources, see [Using the Artix Library](#).

# 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:

<a href="#">Introduction</a>	<a href="#">page 12</a>
<a href="#">Managed Bus Components</a>	<a href="#">page 17</a>
<a href="#">Managed Service Components</a>	<a href="#">page 23</a>
<a href="#">Managed Port Components</a>	<a href="#">page 31</a>

# 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 14](#)).

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

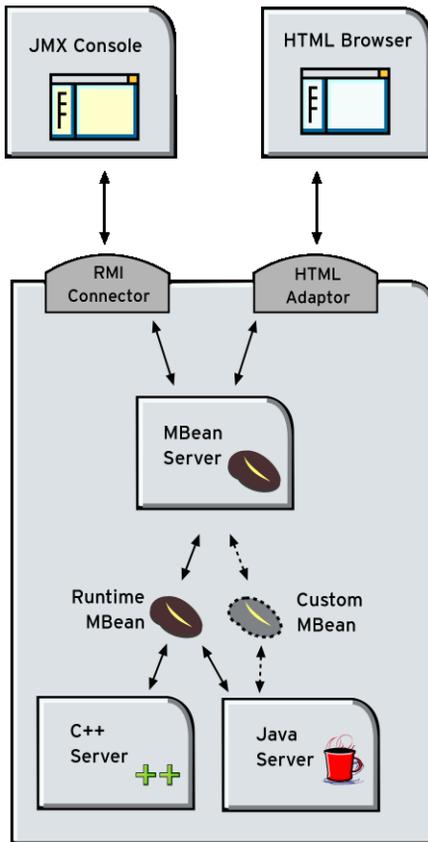


Figure 1: *Artix JMX Architecture*

---

## 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 16 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_runtime
```

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,name=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
<code>scope</code>	Bus scope used to initialize a bus.	<code>String</code>	No
<code>identifier</code>	Bus identifier, typically the same as its scope.	<code>String</code>	No
<code>arguments</code>	Bus arguments, including the executable name.	<code>String[]</code>	No
<code>servicesMonitoring</code>	Used to enable/disable services performance monitoring.	<code>Boolean</code>	Yes
<code>services</code>	A list of object names representing services on this bus.	<code>ObjectName[]</code>	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 [“Managing Artix Services with JMX Consoles” on page 39](#).

**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.ionia.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 reenale 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. For information on WS-RM persistence instrumentation, see [Chapter 4](#).

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(),
    serviceWsdURL, 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
<code>name</code>	Service <code>QName</code> in expanded form.	<code>String</code>	No
<code>state</code>	Service state.	<code>String</code>	No
<code>serviceCounters</code>	Service performance data.	<code>CompositeData</code>	No
<code>ports</code>	A list of <code>ObjectNames</code> representing ports for this service.	<code>ObjectName[]</code>	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 [“Managing Artix Services with JMX Consoles” on page 39](#)

**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 its own specific set of attributes.

---

### Locator attributes

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

**Table 6:** *Locator MBean Attributes*

Name	Description	Type
<code>registeredEndpoints</code>	Number of registered endpoints.	Integer
<code>registeredServices</code>	Number of registered services, less or equal to number of endpoints.	Integer
<code>serviceLookups</code>	Number of service lookup requests.	Integer
<code>serviceLookupErrors</code>	Number of service lookup failures.	Integer
<code>registeredNodeErrors</code>	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(serviceName, "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 its own specific 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.ionainstrumentation:type=Bus.Service" +
    ",name=\"{http://ws.ionainstrumentation.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 [“Managing Artix Services with JMX Consoles” on page 39](#)

**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.ionainstrumentation:type=Bus.Service.Port" +
    ",name=SoapPort" +

    ",Bus.Service=\"{http://www.ionainstrumentation.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
}

```

# 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:

<a href="#">Artix JMX Configuration</a>
---

page 36
---------

---

# 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>

# Managing Artix Services with JMX Consoles

*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:

<a href="#">Managing Artix Services with MC4J</a>	page 40
<a href="#">Managing Artix Services with JConsole</a>	page 59
<a href="#">Managing Artix Services with the JMX HTTP adaptor</a>	page 63

---

# 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\Version\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\Version\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\Version\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](#).

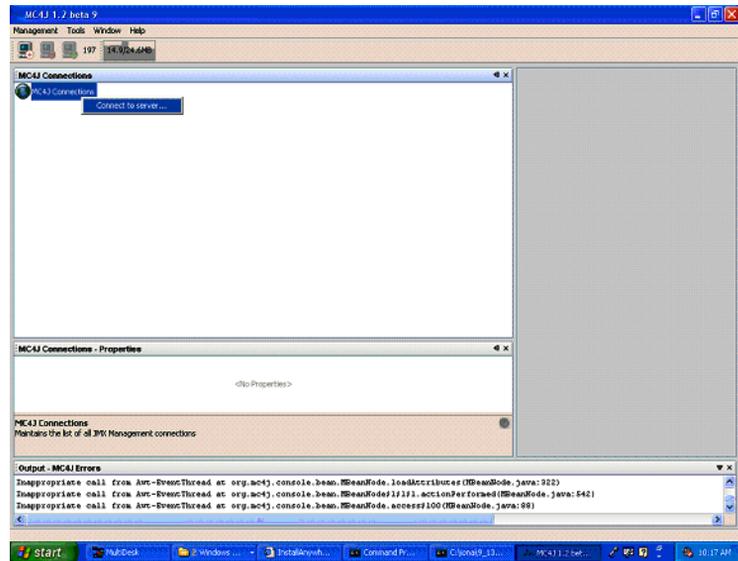
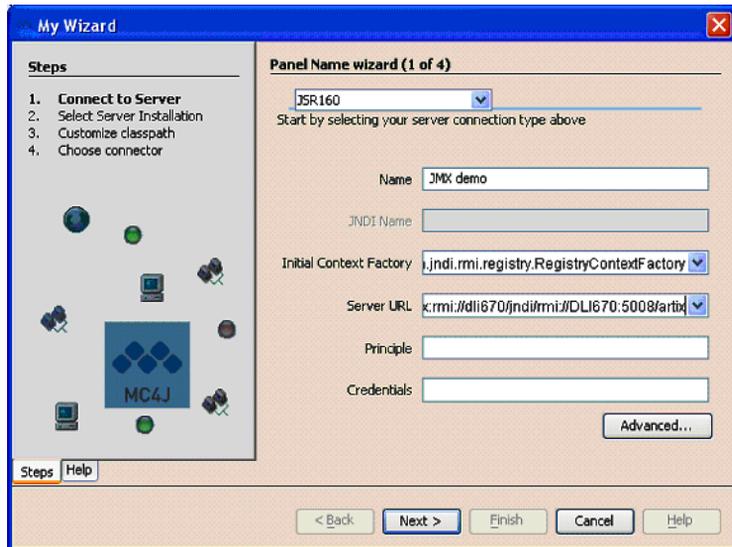


Figure 2: *Connecting to a Server*

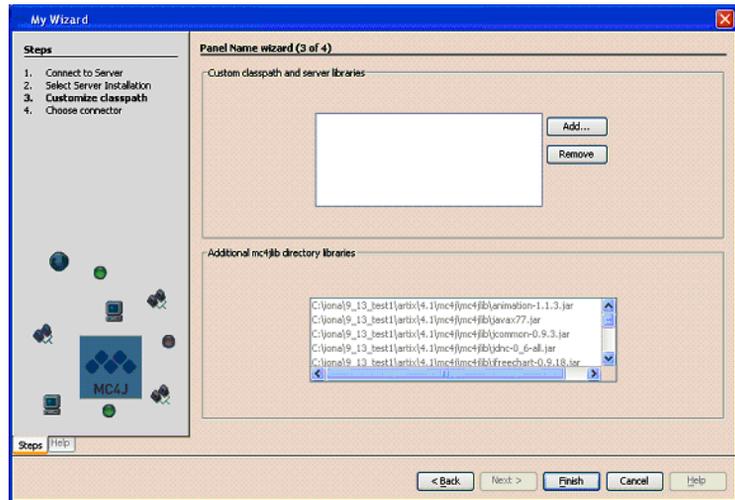
- Click **Connection server...** to launch the **My wizard** dialog, as shown in Figure 3.



**Figure 3:** *Server Connection Details*

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

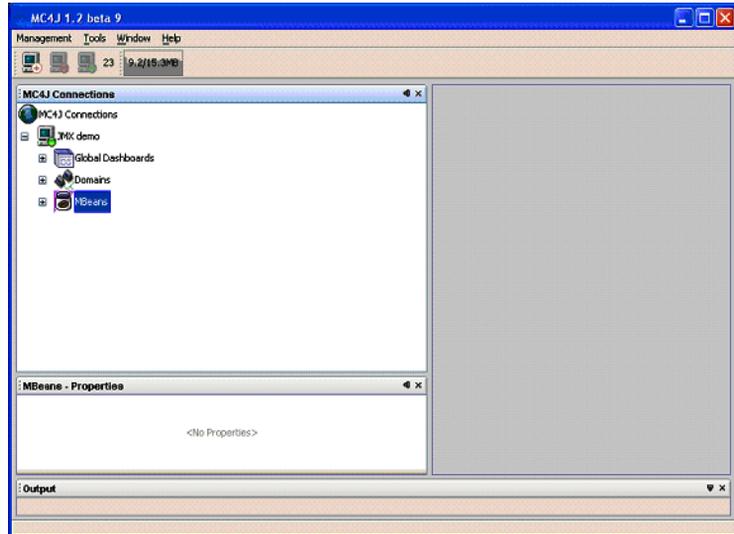
- Click **Next** to go to next screen, as shown in [Figure 4](#).



**Figure 4:** *Creation of Server Connection*

- 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.

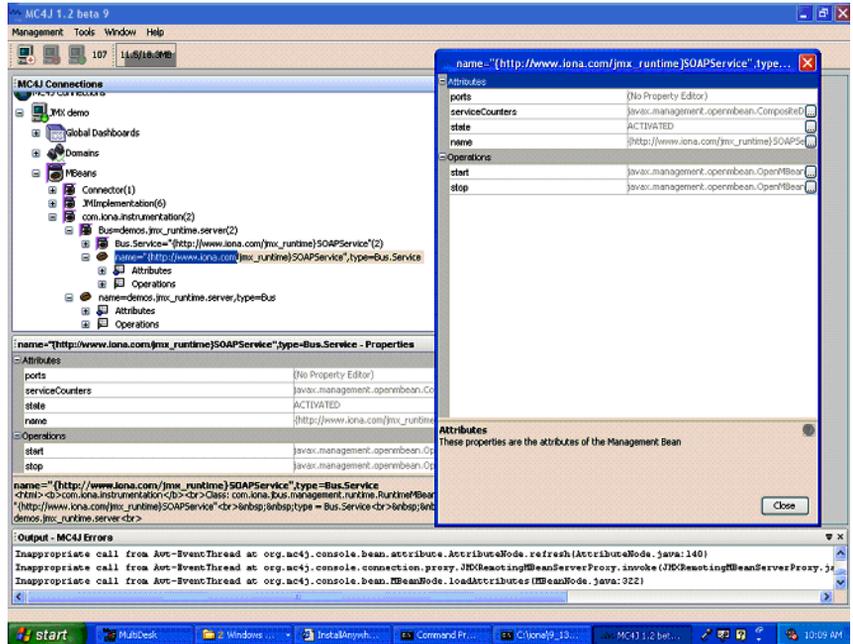
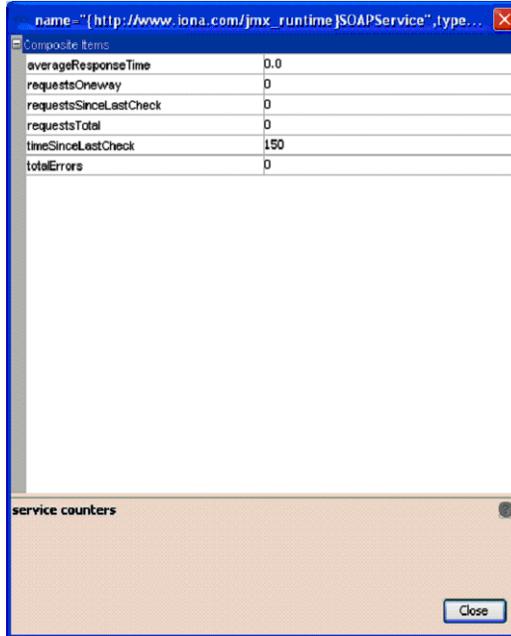


Figure 6: Viewing Service Properties

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](#).



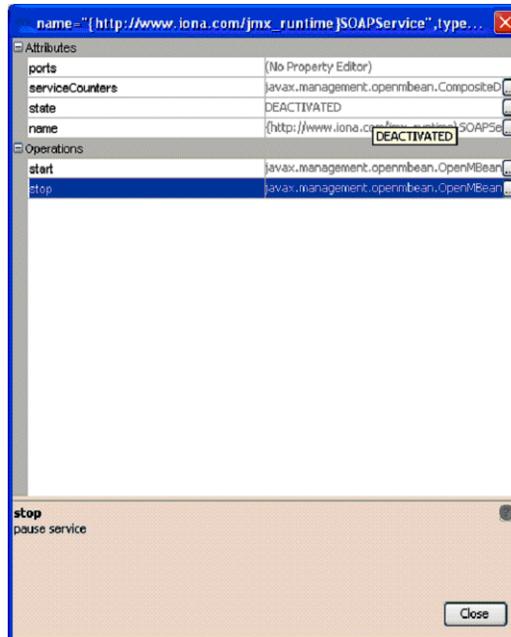
**Figure 7:** *Viewing Service Counters Properties*

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*

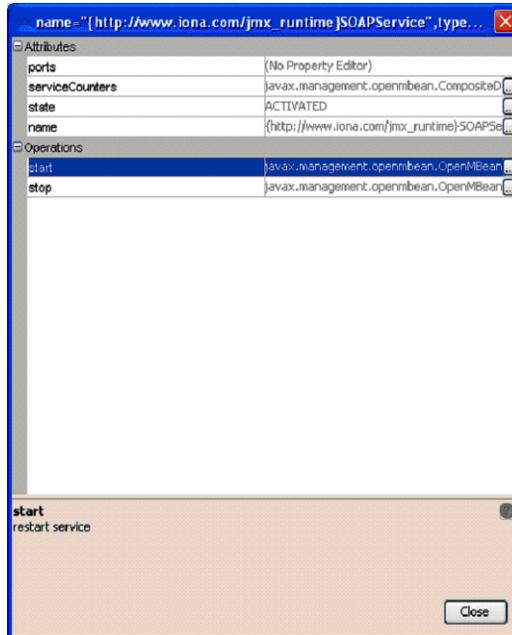
- Click **Execute...** to stop the service. In the SOAPservice properties dialog, the state attribute of the service becomes `DEACTIVATED`, as shown in [Figure 9](#).



**Figure 9:** *Deactivated Service*

- 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, type=Bus.Service.Port
```

This displays the attributes for SoapPort, as shown in [Figure 11](#).

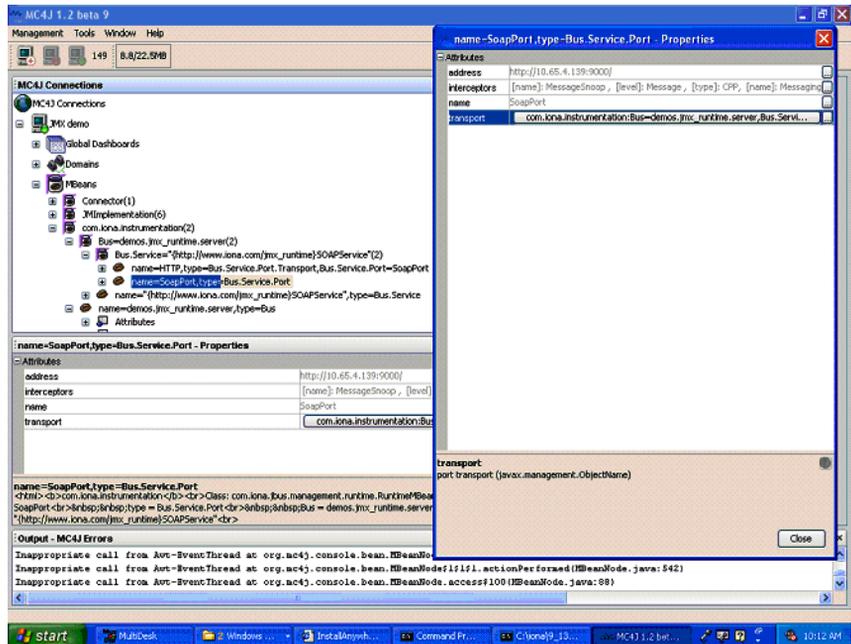
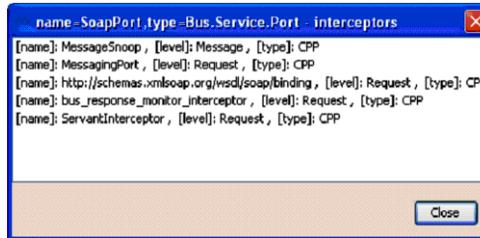


Figure 11: Viewing Port Properties

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
<code>IT_BUS</code>	<code>LOG_ERROR</code>
<code>IT_BUS.CORE</code>	<code>LOG_INFO_LOW</code>

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.

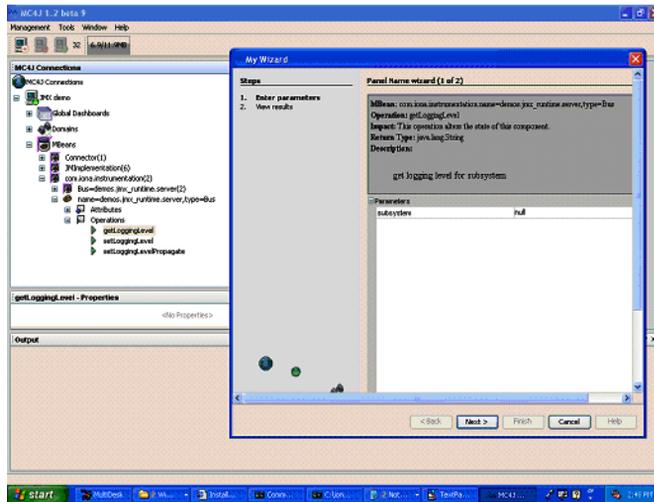
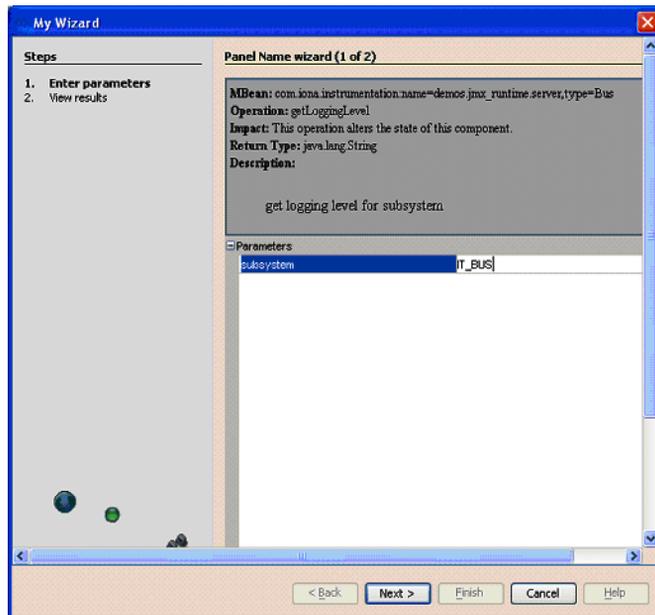


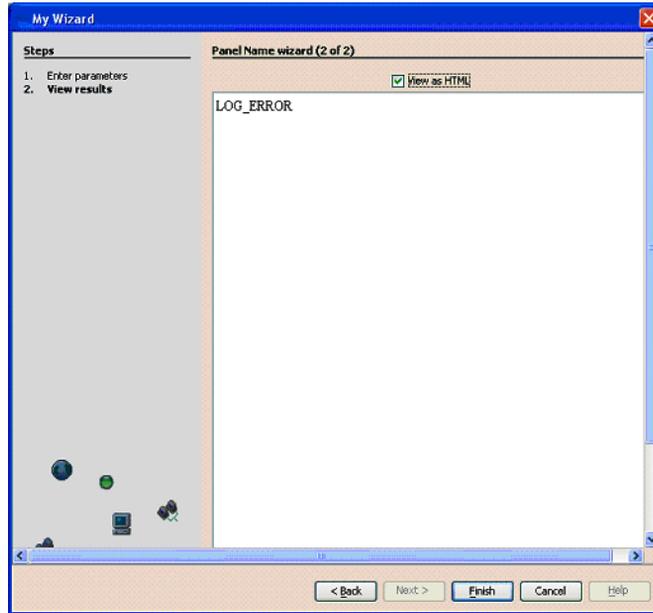
Figure 13: Logging Viewing Wizard

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 51](#)).

## 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](#).

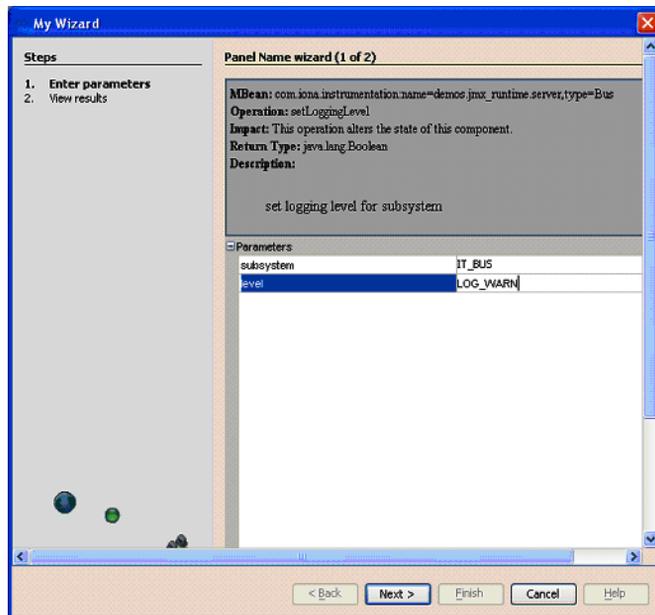
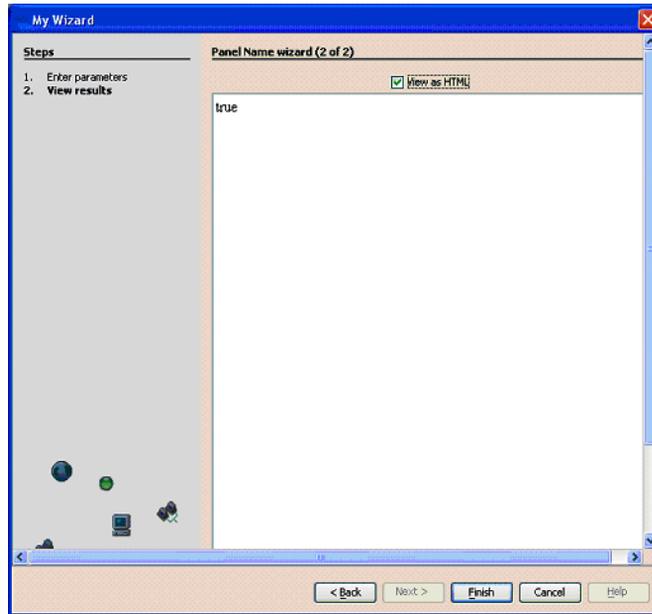


Figure 16: Setting a Logging Level

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



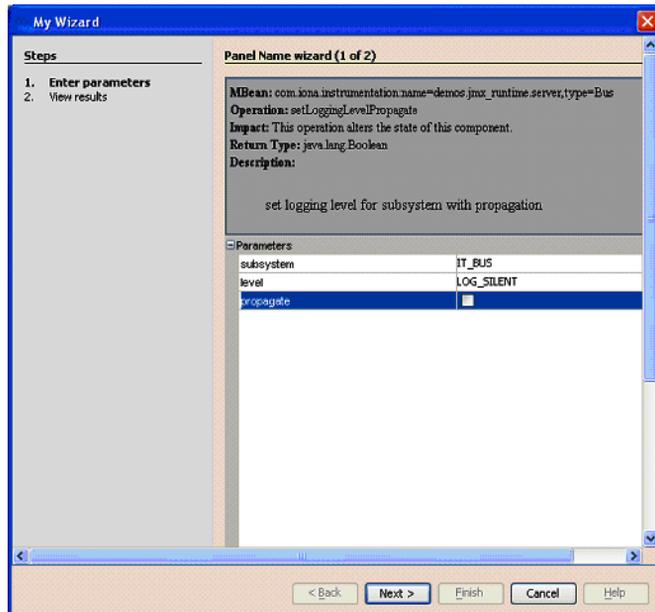
**Figure 17:** *Logging Level Set Successfully*

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 51). 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 51).

## 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 51). 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 24.

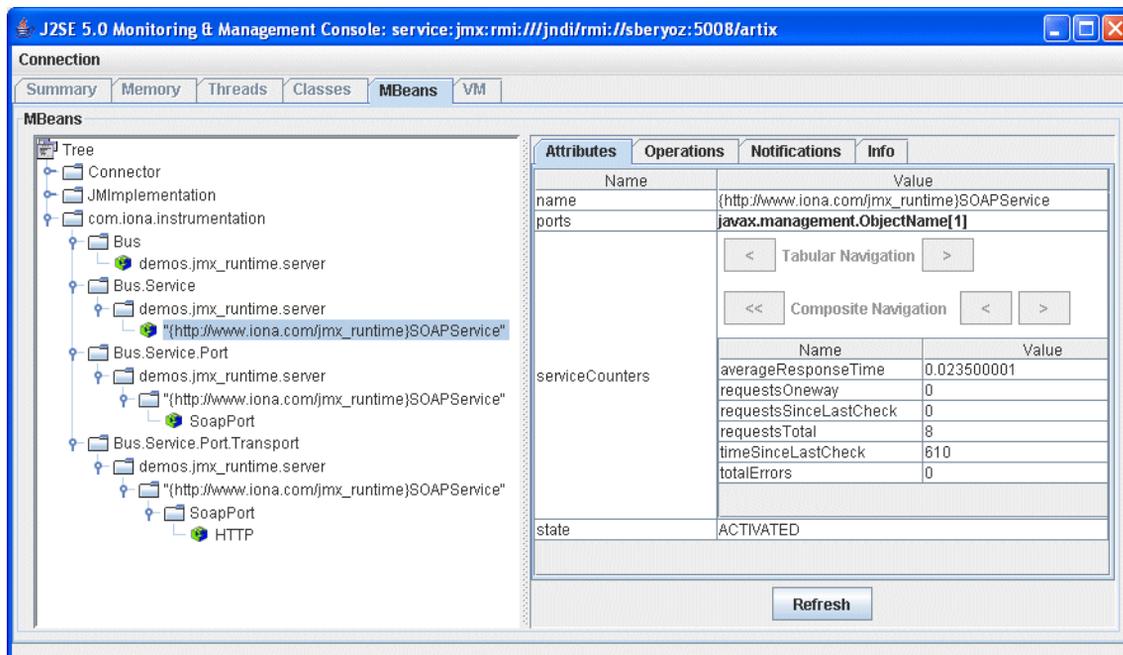


Figure 19: Managed Service in JConsole

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 31.

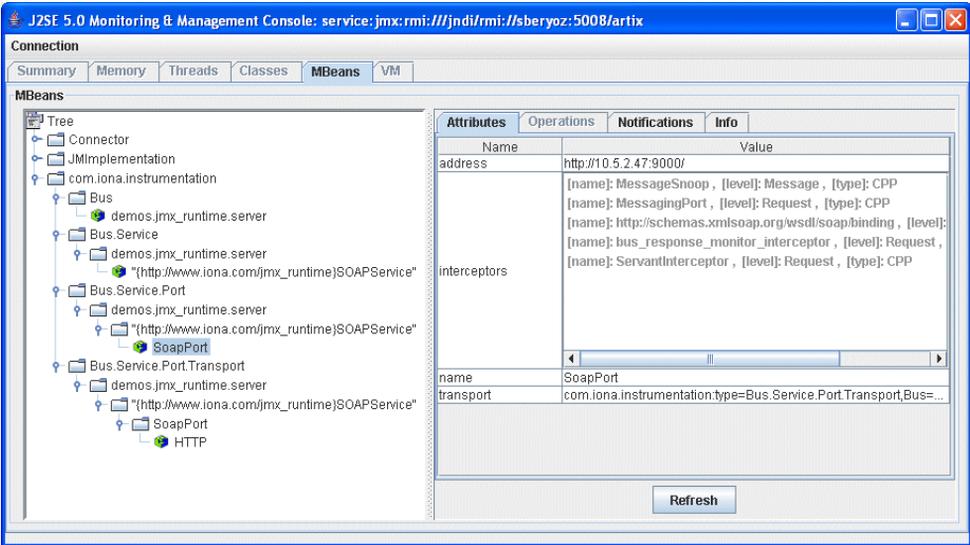


Figure 20: Managed Port in JConsole

## Managing containers

Figure 21 shows an example of a locator service deployed into an Artix container. For more information, see “Locator attributes” on page 28.

The screenshot shows the J2SE 5.0 Monitoring & Management Console window. The title bar reads "J2SE 5.0 Monitoring & Management Console: service:jmx:rmi:///jndi/rmi://sberyoz:5008/artix". The window has tabs for Summary, Memory, Threads, Classes, MBeans, and VM. The MBeans tab is active, showing a tree view on the left and a detailed view on the right.

The tree view on the left shows the following structure:

- Tree
  - Connector
  - JMImplementation
  - com.ionainstrumentation
    - Bus
      - demos.locator\_load\_balancing.locator
        - Bus.Service
          - demos.locator\_load\_balancing.locator
            - {http://ws.ionainstrumentation.com/2005/11/locator}LocatorService
            - {http://ws.ionainstrumentation.com/container}ContainersService
            - {http://ws.ionainstrumentation.com/peer\_manager}PeerManagerService
          - Bus.Service.Port
          - Bus.Service.Port.Transport

The detailed view on the right shows the following attributes:

Name	Value														
name	{http://ws.ionainstrumentation.com/2005/11/locator}LocatorService														
ports	javax.management.ObjectName[1]														
registeredEndpoints	1														
registeredNodeErrors	0														
registeredServices	1														
<div style="text-align: center;"> <input type="button" value="Tabular Navigation"/> </div>															
<div style="text-align: center;"> <input type="button" value="Composite Navigation"/> </div>															
<table border="1"> <thead> <tr> <th>Name</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>averageResponseTime</td> <td>0.0010</td> </tr> <tr> <td>requestsOneway</td> <td>0</td> </tr> <tr> <td>requestsSinceLastCheck</td> <td>1</td> </tr> <tr> <td>requestsTotal</td> <td>1</td> </tr> <tr> <td>timeSinceLastCheck</td> <td>3</td> </tr> <tr> <td>totalErrors</td> <td>0</td> </tr> </tbody> </table>		Name	Value	averageResponseTime	0.0010	requestsOneway	0	requestsSinceLastCheck	1	requestsTotal	1	timeSinceLastCheck	3	totalErrors	0
Name	Value														
averageResponseTime	0.0010														
requestsOneway	0														
requestsSinceLastCheck	1														
requestsTotal	1														
timeSinceLastCheck	3														
totalErrors	0														
serviceCounters															
serviceLookupErrors	0														
serviceLookups	1														
state	ACTIVATED														

At the bottom of the detailed view is a "Refresh" button.

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 18).

## 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](#).

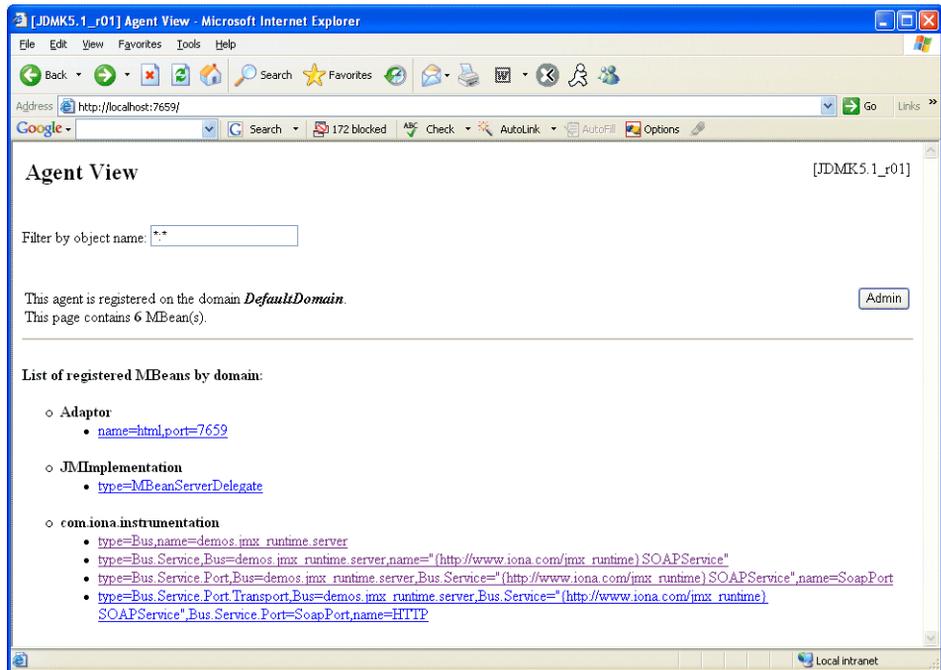


Figure 22: HTTP Adaptor Main View

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 18.

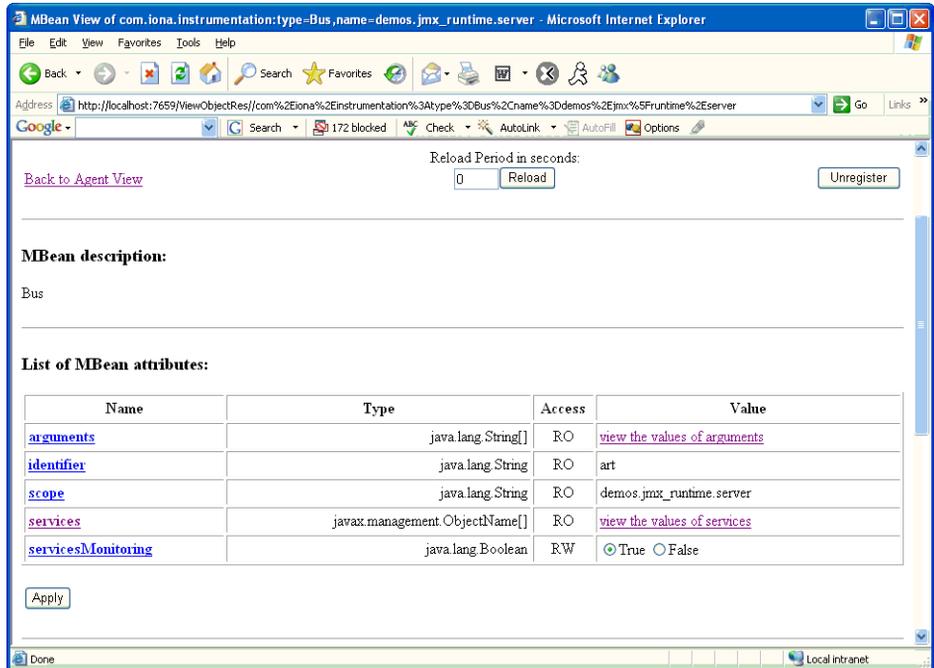


Figure 23: HTTP Adaptor Bus View

**Further information**

For further information on using the HTTP JMX adaptor, see the following:  
<http://java.sun.com/developer/technicalArticles/J2SE/jmx.html>



# Managing WS-RM Persistence with JMX

*You can manage Web Services Reliable Messaging persistence in Artix using any JMX console.*

---

**In this chapter**

This chapter discusses the following topics:

<a href="#">WS-RM Persistence Management</a>	<a href="#">page 68</a>
<a href="#">Viewing Messages in the WS-RM Persistence Database</a>	<a href="#">page 70</a>

# WS-RM Persistence Management

## Overview

You can use any JMX console to view messages in the WS-RM persistence database both locally and remotely. You also can monitor the WS-RM persistence enabled endpoint, acknowledgement URI, and the client sequence ID. This section explains the WS-RM persistence information that can be managed in a JMX console.

## Managed WS-RM persistence components

The following WS-RM persistence components can be managed in a JMX console:

- Managed WS-RM persistence endpoints  
(*RMEndpointPersistentStore*)
- Managed WS-RM persistence sequences  
(*RMSequencePersistentStore*)

## Managed WS-RM persistence endpoints

WS-RM persistence endpoint managed components are used to represent WS-RM persistence enabled endpoints. When a WS-RM persistence destination endpoint is created, it is registered as an MBean. When an WS-RM persistence destination endpoint is closed, the MBean is unregistered from the MBeanServer.

The MBean naming convention is as follows:

```
com.ionainstrumentation:type=Bus.Service.Port.EndpointPersistent,
name=WSRM_ENDPOINT_PERSISTENCE,
Bus.Service.Port=portName,
Bus.Service="{namespace}localname",
Bus=busIdentifier
```

## WS-RM persistence endpoint attributes

You can view the following attributes for a WS-RM persistence endpoint in a JMX console:

	Name	Description	Type
	service name	WS-RM persistence enabled service name	String
	port name	WS-RM persistence enabled port	String

## Managed WS-RM persistence sequences

WS-RM persistence sequence managed components are used to represent WS-RM persistence clients. A destination sequence with a unique ID is created for each client. When a WS-RM persistence destination sequence is created, it is registered as an MBean. When a WS-RM persistence destination sequence is recovered from database, it is also registered as an MBean. When a WS-RM persistence destination sequence is terminated, it is unregistered from the MBeanServer.

The MBean naming convention is as follows:

```
com.iona.instrumentation:type=Bus.Service.Port.EndpointPersistent.SequencePersistent,
name=sequenceName,
Bus.Service.Port.EndpointPersistent=WSRM_ENDPOINT_PERSISTENCE,
Bus.Service.Port=portName,
Bus.Service="{namespace}localName",
Bus=busIdentifier
```

In this syntax, *sequenceName* includes the string *sequence\_id* and the sequence ID.

### WS-RM persistence sequence attributes

You can view the following attributes for a WS-RM persistence sequence in a JMX console:

Name	Description	Type
acksto uri	WS-RM acknowledgement URI	String
messages	Messages in the WS-RM persistence database	String[]
sequence id	Sequence unique ID representing a client	String

The messages attribute is a list of messages in the WS-RM persistence database. The messages are returned as a list of strings, where each string has the following format:

```
[message id]: messageId [message]: soapMessage
```

---

# Viewing Messages in the WS-RM Persistence Database

---

## Overview

Before you start viewing in the WS-RM persistence database, you must set your Artix configuration to enable JMX management for WS-RM persistence. This section uses the Artix WS-RM example demo (`...demos\advanced\wsrm`) to explain how to view and monitor messages in the WS-RM persistence database.

---

## Enable JMX management for WS-RM persistence

To enable JMX management for WS-RM persistence in your Artix configuration file, perform the following steps:

1. Open the `...demos\advanced\wsrm\etc\wsrm.cfg` file.
2. Edit the `demos.wsrm_persistence_enabled.server` scope as follows:

```
server {
  plugins:artix:db:home = "./server_db";
  plugins:bus_management:enabled="true";
  plugins:bus_management:connector:enabled="true";
  plugins:bus_management:connector:url:file="../../etc/connector.url";

  # optional port, default is 1099
  plugins:bus_management:connector:port="5008";
};
```

**Note:** Enabling JMX management for WS-RM persistence is similar to enabling JMX management for other Artix components.

**Start the server**

3. To start the server, go to the `demos\advanced\wsm\bin` directory, and run the following command:

```
run_cxx_server_persistence.bat
```

This starts the server using following example command:

```
start server.exe -ORBname demos.wsm_persistence_enabled.server
```

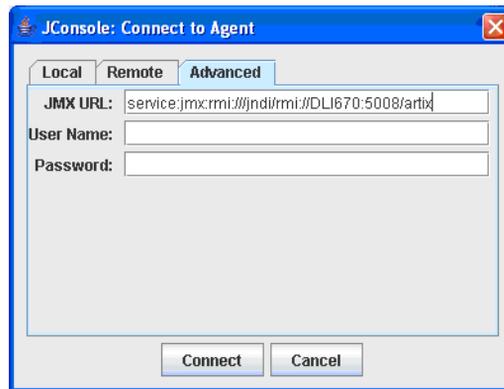
When the server runs, a file named `connector` is created in the `...demos\advanced\wsm\etc\` directory.

**Start a JMX console**

You can start any JMX console. For example, to start JConsole, execute the following command:

```
%jdk1.5_home%\bin\jconsole.exe
```

This displays the **JConsole: Connect to Agent** dialog, as shown in [Figure 24](#).

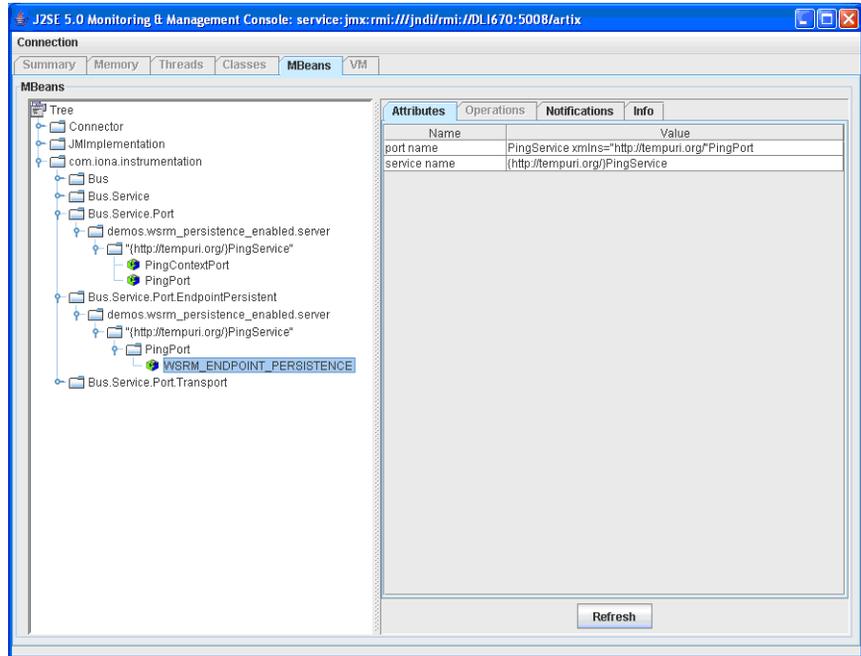


**Figure 24:** Connecting to a JMX Agent

Copy the contents of the `connector` file into the JMX URL field, and click **Connect**. This displays the **J2SE 5.0 Monitoring and Management Console**, as shown in [Figure 25](#).

### View WS-RM persistence enabled endpoints

You can view a WS-RM persistence enabled endpoint in the **MBeans** tab of the JMX console, as shown in [Figure 25](#):



**Figure 25:** WS-RM Persistence Enabled Endpoint

In this example, PingPort is a WS-RM persistence enabled port. You can view the port and service name in the **Attributes** tab on the right of the console.

## View messages in the WS-RM persistence database

To view messages in the WS-RM persistence database, perform the following steps:

1. Edit the client code in

...\demos\advanced\wsrm\cxx\client\PingClientSample.cxx as follows:

```
int
run_persistence_client(
    int argc,
    char* argv[]
)
...
for (int i=0; i < 10; i++)
{
    cout << "Invoking PingOneway " << i << endl;
    PingType param1;
    param1.setText("PingOneway message from client");
    client1.PingOneway(param1);
    cout << i << " PingOneway invoked" << endl;
}
...
```

This adds a loop to the client that invokes the server 10 times in order to easily view messages in WS-RM persistence database.

2. Start the client. For example, go to the demos\advanced\wsrm\bin directory, and run the following command:

```
run_cxx_client_persistence.bat
```

3. You can view the attributes for the client in the JMX console, as shown in Figure 26. The client name consists of the `sequence_guid` string and a sequence ID.

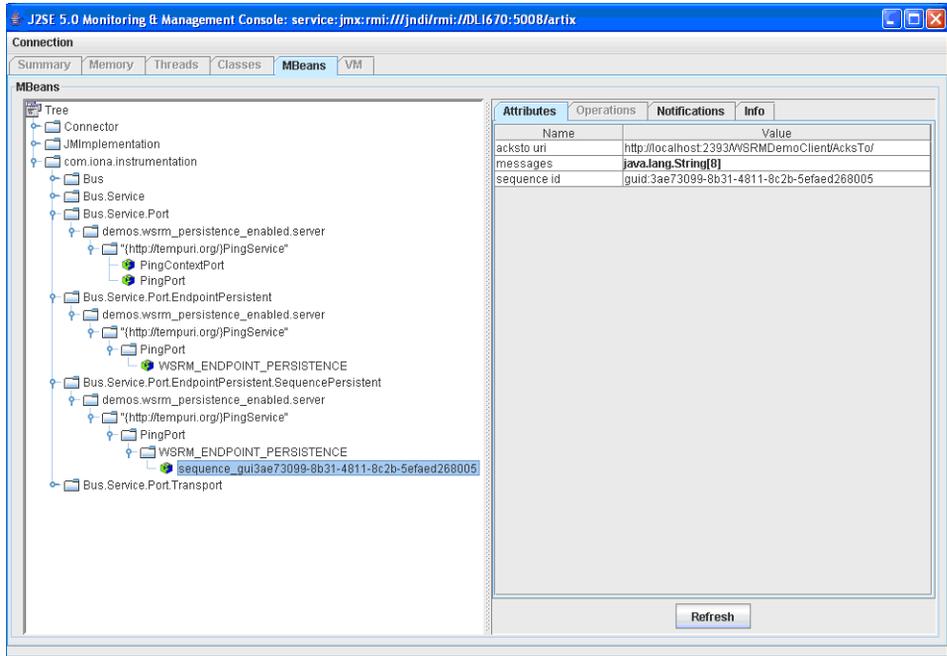
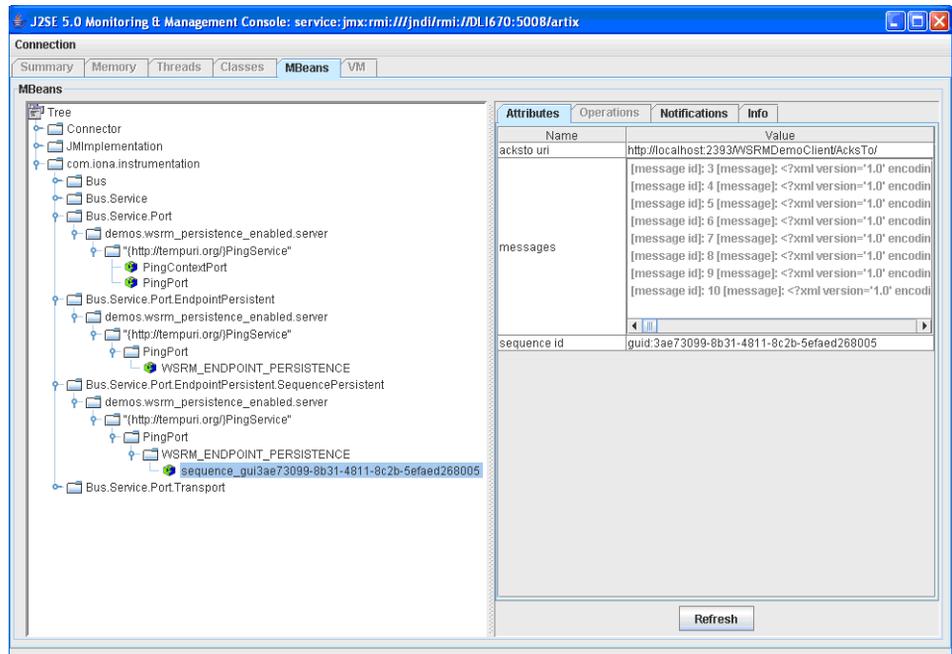


Figure 26: WS-RM Persistence Enabled Client

- You can view all the messages in WS-RM persistence database by clicking in the **Attributes** tab on the right of the console, as shown in [Figure 27](#). Each message consists of a message ID and a SOAP message.



**Figure 27:** WS-RM Persistence Enabled Messages

You can click the **Refresh** button to view the current messages in WS-RM persistence database.



# Index

## A

- acksto uri 69
- address 31
- arguments 18
- Attributes tab 72
- averageResponseTime 25

## B

- bus
  - attributes 18
  - ObjectName 17
- bus\_management 36

## C

- CompositeData 32
- connector.url 59
- custom JMX MBeans 14

## G

- getLoggingLevel 19

## H

- HTTP adaptor 63

## I

- identifier 18
- interceptors 31, 61

## J

- J2SE 5.0 Monitoring and Management Console 71
- Java Management Extensions 11, 35
- JConsole 59, 71
  - Connect to Agent 71
- JMX 11, 35
- JMX HTTP adaptor 63
- JMX Remote 15
- JMXServiceURL 36

## L

- locator

- managed attributes 28
- logging
  - levels 19
  - subsystems 19

## M

- Managed Beans 12
- management consoles 39
- MBeans 12
- MBeanServer 12, 68
- MBeanServerConnection 14
- MBeans tab 72
- MC4J 40
- messages 69

## P

- persistence endpoint 68
- plugins:artix:db
  - home 70
- plugins:bus\_management:connector
  - enabled 70
- plugins:bus\_management:connector:enabled 36
- plugins:bus\_management:connector:port 70
- plugins:bus\_management:connector:registry:required 37
- plugins:bus\_management:connector:url
  - file 70
- plugins:bus\_management:connector:url:file 37
- plugins:bus\_management:connector:url:publish 37
- plugins:bus\_management:enabled 36, 70
- plugins:bus\_management:http\_adaptor:enabled 63
- plugins:bus\_management:http\_adaptor:port 63
- port
  - name 31
  - ObjectName 31
- port name 68
- ports 24

## R

- registeredEndpoints 28, 30
- registeredNodeErrors 28
- registeredServices 28, 30

## INDEX

- remote access port 37
- remote JMX clients 36
- requestsOneway 25
- requestsSinceLastCheck 25
- requestsTotal 25
- RMEndpointPersistentStore 68
- RMI Connector 36
- RMSequencePersistentStore 68
- runtime MBeans 14

### S

- scope 18
- sequence id 69
- service
  - attributes 24
  - managed components 23
  - methods 26
  - name 24
  - ObjectName 24
- serviceCounters 24
- serviceGroups 30

- serviceLookupErrors 28
- serviceLookups 28
- service name 68
- services 18
- serviceSessions 30
- servicesMonitoring 18
- session manager
  - managed attributes 30
- setLoggingLevel 19
- setLoggingLevelPropagate 19
- state 24

### T

- TabularData 32
- timeSinceLastCheck 25
- totalErrors 25
- transport 32

### W

- WS-RM persistence 68