Artix™ ESB

Managing Artix Solutions with JMX, Java Runtime

Version 5.1, December 2007

Making Software Work Together™

Updated: January 25, 2008

Contents

List of Figures 5

List of Tables 7

Preface 9
- What is Covered in this Book 9
- Who Should Read this Book 9
- How to Use this Book 9
- The Artix Documentation Library 9

Chapter 1 Monitoring and Managing an Artix Java Runtime 11
- Introduction 12
- Managed Runtime Components 16

Chapter 2 Instrumenting Artix Java Services 19
- Using the JMX MBean Interfaces 20
- Using the Artix ManagedComponent interface 23

Chapter 3 Configuring JMX in an Artix Java Runtime 33
- Artix JMX Configuration 34

Chapter 4 Managing Java Services with JMX Consoles 39
- Managing Artix Services with JConsole 40

Index 47

List of Figures

Figure 1: Artix Java Runtime JMX Architecture 13
Figure 2: Managed Bus Info in JConsole 41
Figure 3: Managed Bus Operation in JConsole 42
Figure 4: Managed Endpoint Attributes in JConsole 43
Figure 5: Managed Endpoint Operations in JConsole 44
Figure 6: Custom MBean Attributes in JConsole 45
Figure 7: Custom MBean Operations in JConsole 46

List of Tables

Table 1: Managed Bus Methods 16
Table 2: Managed Endpoint Attributes 16
Table 3: Managed Endpoint Operations 17
Table 4: JDK 5.0 JMX Annotations 24
Table 5: JMX Annotation Metadata 24

Preface

What is Covered in this Book

Managing Artix Solutions with JMX, Java Runtime explains how to monitor and manage Artix services in an Artix Java runtime environment using Java Management Extensions (JMX). It applies to Artix Java services written using the Java API for XML-Based Web Services (JAX-WS).

Who Should Read this Book

The main audience of *Managing Artix Solutions with JMX, Java Runtime* is Artix Java developers and system administrators.

How to Use this Book

This book includes the following:

- Chapter 1 introduces the JMX features supported by the Artix Java runtime, and describes the Artix components that can be managed using JMX.
- Chapter 2 explains how to instrument your Artix Java services using custom MBeans.
- Chapter 3 explains how to configure an Artix Java runtime for JMX.
- Chapter 4 explains how to manage and monitor Artix Java services using JMX consoles.

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.

CHAPTER 1

Monitoring and Managing an Artix Java Runtime

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

In this chapter

This chapter discusses the following topics:

Introduction	page 12
Managed Runtime Components	page 16

Introduction

Overview

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

- View service status
- View a service endpoint's address
- Stop or start a service
- Shutdown an Artix Java bus

How it works

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

Artix Java runtime components can be exposed as JMX MBeans out-of-the-box (for example, Artix Java service endpoints and the Artix Java bus). In addition, the Artix Java runtime supports the registration of custom MBeans. Java developers can create their own MBeans and register them either with their JMX MBean server of choice, or with a default MBean server created by Artix (see "Relationship between runtime and custom MBeans" on page 14).

Artix Java services can be monitored and managed by any JMX-compliant client application (for example JConsole). Figure 1 shows an overview of how the various components interact.

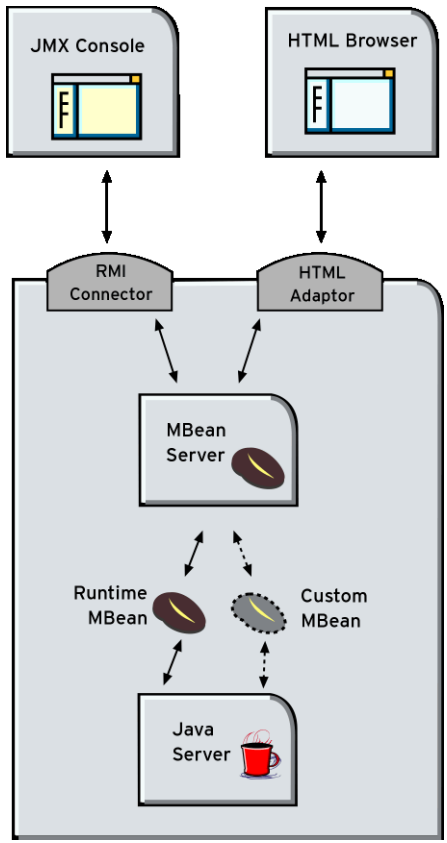

Figure 1: *Artix Java Runtime JMX Architecture*

The custom MBeans shown in Figure 1 are optional components that can be implemented as required (for details, see Chapter 2).

Enabling JMX

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

For details on how to configure JMX support in Artix applications, see Chapter 3.

What can be managed

Artix JAX-WS servers can have the following runtime components exposed as JMX MBeans:

- Artix Java bus
- Service endpoint

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

Naming conventions

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 `org.apache.cfx` as their domain name when creating managed components.

Relationship between runtime and custom MBeans

The Artix Java runtime instrumentation provides an out-of-the-box JMX view of JAX-WS services. Java developers can also create custom JMX MBeans to manage Artix Java components such as service endpoint attributes and operations.

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, 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 more details, see Chapter 2.

Further information

For further information, see the following:

JMX

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

JMX Remote

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

Dynamic MBeans

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

MBeanServer

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

Managed Runtime Components

Overview

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

- Shutdown an Artix Java bus
- View service status
- View a service endpoint's address.
- Stop or start a service

Artix bus operations

The following Artix Java bus method can be accessed using any JMX console or client:

Table 1: *Managed Bus Methods*

Name	Description	Parameters
`shutdown()`	Shuts down the current Artix Java bus.	`boolean`

Endpoint attributes

The following Artix service endpoint attributes can be managed by any JMX console or client:

Table 2: *Managed Endpoint Attributes*

Name	Description	Type
`Address`	Endpoint address (for example, `http://localhost:9000/SoapContext/SoapPort`).	`String`
`State`	Current service state manipulated by stop and start methods. Possible values are `STARTED` or `STOPPED`.	`String`
`TransportID`	Endpoint transport ID (for example, `http://schemas.xmlsoap.org/soap/http` for the HTTP transport).	`String`

Endpoint operations

The following Artix service endpoint operations can be managed by any JMX console or client:

Table 3: *Managed Endpoint Operations*

Name	Description	Parameters	Return Type
`getAddress()`	Get the service address (for example, `http://localhost:9000/SoapContext/SoapPort`).	None	`String`
`getState()`	Get the current service state. Possible values are `STARTED` or `STOPPED`.	None	`String`
`getTransportID()`	Get the service transport ID (for example, `http://localhost:9000/SoapContext/SoapPort`).	None	`String`
`start()`	Activate a service.	None	`Void`
`stop()`	Deactivate a service.	None	`Void`

For examples of operations and attributes displayed in a JMX console, see Chapter 4

MBeanInfo

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

CHAPTER 2

Instrumenting Artix Java Services

This chapter explains how to instrument your Artix Java services using custom MBeans. There are two different approaches. You can use either the JMX MBean interfaces or the Artix ManagedComponent interface. This applies to applications written using the Java API for XML-Based Web Services (JAX-WS).

In this chapter

This chapter discusses the following topics:

Using the JMX MBean Interfaces	page 20
Using the Artix ManagedComponent interface	page 23

Using the JMX MBean Interfaces

Overview

This section shows how to implement a JMX MBean interface and register it with the Artix MBean server.

The Artix MBean server can be accessed through the Artix Java bus and enables the registration of custom MBeans. You can instrument your service implementation by developing a custom MBean using one of the JMX MBean interfaces and registering it with the Artix MBean server. Your custom instrumentation can be accessed using the same JMX connection as the Artix internal components used by your service.

Creating your custom MBean

When using the JMX APIs to instrument your service implementation, follow the design methodology laid out by the JMX specification. This involves the following steps:

1. Decide what type of MBean you wish to use.
   - Standard MBeans expose a management interface defined at development time.
   - Dynamic MBeans expose their management interface at runtime.
2. Create the MBean interface to expose the properties and operations used to manager your service implementation.
   - Standard MBeans use the `MBean` interface.
   - Dynamic MBeans use the `DynamicMBean` interface.
3. Implement the MBean class.

Example 1 shows the interface for a standard MBean.

Example 1: *Standard MBean Interface*

```
public interface ServerNameMBean
{
   String getServiceName();
   String getAddress();
}
```

Example 2 shows a class that implements the MBean defined in Example 2.

Example 2: *Standard MBean Implementation Class*

```
public class ServerName{

   String getServiceName()
   {
      return "mySOAPservice";
   }

   String getAddress()
   {
      return "myServiceAddress";
   }
}
```

Registering the MBean

To expose your MBean in a JMX management console, it must be registered with the Artix MBean server. The Artix MBean server can be accessed through the Artix Java bus. Typically, this happens when your service is initialized.

To register a custom MBean, perform the following steps:

1. Instantiate your custom MBean.
2. Get an instance of the bus.
3. Get the Artix MBean server from the bus.
4. Create an `ObjectName` for your MBean.

Note: It is recommended that your MBeans follow the "Naming conventions" on page 14. However, you can choose any naming scheme.

5. Register your MBean server using the server's `registerMBean()` method.

Example 3 shows the steps for registering a custom MBean with the Artix MBean server.

Example 3: *Registering a Custom MBean*

```
import javax.management.MBeanServer;
import javax.management.ObjectName;
import org.apache.cfx.Bus;

...

//Instantiate the MBean
ServerName sName = new ServerName();

//Get the MBean server from the bus
InstrumentationManager im =
   BusFactory.getDefaultBus().getExtension(org.apache.cxf.management.
   InstrumentationManager.class);
MBeanServer server = im.getMBeanServer();

//Create ObjectName for the MBean
ObjectName name = new
   ObjectName(my..instrumentation:type=CustomMBean,Bus="+
   bus.getBusID() + name="ServerNameMBean");

//Register the MBean
server.registerMBean(sName, name);
```

Alternatively, you do not have to register the MBean directly with the `MBeanServer`. You can also use the `register(Object, ObjectName)` utility method on the `InstrumentationManager` to register a `StandardMBean`.

Further information

For further information, see the following:

ObjectName

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

MBeanServer

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

JMX specifications

http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/docs.jsp.

Using the Artix ManagedComponent interface

Overview

If you do not wish to use the JMX APIs to add instrumentation to your service, you can use the Artix `ManagedComponent` interface. This interface wraps the JMX subsystem in an Artix-specific API. You do not need to access the Artix MBean server to register your managed components because the Artix wrappers take care of this for you. This approach uses JMX annotations to specify which methods and attributes are exposed.

Adding custom instrumentation using the Artix `ManagedComponent` interface involves the following steps:

1. Write an instrumentation class that implements the `org.apache.cxf.management.ManagedComponent` interface.
2. When your service is starting up, activate your `ManagedComponent` object by instantiating it and registering it with the `InstrumentationManager`.
3. When your service is shutting down, deactivate your `ManagedComponent` by unregistering it and cleaning it up.

This section shows how to implement the Artix `ManagedComponent` interface using JMX annotations. It uses example code from the Artix `jmx` sample application:

```
ArtixInstallDir\Version\java\samples\advanced\management\jmx
```

Writing the instrumentation class

Like an MBean, an Artix instrumentation class is responsible for providing access to the attributes that you wish to track, and implementing any management operations that you want to expose. Unlike an MBean, an Artix instrumentation class does not implement a user-defined interface. Instead, it implements the Artix-defined `ManagedComponent` interface, and defines the operations required to expose your managed attributes and operations.

JMX annotations

The Artix management API uses JDK 5.0 annotations to create an object that implements the `ModelMBeanInfo` interface. This reads the Artix annotations to identify the attributes and operations that are exposed. It then uses this information to create a `ModelMBean` and registers it with the MBean server.

Table 4 lists the JDK 5.0 annotations that can be used when implementing your instrumentation class.

Table 4: *JDK 5.0 JMX Annotations*

Annotation	Type	Description
`@ManagedResource`	Class	Marks all instances of a class as a JMX managed resource.
`@ManagedOperation`	Method	Marks a method as a JMX operation.
`@ManagedAttribute`	Method	Marks a getter or a setter as one half of a JMX attribute.
`@ManagedNotification`	Method	Marks a JMX notification issued by an MBean.
`@ManagedOperationParameter` `@ManagedOperationParameters`	Method	Describes the parameters of a managed operation.

Annotation parameters

Table 5 lists parameters that can be supplied to the JMX annotations.

Table 5: *JMX Annotation Metadata*

Parameter	Annotation	Description
`componentName`	`@ManagedResource`	Specifies the name of the managed resource.

`description`	`@ManagedResource`
	`@ManagedAttribute`
	`@ManagedOperation`
	`ManagedNotification`
	`@ManagedOperationParameter`
		Specifies a user-friendly description of the resource, attribute, or operation.

Table 5: *JMX Annotation Metadata*

Parameter	Annotation	Description
currencyTimeLimit	@ManagedResource @ManagedAttribute	Specifies how long in seconds a cached value is valid (0 for never, =0 for always, or >0).
defaultValue	@ManagedAttribute	Specifies a default cached value.
log	@ManagedResource @ManagedNotification	Enables logging. Specify true to log all notifications or false to log no notifications.
logFile	@ManagedResource @ManagedNotification	Specifies the a fully qualified filename to log events to.
persistPolicy	@ManagedResource	Specifies the persistence policy. Values are: OnUpdate OnTimer NoMoreOftenThan Always Never
persistPeriod	@ManagedResource	Specifies the frequency of the persist cycle in seconds for the OnTime and NoMoreOftenThan policies.
persistLocation	@ManagedResource	Specifies the filename in which the MBean should be persisted.
persistName	@ManagedResource	Specifies the name that is persisted.
name	@ManagedOperationParameter	Specifies the display name of an operation parameter.
index	@ManagedOperationParameter	Specifies the index of an operation parameter.

Table 5: *JMX Annotation Metadata*

Parameter	Annotation	Description
`currencyTimeLimit`	`@ManagedResource` `@ManagedAttribute`	Specifies how long in seconds a cached value is valid (`0` for never, `=0` for always, or `>0`).
`defaultValue`	`@ManagedAttribute`	Specifies a default cached value.
`log`	`@ManagedResource` `@ManagedNotification`	Enables logging. Specify `true` to log all notifications or `false` to log no notifications.
`logFile`	`@ManagedResource` `@ManagedNotification`	Specifies the a fully qualified filename to log events to.

`persistPolicy`	`@ManagedResource`	Specifies the persistence policy. Values are:
		`OnUpdate`
		`OnTimer`
		`NoMoreOftenThan`
		`Always`
		`Never`

`persistPeriod`	`@ManagedResource`	Specifies the frequency of the persist cycle in seconds for the `OnTime` and `NoMoreOftenThan` policies.
`persistLocation`	`@ManagedResource`	Specifies the filename in which the MBean should be persisted.
`persistName`	`@ManagedResource`	Specifies the name that is persisted.
`name`	`@ManagedOperationParameter`	Specifies the display name of an operation parameter.
`index`	`@ManagedOperationParameter`	Specifies the index of an operation parameter.

Adding annotations

When implementing your custom MBean's instrumentation class using the `ManagedComponent` interface, you should annotate the class with the `@ManagedResource` attribute. Any management operation that you wish to expose in your class should be annotated with the `@ManagedOperation` attribute. Any attributes that you wish to expose should have their getter and setter methods annotated with the `@ManagedAttribute` attribute. If you want to make an attribute read-only or write-only, you can omit the annotation from either its setter method or its getter method.

Example 4 shows an custom MBean class taken from the Artix `jmx` sample application.

Example 4: *Example Artix Instrumentation Class*

```
// GreaterImpl.java

package .hw.server;

import java.util.logging.Logger;

import javax.management.JMException;
import javax.management.ObjectName;

import org.apache.cxf.management.ManagedComponent;
import org.apache.cxf.management.annotation.ManagedAttribute;
import org.apache.cxf.management.annotation.ManagedOperation;
import org.apache.cxf.management.annotation.ManagedResource;

import org.apache.hello_world_soap_http.Greeter;
import org.apache.hello_world_soap_http.PingMeFault;
import org.apache.hello_world_soap_http.types.FaultDetail;

@ManagedResource(componentName = "GreeterImpl",
                 description = "A typical Greeter implementation.")

@javax.jws.WebService(portName = "SoapPort", serviceName = "SOAPService",
                      targetNamespace = "http://apache.org/hello_world_soap_http",
                      endpointInterface = "org.apache.hello_world_soap_http.Greeter")
```

```
public class GreeterImpl implements Greeter, ManagedComponent {

    private static final Logger LOG = Logger.getLogger(GreeterImpl.class.getPackage().getName());
    private int count;

    /* (non-Javadoc)
     * @see org.apache.hello_world_soap_http.Greeter#greetMe(java.lang.String)
     */
    public String greetMe(String me) {
        LOG.info("Executing operation greetMe");
        count++;
        System.out.println("Executing operation greetMe");
        System.out.println("Message received: " + me + "\n");
        return "Hello " + me;
    }

/* (non-Javadoc)
     * @see org.apache.hello_world_soap_http.Greeter#greetMeOneWay(java.lang.String)
     */
    public void greetMeOneWay(String me) {
        LOG.info("Executing operation greetMeOneWay");
        count++;
        System.out.println("Executing operation greetMeOneWay\n");
        System.out.println("Hello there " + me);
    }

 /* (non-Javadoc)
     * @see org.apache.hello_world_soap_http.Greeter#sayHi()
     */
    public String sayHi() {
        LOG.info("Executing operation sayHi");
        count++;
        System.out.println("Executing operation sayHi\n");
        return "Bonjour";
    }

    public void pingMe() throws PingMeFault {
        count++;
        FaultDetail faultDetail = new FaultDetail();
        faultDetail.setMajor((short)2);
        faultDetail.setMinor((short)1);
        LOG.info("Executing operation pingMe, throwing PingMeFault exception");
        System.out.println("Executing operation pingMe, throwing PingMeFault exception\n");
        throw new PingMeFault("PingMeFault raised by server", faultDetail);
    }
```

```
   public ObjectName getObjectName() throws JMException {
        return new ObjectName("samples:name=GreeterImpl");
    }

    @ManagedAttribute(description = "The Count Attribute", currencyTimeLimit = 15)
    public int getCount() {
        return count;
    }

    @ManagedOperation(description = "Add Two Numbers Together")
    public void resetCount() {
        count = 0;
    }
}
```

Registering your custom MBean

To make your custom instrumentation available to JMX management consoles, you must create an instance of your custom class and register it with the bus. This handles the creation of the `ModelMBean` to represent your custom MBean. It also handles the registration of the MBean with the MBean server.

To activate your custom MBean, do the following:

1. Get the current Artix Java bus instance.
2. Get the `InstrumentationManager` from the bus using `bus.getExtension()`.
3. Create an instance of your instrumentation class.
4. Register your custom MBean instance with the `InstrumentationManager`.

Example 5 shows these steps in the sample server code for activating a custom MBean.

Example 5: *Example Server Code*

```
// Server.java

package .hw.server;

import javax.xml.ws.Endpoint;
import org.apache.cxf.Bus;
import org.apache.cxf.bus.spring.SpringBusFactory;
import org.apache.cxf.management.InstrumentationManager;
import org.apache.cxf.management.ManagedComponent;

public class Server {

    protected Server() throws Exception {
        System.out.println("Starting Server");

        SpringBusFactory factory = new SpringBusFactory();
        Bus bus = factory.createBus();
       InstrumentationManager im = bus.getExtension(InstrumentationManager.class);
        ManagedComponent component = new GreeterImpl();
        im.register(component);
        String address = "http://localhost:9000/SoapContext/SoapPort";
        Endpoint.publish(address, component);
    }

    public static void main(String args[]) throws Exception {
        new Server();
        System.out.println("Server ready...");

        Thread.sleep(10 * 60 * 1000);
        System.out.println("Server exiting");
        System.exit(0);
    }
}
```

Deactivating your custom instrumentation

You can explicitly tell the bus to remove the `ModelMBean` created for your instrumentation using the `InstrumentationManager.unregister()` method. This method removes the MBean from the Artix MBean server, destroys the associated `ModelMBean`, and frees up any resources used by it.

In the Artix `jmx` sample application MBean is not explicitly unregistered, but is destroyed when the server process is destroyed.

Further information

For further information, see the following:

ModelMBeanInfo

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

CHAPTER 3

Configuring JMX in an Artix Java Runtime

This chapter explains how to configure an Artix Java runtime for JMX using the XML-based Spring Framework.

In this chapter

This chapter discusses the following topic:

Artix JMX Configuration	page 34

Artix JMX Configuration

Overview

JMX support in an Artix Java runtime is enabled using configuration settings only. You do not need to write any Artix code to enable JMX support in the Artix runtime.

JMX is not configured by default. When configured, you can use any third party console that supports JMX Remote to monitor and manage Artix services. This section shows the Artix configuration settings required to enable JMX monitoring of the Artix runtime, and access for remote JMX clients.

Configuring Artix JMX features

The Artix Java configuration mechanism uses the XML-based Spring Framework. In the Artix `jmx` sample application, the JMX support is configured using the `org.apache.cxf.management.jmx.InstrumentationManagerImpl` class. This class includes the following properties:

`bus`	Specifies the name of the Artix bus. The name of the Artix Java bus is `cfx`.
`enabled`	Specifies whether JMX monitoring and management is enabled. Possible values are `true` or `false`. Specifying `true` enables remote JMX clients to access runtime and custom MBeans.
`JMXServiceURL`	Specifies the connection URL for the JMX server. The default URL is: `service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi/server`

Example 6 shows the JMX configuration settings taken from the `managed_server.xml` file in the Artix `jmx` sample application:

```
ArtixInstallDir\Version\java\samples\advanced\management\jmx
```

Example 6: *Contents of managed_server.xml*

```
?xml version="1.0" encoding="UTF-8"?>
<!-- -->
<!-- Copyright (c) 1993-2007 IONA Technologies PLC. -->
<!--            All Rights Reserved. -->
<!-- -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:im="http://cxf.apache.org/management"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- InstrumetationManager's setting -->
    <bean id="InstrumentationManager"
   class="org.apache.cxf.management.jmx.InstrumentationManagerImpl">
        <property name="bus" ref="cxf" />
        <property name="enabled" value="true" />
        <property name="JMXServiceURL"
   value="service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi/server" />
    </bean>
</beans>
```

The `InstrumentationManagerImpl` class extends `JMXConnectorPolicyType`. For more details, see "Artix JMX schema" on page 36.

Accessing Artix Java configuration

You can make your configuration available to the Artix Java runtime in one of the following ways:

- Use one of the following command-line flags to point to your XML configuration file:
  - `-Dcxf.config.file=<myCfgResource>`
  - `-Dcxf.config.file.url=<myCfgURL>`

  This enables you to save your XML configuration file anywhere on your system and avoid adding it to your `CLASSPATH`. This is the approach used in most of the Artix Java samples (for example, `managed_server.xml`).

- Specify the XML configuration file on your `CLASSPATH`.
- Programmatically, by creating a bus and passing the configuration file location as either a URL or string, as follows:

```
(new SpringBusFactory()).createBus(URL myCfgURL)
(new SpringBusFactory()).createBus(String myCfgResource)
```

Artix JMX schema

The complete schema for configuring JMX in an Artix Java runtime is contained in the `instrumentation.xsd` file shown in Example 7:

Example 7: *JMX Configuration Schema*

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema targetNamespace="http://cxf.apache.org/management"
    xmlns:tns="http://cxf.apache.org/management"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified"
    jaxb:version="2.0">

    <xs:complexType name="JMXConnectorPolicyType">
        <xs:attribute name="Enabled" type="xs:boolean" use="required" />
        <xs:attribute name="Threaded" type="xs:boolean" use="required" />
        <xs:attribute name="Daemon" type="xs:boolean" use="required" />
        <xs:attribute name="JMXServiceURL" type="xs:string"
   default="service:jmx:rmi:///jndi/rmi://localhost:9913/jmxrmi"/>
    </xs:complexType>

    <xs:element name="JMXConnectorPolicy" type="tns:JMXConnectorPolicyType"/>

</xs:schema>
```

These attributes are explained as follows:

`Enabled`	Specifies whether the JMX infrastructure is available to the Artix Java runtime. JMX is disabled by default. The MBean server is an unnecessary overhead if you do not require JMX.
`Threaded`	Specifies whether the JMX server starts in a separate thread.
`Daemon`	If the JMX server is running in a separate thread, specifies whether it is run as daemon thread

`JMXServiceURL` Specifies the `JMXServiceURL` to connect to remotely. Remote access is performed using JMX Remote, using an RMI Connector on a default port of `1099`. Use the following JNDI-based `JMXServiceURL` to connect remotely:

```
service:jmx:rmi:///jndi/rmi://localhost:1099
/jmxrmi/server
```

Further information

For more information, see the following:

Artix Java configuration

- Configuring and Deploying Artix Solutions, Java Runtime
- Artix Configuration Reference, Java Runtime

Spring Framework

www.springframework.org

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

CHAPTER 4

Managing Java Services with JMX Consoles

You can use any third-party management console that supports JMX Remote to monitor and manage Artix services (for example, JConsole or MC4J).

In this chapter

This chapter discusses the following topics:

Managing Artix Services with JConsole	page 40

Managing Artix Services with JConsole

Overview

The JConsole management console is provided with JDK 1.5 to monitor and manage Artix Java applications. For convenience, Artix installs JConsole, which can be run out-of-the-box with the Artix `jmx` sample application:

```
ArtixInstallDir\Version\java\samples\advanced\management\jmx
```

Using JConsole with Artix

Artix runtime MBeans can be accessed remotely using JMXRemote. This means that any management console that supports JMXRemote can be used to monitor and manage Artix-enabled applications.

To view the management information for a deployed Artix-enabled application using JConsole, perform the following steps:

1. Launch the JConsole application using the following command:

   ```
   ArtixInstallDir/java/bin/jmx_console_start
   ```

   Alternatively, you can use:

   ```
   JDK_HOME/bin/jconsole.
   ```

2. Select the **Advanced** tab.
3. Enter the URL of your Artix MBean server in the `JMXServiceURL` field. This will either be the default Artix `JMXServiceURL` (`service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi/server`), or the value specified by the `JMXServiceURL` property in your application's Spring configuration file.

Note: When running the `jmx` sample application, steps 2 and 3 are not necessary.

Managing runtime components

JConsole displays managed Artix runtime components in a hierarchical tree, as shown in Figure 2. This shows the MBean information displayed for the managed bus component (for example, the MBean name and Java class).

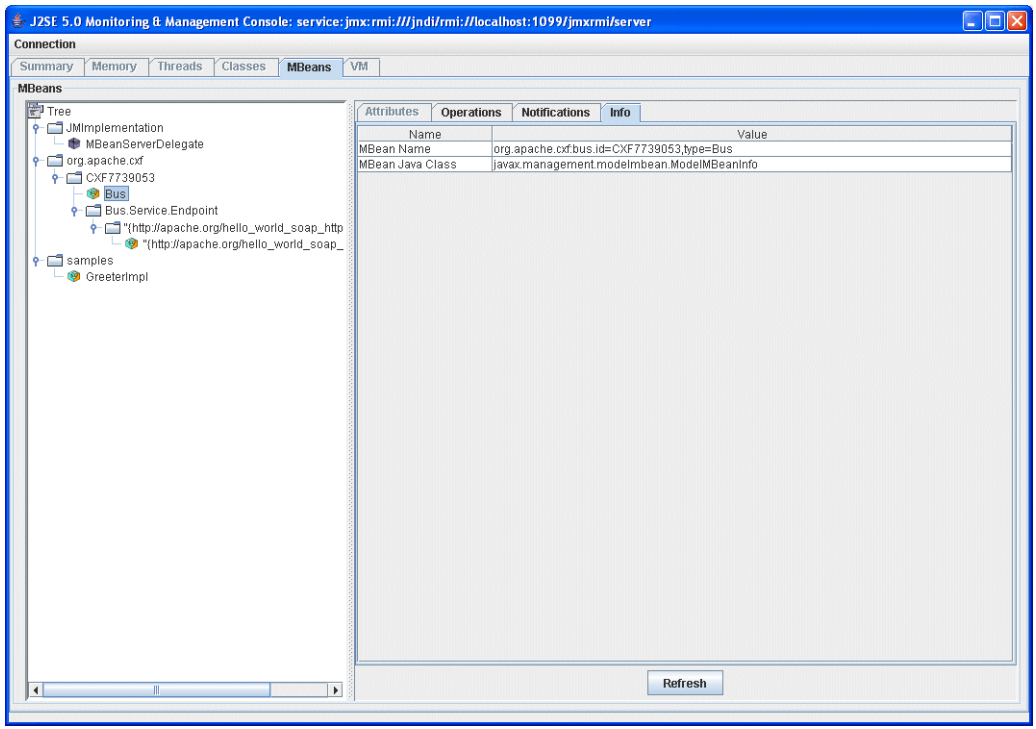

Figure 2: *Managed Bus Info in JConsole*

Figure 3 shows the `shutdown()` operation for the managed bus displayed in JConsole.

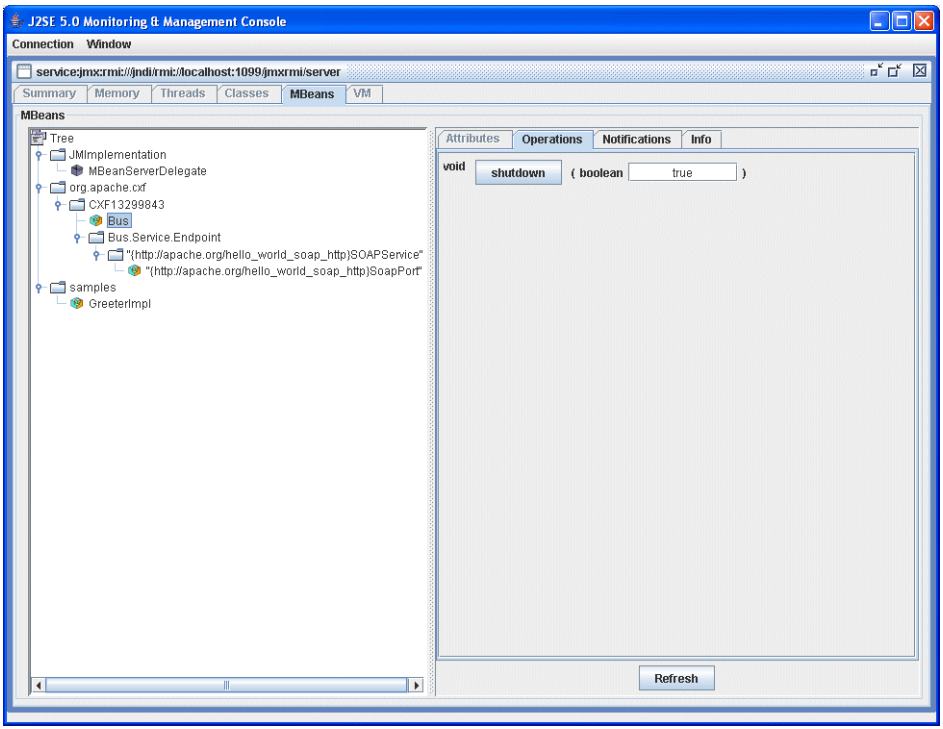

Figure 3: *Managed Bus Operation in JConsole*

Figure 4 shows the attributes displayed for a managed service endpoint displayed in JConsole.

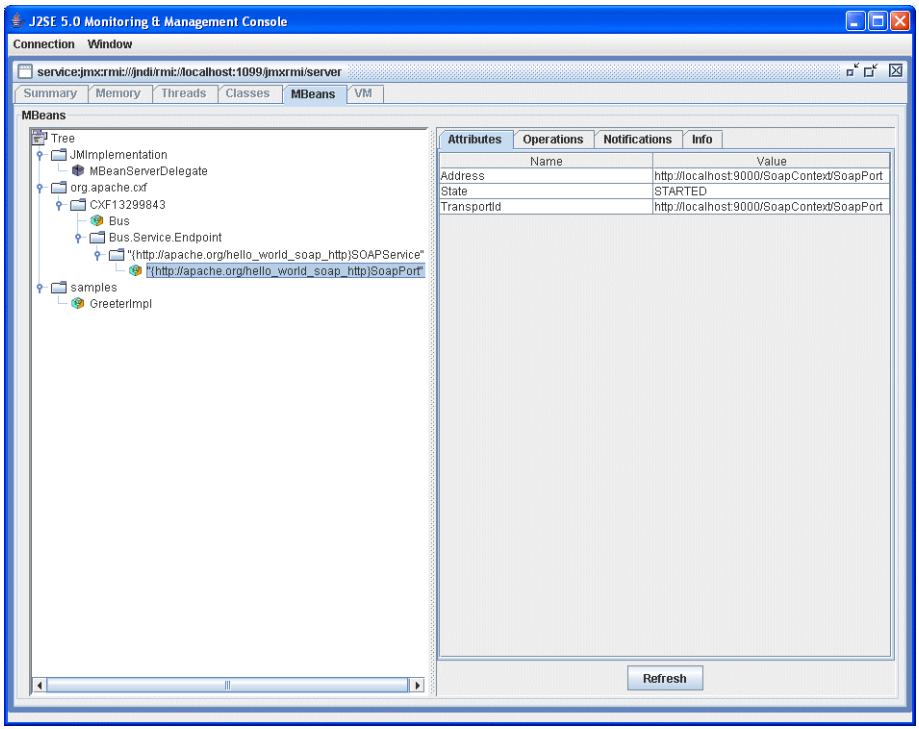

Figure 4: *Managed Endpoint Attributes in JConsole*

Figure 5 shows the operations displayed for managed a service endpoint displayed in JConsole.

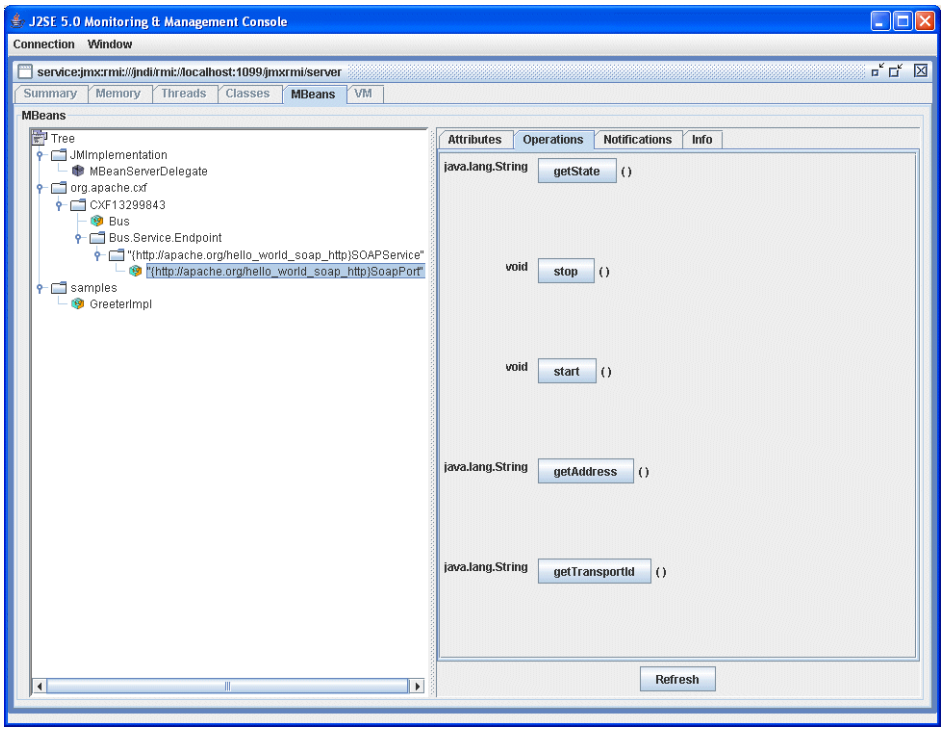

Figure 5: *Managed Endpoint Operations in JConsole*

Managing custom MBeans

Figure 6 shows the attributes displayed for the sample custom MBean displayed in JConsole.

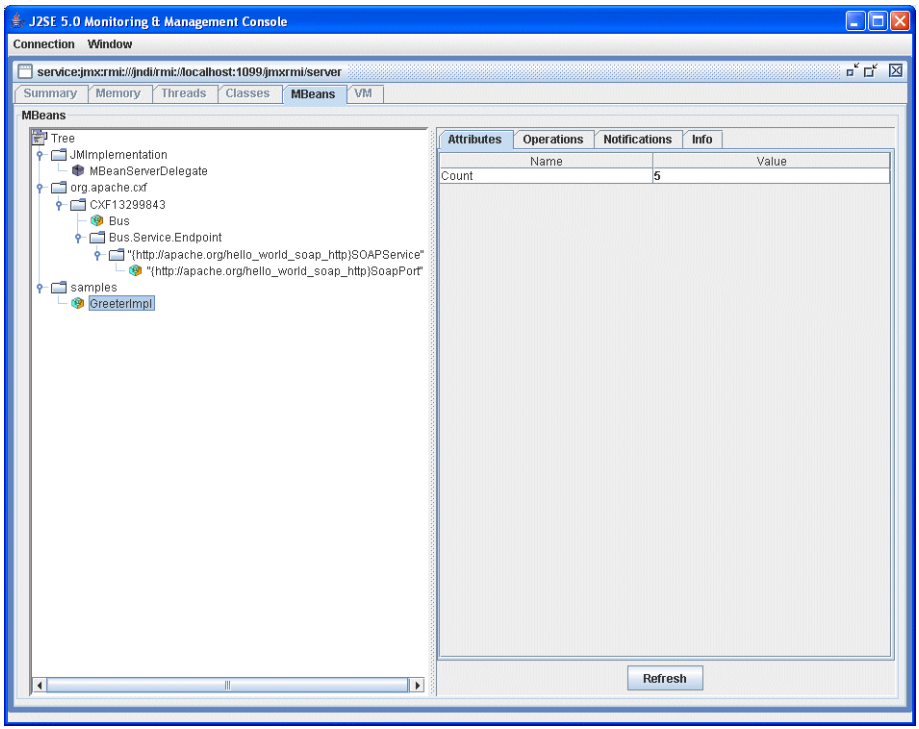

Figure 6: *Custom MBean Attributes in JConsole*

Figure 5 shows the operations displayed for the sample custom MBean displayed in JConsole.

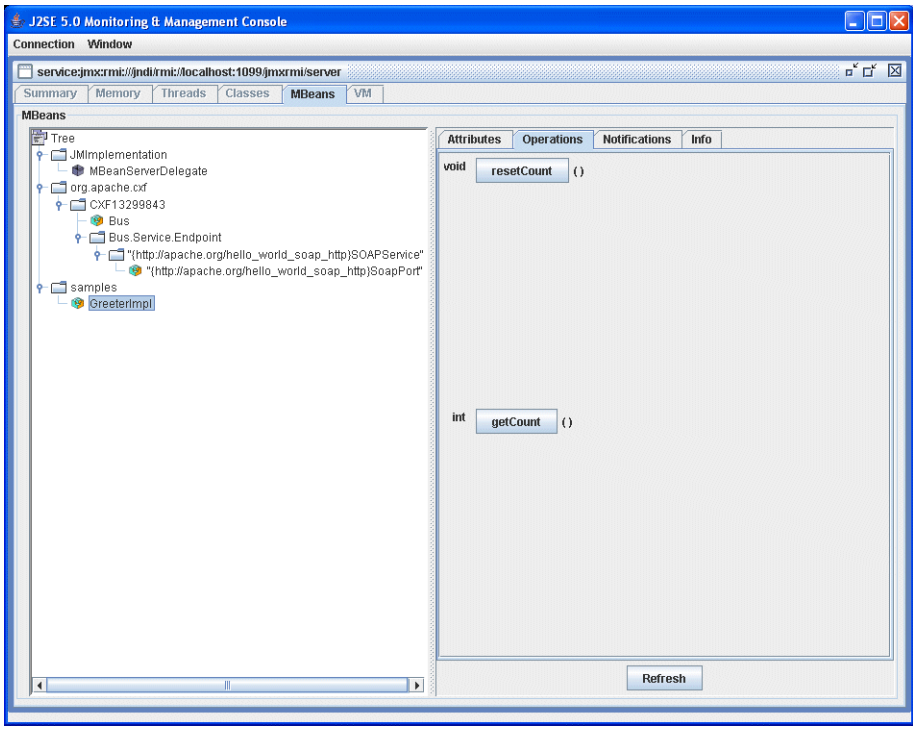

Figure 7: *Custom MBean Operations in JConsole*

Further information

For detailed information on Artix runtime attributes and operations see "Managed Runtime Components" on page 16.

For more information on using JConsole, see the following:

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

Index

Symbols
@ManagedAttribute 24, 27
@ManagedNotification 24
@ManagedOperation 24, 27
@ManagedOperationParameter 24
@ManagedOperationParameters 24
@ManagedResource 24, 27

A
Address 16
Advanced tab 40
annotations 24

B
bus 34
bus.get.Extension() 29

C
componentName 24
createBus() 36
currencyTimeLimit 25, 26
custom JMX MBeans 14

D
Daemon 36
-Dcxf.config.file 35
defaultValue 25, 26
description 24
dynamic MBeans 20

E
enabled 34, 36
endpoint
  attributes 16
  operations 17

G
getAddress() 17
getState() 17
getTransportID() 17

I
index 25, 26
InstrumentationManager 22, 29
InstrumentationManagerImpl 34

J
Java API for XML-Based Web Services 9
Java Management Extensions 11
JAX-WS 9
JConsole 40
JMX 11
JMX annotations 24
jmx_console_start 40
JMX Remote 14, 34
JMXRemote 40
JMXServiceURL 34, 37, 40

L
log 25, 26
logFile 25, 26

M
Managed Beans 12
ManagedComponent 23
managed_server.xml 35
management consoles 39
MBeanInfo 17
MBean information 41
MBean Java class 41
MBean name 41
MBeans 12
MBeanServer 22
ModelMBean 24, 29, 31
ModelMBeanInfo 24

N
name 25, 26

O
ObjectName 21

P

persistLocation 25, 26
persistName 25, 26
persistPeriod 25, 26
persistPolicy 25, 26

R

register() 22
registerMBean() 21
remote JMX clients 34
RMI Connector 37
runtime MBeans 14

S

service
  attributes 16
  operations 17
shutdown() 16, 42
SpringBusFactory() 36
Spring Framework 34
StandardMBean 22
standard MBeans 20
start() 17
STARTED 16
State 16
stop() 17
STOPPED 16

T

Threaded 36
TransportID 16

U

unregister() 31