

JMX Accelerated Howto

Blaine Simpson

JMX Accelerated Howto

Blaine Simpson

\$Revision: 2036 \$

Copyright © 2007, 2008, 2009 Axis Data Management Corp.



[<http://admc.com/>]

This document may be freely served, duplicated, or distributed, as long as the contents are not modified. Usage of described software is covered by the licenses of the individual products.

Table of Contents

Introduction	vii
Available formats for this document	vii
License	vii
Purpose	viii
Support	viii
1. Required Software	1
2. JMX Architecture	2
Main Architectural Components	2
JMX Conformance Levels	4
3. JMX Features	5
4. JMX Resources	7
5. Introduction to MBeans	10
MBeans	10
MBean Types: Standard vs. Dynamic	11
Standard MBeans	11
Dynamic MBeans	11
6. Instrumentation	13
General MBean Coding Gotchas	13
Set up a JMX Dev Environment	14
Create a Standard MBean	14
Create a Dynamic MBean	20
Create a Model MBean	20
Create a Dynamic MBean by subclassing AbstractDynamicMBean	21
Wrappers for existing Java classes	21
Using AbstractDynamicMBean for MBeans that subclass something else	21
7. JMX Client Implementation	22
8. Distributed Services and Connectors	24
RMI Connector Details	27
JMXMP Connector Details	27
A Flexible JMXConnectorServer Agent	27
9. TLS	31
TLS Considerations	31
Encryption	31
Password Authentication	31
Certificates	31
Authenticating and Authorizing the Server to the Client	31
Authenticating and Authorizing the Client to the Server	31
TLS Examples	32
A few gotchas	35
10. Protocol Adaptors	36
11. Agent Implementation	37
A. Running MX4J's HttpAdaptor	38

List of Tables

1. Available formats of this document vii

List of Examples

8.1. ConnectorClient.java	25
8.2. ConnectorServerAgent.java	28

Introduction



Note

Significantly updated 07/2007. Text and instructions have been greatly simplified by assuming that you are using the default JMX implementation in Sun Java 1.5 (or later).

If you notice any mistakes in this document, please email the author at [blaine dot simpson simpson at admc dot com](mailto:blaine_dot_simpson_simpson@admc_dot_com) [mailto:[blaine dot simpson simpson at admc dot com?subject=jmx Howto](mailto:blaine_dot_simpson_simpson@admc_dot_com?subject=jmx%20Howto)] so that he can correct them. (We require you to convert the "and" and "dot"s with the corresponding punctuation, in order to thwart spammers). See the Support section below about any other issues.

Available formats for this document

This document is available in several formats.



Note

Standalone PDF readers (those that do not display inside of a web browser frame) can't resolve relative URLs. Therefore, if you are viewing this document with a standalone PDF reader, links to external files distributed with this document will not work.

You may be reading this document right now at <http://pub.admc.com/howtos>, or in a distribution somewhere else. I hereby call the document distribution from which you are reading this, your *current distro*.

<http://pub.admc.com/howtos> hosts the latest production versions of all available formats. If you want a different format of the same *version* of the document you are reading now, then you should try your current distro. If you want the latest production version, you should try <http://pub.admc.com/howtos>.

Sometimes, distributions other than <http://pub.admc.com/howtos> do not host all available formats. So, if you can't access the format that you want in your current distro, you have no choice but to use the newest production version at <http://pub.admc.com/howtos>.

Table 1. Available formats of this document

format	your distro	at http://pub.admc.com/howtos
Chunked HTML	index.html	http://pub.admc.com/howtos/jmx/
All-in-one HTML	jmx.html	http://pub.admc.com/howtos/jmx/jmx.html
PDF	jmx.pdf	http://pub.admc.com/howtos/jmx/jmx.pdf

If you are reading this document now with a standalone PDF reader, the your distro links may not work.

License

This HOWTO document is copyrighted by Axis Data Management Corp. [<http://admc.com/>] and may not be modified. The ADMC-supplied source code (scripts and Java code) are open sourced by the Apache 2.0 license [<resources/LICENSE.txt>].

Purpose

This is an accelerated Howto for experienced Java programmers to really master, both theoretically and practically, the essential features of JMX, a.k.a. Java Management Extensions. You should already understand Java reflection, and how traditional Java Beans work. If you have dabbled with some facets of JMX, but want a more comprehensive understanding of it, this is for you too.

My goal is to get right to what you want to know and skip all the stuff that would be needed to accommodate a wider audience. I provide you with command-line and graphical Adaptors, and refer you to my publicly accessible JMX Manager where you can work with the objects discussed here.



Tip

Keep the JMX API Spec(s) handy. The distros for all of the JMX implementations that I know of come with API Specs under their doc (or docs) directory. If you need to code to a JMX class, I assume that you're smart enough to know to look up that class in the corresponding API Spec.

Support

Use the designated topic at the support forum at <http://admc.com/jforum/> with questions, suggestions, etc., about this document or its subject. Axis Data Management Corp. [<http://admc.com/>] provides professional development, support, and custom education for JMX implementations, as well as for many other subjects in the realms of computer systems and software development.

Chapter 1. Required Software

This chapter is really unnecessary, now that Java 1.5 and newer come with a basic JMX implementation.

Note that a full JMX implementation only requires Instrumentation, Agent and Distributed Services levels, and some JMX implementations, most notably Sun's RI, distribute the Distributed Services part separately (Sun call's that part the *JMX Remote API RI*).

If you use another JMX implementation, you will have to modify your CLASSPATH to include one or more jar files from the JMX distribution. The commands in this document will work exactly as given only if you are using Java JDK 1.5 or greater.

If you are going to do distributed JMX with a JMXConnector, then I advise you to avoid Java 1.3. I hear that the problems with remote JMX on 1.3 are being resolved, but I won't be spending my time working around the remaining problems. (This doesn't apply to remote administration via *Adaptors*, only with *Connectors*).

Chapter 2. JMX Architecture

Main Architectural Components

The primary goal of JMX is to provide easy and flexible availability to a repository of Java objects. The *registered* objects are *Managed MBeans*. One class could register a Managed Bean today. Tomorrow, some unrelated Java object could invoke methods of that Managed Bean instance.

The main architectural components

MBeans This is the more common name for *Managed Beans*. As explained above, these are objects that can be stored in the repository (which is the *MBean Server*, as described elsewhere). After an MBean is stored in a repository, other Java classes can access specific methods and descriptions within the MBean. Making an MBean is not as simple as implementing an *MBean* interface (there is no class or interface named *MBean* in a JMX implementation). I'll describe how to make MBeans of the different types latter.

MBeanServer The repository for accessing, creating, registering, removing MBeans. (*Registering* and *Removing* MBeans is how you have the MBeanServer manage or stop managing them). MBeanServers make MBeans available to the running JVM, they do not write them to permanent storage. If you destroy an MBeanServer, you will lose all of the MBeans that were in it (this would happen, of course, if your JVM process died). For this reason and many others, you probably want to encapsulate control of your MBeanServer instance in an *Agent* as described elsewhere. You will get your MBeanServer object from a factory (i.e., you will not implement your own).

The MBeanServer is normally accessed through its MBeanServerConnection interface, which works both locally and remotely. (The exception to this is the Agent, which accesses it as an MBeanServer).

JMX Client I am stipulating this term because the JMX specification docs fail to differentiate between Agents and other local JMX programs, and then call remote all JMX programs "remote clients". It's all very turbid.

A JMX Client is any class that uses an MBean object directly (by *directly* I mean, not through a non-JMX facade). In order to do this, the class must be given (or instantiate) an MBean, MBeanServer or MBeanServerConnection, and must have the JMX Implementation jar(s) in its classpath. A JMX Client doesn't invoke methods on MBeans like `mbeanobject.methodname()`, but rather like `mbeanServerConnection.invoke(mbeanID, "methodname", ...)`, `mbeanServerConnection.getAttribute(mbeanID, "attname")`, etc.

Unlike the MBeanServer, JMX Clients do not normally come with the JMX Implementation. There are very few open reusable, open source JMX Clients out there, so, for now, you'll be implementing your own.

In general, when JMX Clients use an MBeanServer instance, they should use the MBeanServerConnection interface whenever possible, because code that uses MBeanServerConnection is portable (i.e. will work as-is either locally or remotely).

The three types of JMX Client

Local JMX Client A Local JMX Client runs in the same JVM as the MBeanServer.

Agent The Agent is the Local JMX Client that instantiates and contains the MBeanServer. Other Local JMX Clients can get references to the MBeanServer to work with, but the Agent has primary responsibility for it. The Agent instantiates, configures, initializes and controls an MBeanServer.

Some Agent responsibilities

- Load some MBeans upon startup.
- Load and initialize Connector(s) (Connectors are described elsewhere).
- Load and initialize Protocol Adaptor(s) (Protocol Adaptors are described elsewhere).
- Store MBeans to permanent storage so that they can be restored when the JVM is restarted.
- Initialize and manage Agent services such as dynamic class loading, monitors, timers, relation services.

Remote JMX Client A Remote JMX Client is just like a Local JMX Client except that it must instantiate a Connector client and connect it up to a Connector server to get a MBeanServerConnection. It then uses the MBeanServerConnection in the same way as a Local JMX Client.

Management Client A client program used for managing MBeans. The management client is usually not a JMX Client, but could be. The essential point is that it is an end-user program that somehow initiates commands that end up invoking MBean methods or services. One example is a web browser connected to a HTML Protocol Adaptor (Protocol Adaptors are explained below). Another example is an existing SNMP Management Gui which can send SNMP commands to a SNMP Protocol Adaptor. In these typical cases, the Management Client programs, the browser and the SNMP Management Gui, know nothing about JMX. Alternatively, you could make a custom Management Client Gui that is a Local or Remote JMX Client.

Connector A connector lets classes in remote JVMs (i.e., *Remote JMX Clients* use MBeans and the MBeanServer via JMX as if they were local. There is a Connector client side piece and a Connector server side piece for every Connector connection. A JMX Agent sets up and configures Connector servers. Remote JMX Clients set up and configure Connector clients.

One Connector object could use the Corba protocol to connect a remote JMX Client to a MBeanServer, for example. Compliant Connector implementations must follow the JMX protocol-specific rules for how to exchange data. In this way, in theory, any Connector client supporting protocol X should be able to connect to any Connector server that supports protocol X.

Protocol Adaptor A protocol adaptor is just a JMX Client that is written to enable non-JMX programs to interface to your Agent.

The name is unfortunate, because it describes a *Connector* better than a *Protocol Adaptor*. Protocol Adaptors have nothing to do with protocols, but Connectors do. The purpose of

a Protocol Adaptor is to serve some type of client(s). The adaptor can support any (non-JMX) interface(s) whatsoever-- for example, by stdin/stdout. (If you call the interface mechanism a "protocol", then every program in the world is a *Protocol Adaptor*.) The sole purpose, on the other hand, of a Connector, is to serve a stated protocol according to the spec (as soon as the JMX use for that protocol has been standardized).

It is very easy to differentiate between Connectors and Protocol Adaptors. The server side of both are connected to a MBeanServer, but Connectors have a client piece and Protocol Adaptors do not. The client of a Connector is a Remote JMX Client using a Connector client. The client of a Protocol Adaptor has no JMX code-- it is just any program that can connect to the Protocol Adaptor in any (non-JMX) way that the Protocol Adapter makes available.

To exemplify how Protocol Adaptors are misnamed... I may want to access my Agent both from an existing management web interface, and from a new, dedicated web interface. This would call for two Protocol Adaptors, each one creating HTML appropriate for each user application: One protocol but two Protocol Adaptors that communicate to the external program however they want to over that protocol. But if you use a Connector so that two client programs of any type can access your Agent over, say, a TLS TCP socket, the server runs only one Connector because there is only one network protocol, and the way that data is transmitted over the TLS TCP socket is mandated by the spec.

JMX Conformance Levels

It's good to become conversant with the JMX Conformance Levels, because they come up in the JMX API and literature.

JMX Conformance Levels

Instrumentation	Implementation of MBeans. You are doing Instrumentation if you are writing MBeans. Requirements for the Instrumentation level are documented in the main JMX Specification.
Agent Level	Everything to do with the MBeanServer and JMX Clients, both what the JMX implementation comes with, and what you code. When explaining how to use MBeans (as opposed to how to implement them), the main JMX Spec always calls the JMX client an <i>Agent</i> , but what they say nearly always applies to remote JMX clients also (and is in fact the only place to find requirements about coding remote JMX clients-- other than the Connector setup part). Requirements for the Agent level are documented in the main JMX Specification.
Distributed Services Level	Connectors (including how Remote JMX Clients need to do to use Connectors).
charset	(According to the JMX Levels diagram in the main JMX Spec, Protocol Adaptors are supposed to be covered in this level, but they aren't). Requirements for the Agent level are documented in the JMX Remote API Specification.

Chapter 3. JMX Features

JMX Features

- A Management Client (as well as any MBean) can

JMX Features

- *Register MBeans with the MBeanServer (i.e. add them to the MBean repository).
- *Create and register an MBean with the MBeanServer.
- *Remove an MBean from the MBeanServer.
- *View the published methods of an MBean (these methods are called Operations and Attributes), along with a description of each published method.
- *Invoke the published methods of an MBean.
- Receive notifications when an MBean triggers or throws an event (which is just like throwing a Throwable).
- List (a.k.a. "query") subsets of MBeans using JMX *domains*, wildcards and/or name/value attribute pairs.
- Create, initialize and register MBeans by specifying the class file location and initialization parameters in an *Mlet* file.
- Generate JMX notifications with a Timer Service.
- Enforce and maintain relationships (like *one-to-many*) of subsets of MBeans.
- Control local and remote access to MBeans.
- Precise control and fine-tuning of class loading.
- *Remote JMX Clients that work just like Local JMX Clients after you get a MBeanServerConnection of an MBeanServer.

This document explains only the asterisked * features above. The other features are not explained due to the following reasons.

- I cover MBean name search/queries trivially because that's all that is needed. If you want complex search capabilities, you can do much better with existing Java mechanisms than the poorly designed JMX query mechanisms.
- Narrowing down subsets of MBeans with JMX-specific methods is easy to figure out from the API, and I have yet to run across a case where an application would suffer by retrieving all of the MBean names and using Java 1.0 features to choose the names that you want.
- Notifications are definitely important for dynamic applications with inter-relations among MBeans. Most importantly, to update persistent storage upon MBean removal or registration. Originally, I didn't give high priority to documenting them since the early R.I. releases didn't fully implement it, and not until recently did Sun provide examples for them. I do intend to document Notifications when I have time.
- Sun's Mlets aren't working right for me any more, and the error handling is REALLY BAD. Every substantial JMX product I have seen uses their own method for loading MBeans from a xml file list. Since most people are not satisfied with Mlets, I recommend that, if you want to load MBeans from a file list, you see what your JMX product or JMX distribution has to offer.

- Timer Service and relationship control are not covered by Sun's examples or tutorials, and I get the feeling that these features may change in the future. I am also not convinced that the goals of these features are not handled better by other existing technologies.
- Access control and class loading are advanced features that are explained adequately in the main JMX Spec.

Chapter 4. JMX Resources

JMX Resources

Command-line and standalone HTML-over-HTTP Adaptors.

<http://admc.com/dist/admcjmx-adaptors.jar> . Source code at <http://admc.com/dist/admcjmx-adaptors-src.zip> .

First set your CLASSPATH to include the <http://admc.com/dist/admcjmx-adaptors.jar> file and the classes which you want to manage. Then run the console or http/html adaptor as follows.

```
java com.admc.jmx.JmxStreamAdaptor
```

or

```
java com.admc.jmx.JmxHtmlAdaptor -h
```

(You can, of course, also embed these classes into other apps). The JmxHtmlAdaptor command above will show you the simple syntax to specify port number and security options. To run with TLS encryption, you need to set up keystores, passwords, etc., according to the JSSE Guide that ships with Sun's JDK. See the Instrumentation chapter of this document for screen shots of both the command line and Http adaptors. The source distro also includes a sample policy file and startup script so that you can securely give unknown users access to a limited set of MBeans.

My JMX Management Web Application (war)}

<http://pub.admc.com/oss-products/jmxhtml.war> . Source code at <http://pub.admc.com/oss-products/jmxhtml-src.zip> .

Just like the html-over-http adaptors of the previous item, except this runs in a servlet container instead of starting up its own HTTP server. (This war actually includes the <http://pub.admc.com/oss-products/admcjmx-adaptors.jar> of the previous item). The war uses a client JMXConnector to connect to a JMXConnectorServer. If you don't have your own JmxConnectorServer running, edit and compile the one included in the source zip file. To run with a security manager, edit the included startup script.

Demo Standalone JMX Manager/Adaptor

<http://admc.com:8111/>

You can manage instances of the sample MBeans used in this HOWTO (other than the ones named Trivial*). This is a live instance of my Html-over-Http Adaptor.

Demo JMX Manager/Adaptor Web App at

<http://admc.com/jmxhtml/jmx>

You can manage instances of the sample MBeans used in this HOWTO (other than the ones named Trivial*). This is a live instance of my Web App Adaptor running on Tomcat 5.

MX4J Open Source JMX

<http://mx4j.sourceforge.net/>

Really good. I document use of MX4J's AbstractDynamicMBean, and HttpAdaptor with XSLTProcessor. Personally, I wouldn't do a job involving considerable Dynamic MBean usage if I couldn't use AbstractDynamicMBean. Since Java 1.5 now comes with a basic JMX implementation, the MX4J core JMX implementation is really redundant and just adds ambiguity (viz, whose

JMX classes are the factories returning?). Fortunately, the AbstractDynamicMBean and adaptor classes are very easy to separate from the rest of the mx4j distribution. My adaptors bundle the two mx4j classes needed for AbstractDynamicMBeans, and I recommend this practice.

JMX 1.2.1 RI

<http://javashopl.m.sun.com/ECom/docs/Welcome.jsp?StoreId=22&PartDetailId=7657-jmx-1.2.1-oth-JPR&SiteId=JSC&TransactionId=noreg>

Sun's latest cut of JMX. I haven't looked into the difference between this distribution and what is included in the 1.5 JDK. I do know that this RI includes samples of dynamic MBeans, whereas the JDK doesn't even describe dynamic MBeans.

JMX 1.2 Spec

<http://javashopl.m.sun.com/ECom/docs/Welcome.jsp?StoreId=22&PartDetailId=7127-jmx-1.2-mr-spec-oth-JSpec&SiteId=JCP&TransactionId=noreg>

Specification 1.2 for the Instrumentation and Agent levels.

JMX Remote API 1.0 Spec

http://javashopl.m.sun.com/ECom/docs/Welcome.jsp?StoreId=22&PartDetailId=jmx_remote-1.0-fr-oth-JSpec&SiteId=JCP&TransactionId=noreg

Specification 1.0 for the Distributed Services level of JMX. (aka *JMX Remote API*).

JMX Remote API RI 1.0.1

<http://java.sun.com/products/JavaManagement/download.html>

Sun's latest cut of the JMX Remote API (the Distributed Services level).

JBoss download site

<http://www.jboss.org/index.html?module=html&op=userdisplay&id=downloads>

JBoss has a JMX implementation. I'm not working with it because

1. For the past year or two JBoss is concentrating on generating revenue by providing services, not products. They don't even have a *Products* item on their main web site menu. On their download page, for the current version of JBoss, you can only download one zip file of everything, or one other component (which is not JMX). I'd much rather go to MX4J on Sourceforge, or to Sun's JMX site and download just what I want.
2. When I last downloaded another component product from JBoss (JBossJMS), the only high-level or howto documentation available was a PDF document that I paid for. While the document was ok, it was extremely inconvenient to have to keep going back to a PDF document without any bookmarking or hypertext capabilities. Before that, I had the same exact experience with the full JBoss distribution.

If JBoss now has modern documentation and makes their JMX product easily accessible, email me at blaine_dot_simpson_simpson@admc_dot_com [mailto:blaine_dot_simpson_simpson@admc_dot_com?subject=jmx Howto] and I will update this document.

Thanks to Monal Daxini for contributing the following update about JBoss:

The following is the link http://sourceforge.net/project/showfiles.php?group_id=22866 where you can download the JBoss JMX implementation. This link was provided at the end of the page with URL: <http://www.jboss.org/index.html?module=html&op=userdisplay&id=downloads> . The current release is not JMX 1.2 ready, and has very poor docs as you mentioned.

FYI: I have the Connectors working fine with java 1.3 but had to jump a couple of hoops before I could get it right. Specifically, had to add jaas.jar in the CLASSPATH in order for the examples with Sun JMX RI to execute. However, jaas.jar is not required to compile the examples. If I find any more quirks will let you know. jaas.jar was downloaded separately from sun's website.

Chapter 5. Introduction to MBeans

MBeans

Each MBean object needs a unique identifier in the MBeanServer, of course. That identifier is called the *MBean Name*, and consists of the JMX Domain and key/value pairs. The whole thing is formatted like `DomainName:key1=val1,key2=val2`. The Java class `javax.management.ObjectName` represents an MBean Name (as well as a wild-card for MBean Names). The entire combination of the domain plus all keys and values must be unique. (That is equivalent to saying that the entire MBean Name must be unique).

Once you implement an MBean by any of the methods in the `Instrumentation` chapter of this doc, then you can use a JMX Client to work with them as MBeans (I'm only describing how to use them as MBeans. There is nothing to prevent you from accessing the classes directly instead of through JMX-- but this Howto is about JMX). A JMX Client can instantiate, interrogate, register, retrieve and invoke methods on MBeans. If a JMX Client program has an interface for non-JMX clients, then you can use a non JMX program, like a web browser or the command line, to work with MBeans via the JMX Client (JMX Clients that perform this function are called Adaptors. The first thing we do in the `Instrumentation` chapter is start up an Adaptor). There are additional, non-essential features useful in specific problem domains. I list these in the `JMX Features` chapter.

When you implement any kind of MBean, you (usually) have to tell the JMX system which methods to expose-- because you don't generally want to expose all of them. (The exception to this is with constructors, as I explain next). I call this *publishing*. If you *publish* method `x.y()`, then method `x.y()` will be available to JMX Clients who get access to MBeans for the class X. JMX divides published methods are divided into three groups.

MBean Published Method Types

Constructors Constructor methods. Constructor usage was designed very poorly in JMX. The problem is that MBeans can only be interrogated after they are instantiated. So, before you instantiate the first instance of MBean of class C, you can not use JMX to interrogate class C. Instead of providing a way to interrogate MBean classes before instantiation,

- JMX Clients must use normal Java reflection to find the public constructors for the desired MBean class. As a result, there is no way to see the JMX constructor descriptions if the MBean developer has provided them.
- JMX constructor information provided by the MBean developer is used by a JMX Client only when the JMX Client queries an existing, instantiated MBean of the same type.
- JMX will use any public constructor for an MBean requested by a JMX Client, regardless of whether the MBean developer has told JMX to expose them.

To restate the confusing results of this.

- JMX Clients can always *see* all of the public constructors of an MBean using normal Java reflection.
- If a JMX Client uses JMX to view the public constructors (you can only do with on an existing MBean), it will only see those that have been exposed through JMX methods.
- Whether a JMX Client can see public MBean constructors through JMX interrogation or not, it can use JMX to instantiate new MBeans using any public constructor.

Notice that all of this makes constructor descriptions pretty close to useless. It is impossible for a JMX Client to use constructor descriptions when they are most important-- before the first MBean of the class is instantiated.

Attributes	These are just getter/setter and/or is methods, just like for traditional Java Beans (e.g., <code>getX()</code> , <code>setX(y)</code> , <code>isX()</code>). You can't overload setters. Getters/is methods only work with no arguments at all. You should not use both a getter and an is-- it's one or the other. N.b. that methods of the form <code>getX(arg1)</code> and <code>setY(arg1, arg2)</code> can only be exposed as <i>operations</i> , since getters must be of the form <code>getX()</code> (i.e. no arguments), and setters must be of the form <code>setY(arg1)</code> (i.e. one argument).
Operations	All remaining methods. You should not overload operations (according to the Spec). There is nothing to prevent you from exposing a getter/setter/is method as an operation instead of an attribute-- just don't do both!



Note

JMX Clients will invoke these methods using an `MBeanServerConnection`, like `mbeanServerConnection.method(mbeanID, "mbeanMethodName", ...)`, not like `mbean.mbeanMethodName(...)`. (The latter is possible with a MX4J delegation feature which works just like Axis delegation stubs for SOAP).

MBean Types: Standard vs. Dynamic

According to the JMX Spec, and common sense, you pretty much never want to *change* the JMX Interface of MBeans dynamically. JMX *always* sets up the original interface of an MBean class dynamically (even though with Standard MBeans this is hidden from you). To dynamically *change* the exposed MBean Interface, you would need to make very complex registration, tracking and notification strategies so that the JVM knows which version of the published methods to use for which MBean instances. A true nightmare. In practice, the coding needed just for one-time dynamic MBean interface setup is extremely complex, and you are hereby advised to try to avoid doing that work (some strategies are recommended below). The main point is, it does not make sense to *modify* an MBean's published interface dynamically.

The main division of MBeans is between *Standard* and *Dynamic*. Both Standard and Dynamic MBeans dynamically publish their JMX interfaces. With Dynamic MBeans, you code the methods to publish your methods. With Standard MBeans, the JMX libraries use reflection on your class to publish the methods for you. Dynamic MBeans additionally provide for runtime modification of the JMX interface-- just what Sun recommends against.

Standard MBeans

For the class to expose, you use a naming convention for the methods to expose, then put the signature of those methods into a new interface (the *MBean Interface*) which follows a naming convention. (I.e., the MBean Interface interface name is not *MBean*, but follows a naming convention). JMX uses reflection to publish the exposed methods at the right time. This is all done very similarly to how traditional Java Beans work.

Unfortunately, Sun decided to not support publishing descriptions for Standard MBeans and their exposed methods.

Dynamic MBeans

Dynamic MBeans can publish a description for each exposed method. IMO, this is the best feature that there is to JMX. Your end users can use a generic interface to browse existing MBeans which themselves tell the user how to use them.

Varieties of Dynamic MBeans

Traditional Dynamic MBean (aka. Dynamic MBean)	You code a few methods to satisfy the <code>DynamicMBean</code> interface. Unfortunately, these methods must assemble ugly, overly-nested objects. Most of these
--	--

methods return different combinations of the same static information all the time-- if they do not, you will have the fiasco explained in the first paragraph in the *MBean Types: Standard vs. Dynamic* section.

Hand-made Dynamic MBeans are a horrific thing. I highly recommend that if you want to use traditional Dynamic MBeans or Open MBeans, you make use of `AbstractDynamicMBeans`.

Open MBean

An Open MBean is just a Dynamic MBean that follows some conventions which makes it less powerful and convenient, but more portable. An Open MBean implements the `DynamicMBean` interface. The basic idea is that if you stick to using primitive data types and the basic Java-defined and JMX-defined classes, then all of this data can be passed without needing anything else in the classpath, and there is never any possibility of version conflicts of outside classes. (Maybe some day they will make a `VeryOpen MBean` that can only use String objects. Since every version of every JVM can handle compatible Strings, and every other class can be converted to and from a String, `VeryOpen MBeans` would be more portable than Open MBeans.)

Model MBean

Developing Dynamic MBeans and OpenMBeans is straight forward in that you code methods in the class that you want to expose (though, as explained, those methods are ugly and difficult). With Model MBeans, you do not code in the class file that you want to expose. You use a factory to instantiate an empty Model Mbean, then tell that Model MBean what methods to expose (these methods can be from any accessible classes). So, you do not have a .java file that *is* a Model MBean, you get a Model MBean from the JMX implementation and work with it. (Just like you usually don't have a .java file that *is* a `Properties`. You retrieve or instantiate a `Properties` instance and work with it.)

`AbstractDynamicMBean`

`AbstractDynamicMBean` is a class implementing `Dynamic MBean` (whereas an Open MBean is a subset of `Dynamic MBean` classes and a `Model MBean` is an interface that extends `DynamicMBean`). `AbstractDynamicMBean` is provided by `MX4J`. To use it with a JMX implementation other than `MX4J`, you just include the following two classes to your CLASSPATH: `jmx.utils.Util`, `jmx.AbstractDynamicMBean`. You can obtain these classes from any `mx4j-impl.jar`, `mx4j.jar`, or my own <http://pub.admc.com/oss-products/admcjmx-adaptors.jar> .

`AbstractDynamicMBeans`, unlike traditional `DynamicMBeans`, are easy to understand by example. In the *Instrumentation* chapter of this document, I provide detailed, documented examples.

Chapter 6. Instrumentation

Definitely read the `Introduction to MBeans` chapter before you read this chapter.

I am giving instructions as if you are the JMX implementation in the 1.5 Java JDK. A Java developer shouldn't have any difficulty figuring out what classpaths should be with some other JMX implementation.

Instrumentation means implementation of MBeans themselves (as opposed to classes that use MBeans).

I don't detail implementation of either Model MBeans or Open MBeans. There are limited situations where Model MBeans are really called for. I don't cover them here because it is a difficult undertaking which you should avoid if you can. If I take it upon myself to simplify them with a template system or something in the future, I'll add a section about them here. Open MBeans are simply Dynamic MBeans that follow certain conventions, so, if you need to work with them, just learn how to make Dynamic MBeans here, then take some No-Doze and read the Open MBean section of the JMX Spec.

General MBean Coding Gotchas

I've had problems with `is()` method behavior with Sun's JMX implementation. They work as advertised with MX4J. (UPDATE: I have still not tested this with Java 1.5).

Jmx has lots of prohibitions and warnings against overloading. Anybody can code up a class where the base name *alpha* is used for a getter, setter, is, and multiple other methods, like

1. `getAlpha();`
2. `getAlpha(arg1);`
3. `getAlpha(arg1, arg2);`
4. `setAlpha(anInt);`
5. `setAlpha(aString);`
6. `setAlpha(arg1, arg2);`
7. `isAlpha();`
8. `alpha();`
9. `alpha(aString);`
10. `alpha(anInt);`
11. `alpha(arg1, arg2);`

That's fine as long as you don't want to publish them with JMX. To publish with JMX, for each base name (*alpha* in this case), you are allowed only

1. One getter with form `getAlpha()` **OR** one getter with form `boolean isAlpha()` **OR** one operation of any form.
2. One setter with one argument (i.e. `setAlpha(x)`).

Some of these rules are enforced and some are not. To minimize changes of obsolescence, follow all of these rules.

Set up a JMX Dev Environment

Read the [Required Software](#) chapter if you haven't already.

If you are just learning to create MBeans, then you will need an Agent server to test them with. Download `adm-cjmx-adaptors.jar` from <http://pub.admc.com/oss-products/admcjmx-adaptors.jar>. This has my generic JMX Adaptors, including the command-line Adaptor that we're going to use now.

```
export CLASSPATH
CLASSPATH=".:path/to/admcjmx-adaptors.jar"
java com.admc.jmx.JmxStreamAdaptor
```

from a Bourne-compatible shell (which includes "bash").

Enter the command "list" to see a list of the MBean names for all the registered MBeans. (I talk about MBean name queries in the [JMX Client Implementation](#) chapter. Since you haven't registered any MBeans yet, all of the MBeans listed are part of the Jmx Implementation itself or the Adaptor. Leave this Adaptor running and use another window to get another shell in the same directory. (We're going to code up some MBean classes in the other window, then use this window to load and inspect them).

Create a Standard MBean

Standard MBeans publish all of their public constructors automatically. You publish attribute and operation methods by putting them into your MBean Interface file. JMX Clients that look at your MBean class will see the same public constructors whether they use normal Java introspection or use JMX interrogation facilities for an existing MBean instance. JMX Clients can instantiate instances of your MBean using any of its public constructors. (The reasons behind these constructor idiosyncracies are explained in the [Introduction to MBeans](#) chapter).

Download `TrivialStd.java` or cut and paste this to a file named `TrivialStd.java` in the same directory.

```
public class TrivialStd implements TrivialStdMBean {
    public String speak() {
        return "Hello world";
    }
}
```

This is your target implementation class. Note that I coded no constructor so Java will create a no-arg public constructor for me by default. Like I said, JMX will automatically publish all public constructor, and this includes the default one.

Download `TrivialStdMBean.java` or cut and paste this to a file named `TrivialStd.java` in the same directory.

```
public interface TrivialStdMBean {
    public String speak();
}
```

This is your MBean Interface.

In your second terminal window, run

```
javac Trivial*.java
```

Back in your `JmxStreamAdaptor` session, enter the following. (Don't type # or what follows on each line-- those are comments).

```
instantiate TrivialStd dom:k=v # Creates MBean of given class and name
```

list dom:*	# List MBean names with domain "dom"
dump dom:k=v	# Displays MBean of given name
invoke dom:k=v speak	# Invokes given operation of given MBean

The descriptions that the Adaptor shows (for anything-- class, attributes, operations, constructors) are just dummy values. Standard MBeans do not support descriptions, so no JMX Client can obtain descriptions for them.

Here's what your output should look like.

```
Created MBeanServer with ID: 6cd7d5:fa884d34b0:-8000:onella.icfconsulting.com:1
Available commands (you can re-execute but not edit previous commands):
  classConstructors CLASS_NAME      (lists them)
  constructors MBEAN_NAME          (lists JMX published cons)
  dump MBEAN_NAME                  (displays details of MBean)
  list [MBEAN_NAME_WILDCARD]       (lists MBean names)
  history                          (displays command history)
  invoke MBEAN_NAME OP [argType1, argType2...]
  instantiate CLASS_NAME MBEAN_NAME [argType1, argtype2...]
  attribute MBEAN_NAME ATT_NAME NEW_VALUE
  -                                (re-execute previous command)
  -X                               (re-execute Xth previous command)
  spool [/SPOOL/FILE.txt]         (no file turns off spooling)
EOD (like Ctrl-D/Z) to exit.

> instantiate TrivialStd dom:k=v > list dom:*
{
  dom:k=v
}
> dump dom:k=v
DESCRIPTION: Manageable Bean

JMX-Declared CONSTRUCTORS:
Constructor exposed for management
  ()

ATTRIBUTES:
OPERATIONS:
java.lang.String: speak [Operation exposed for management]
  ()
> invoke dom:k=v speak
Hello world
>
```

Now that you're not shaking in fear any more, make a subdirectory named "tst" and download this more substantial Standard MBean to that directory. `tst/SampleStd.java` and `tst/SampleStdMBean.java`. While more substantial than the previous MBean, it's still pretty simple. Do study the code and see how specific selected attributes and operations are published. Compile the classes into the `tst` subdirectory, then use your `JmxStreamAdaptor` to instantiate a few instances, invoke setters and operations, and dump the target MBean to see the results. You will definitely need a scrollbar on your window to see the dump. (You can also use the "spool" command to send output to a file).

Use the "classConstructors" and "constructors" commands of the Adaptor. Notice that you can only run "constructors" on an existing MBean instance whereas you can run classConstructors on any class, MBean or not, instantiated or not. Even though the constructor descriptions are dummy values, notice that you can only see the descriptions with the "constructors" command.

Notice that to instantiate an MBean, you only need to know the class name and arguments-- nothing JMX-specific. That solves the chicken-egg problem, but it means that you can't make use of any JMX publishing information (including *descriptions*, which will become useful once we move on to Dynamic MBeans) until after you instantiate an MBean.

Up to this point, we have used the command-line adaptor. I wanted to use the simplest adaptor possible in order to avoid a problem of Sun's JMX tutorial. When first being introduced to MBeans, we want to spend our time learning

about JMX and MBeans, not learning how to operate a specific management tool. The command-line tool is perfect for working with little MBeans like `TrivialStd`.

However, at this point you should have a firm grasp on what MBeans are and what you can do with them. A graphical management tool makes it much easier to view MBeans with many constructors/attributes/operations, like *SampleStdMBean*. If you want to switch to a graphical tool, you can use my demo site at <http://admc.com:8111/>, or run your own instance. On my site you can instantiate and use any of the test.* classes discussed here. To start up your own instance, just run

```
java com.admc.jmx.JmxHtmlAdaptor 1289
```

instead of

```
java com.admc.jmx.JmxStreamAdaptor
```

This will run a http service on port 1289 of *IPADDR_ANY*. Run `java com.admc.jmx.JmxHtmlAdaptor -h` if you're interested in using other settings (like TLS). You then use a browser to go to *http://localhost:1289/* (or whatever port you specified).

The home page consists of an entry field for narrowing down the MBean list (it defaults to listing all MBeans in the repository); a list of MBean names; and a button to display the public constructors for the MBean class name that you enter.



The screenshot shows a Mozilla Firefox browser window with the title "HTML-over-HTTP JMX Agent - Mozilla Firefox". The address bar contains "http://admc.com:8111/jmx?command=list". The main content area features a large heading "Standalone HTML-over-HTTP Adaptor Demo Site". Below the heading, there is a paragraph of text and a bulleted list of MBeans. Further down, there are two paragraphs of text, another bulleted list, and a final paragraph. At the bottom, there is a search filter input field containing "dom:*" and a "Filter/Refresh" button, followed by a section titled "MBeans" with a bulleted list of links.

HTML-over-HTTP JMX Agent - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

http://admc.com:8111/jmx?command=list

Standalone HTML-over-HTTP Adaptor Demo Site

Use the "Display class constructors" entry on this page to create some of these MBeans

- `tst.DelegatedADM`
- `tst.SampleADM`
- `tst.SampleStd`
- `tst.SampleWrapper`

You can download the source code for all of these MBeans from <http://admc.com/blaine/howtos/jmx/>.

You can download this application itself as [binary](#) or [source](#). (Also included are a startup script and a command-line Adaptor). Documentation on how to use the application is available in my [JMX Howto](#).

Because this is a public server, the following restrictions are in place.

- The Adaptor itself is not exposed as an MBean.
- Attribute-set and operator invocations are prohibited on classes `mx4j.*`

Try to instantiate the MBean *TrivialStd*. You should be able to view the constructors. The Security Manager should prevent you from instantiating a *TrivialStd* MBean.

[Email me](#) if you have something constructive to say.

dom:* Filter/Refresh

MBeans

- [dom:class=tst.DelegatedADM,id=1](#)
- [dom:class=tst.SampleADM,id=1](#)

HTML-over-HTTP Adaptor's MBean Listing page.

If you click on an MBean Name, that MBean will be displayed for you. The adaptor can handle Strings and primitives, plus it can read (but not input) Collections of these types. If a constructor, operator or getter has an argument of a type outside of this, then you can only view that item. The constructors on the MBean dump page are the JMX-published constructors. If you use them, you will create an additional MBean instance with the MBean Name that you supply.

HTML-over-HTTP JMX Agent - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

http://admc.com:8111/jmx?command=dump&ε

dom: class=tst.SampleAl tst.SampleADM		No-arg constructor
dom: class=tst.SampleAl tst.SampleADM	<i>java.lang.Integer</i> <input type="text"/> <i>double</i> <input type="text"/>	Instantiates w/ given Integer and double
tst.SampleADM	<i>java.lang.Integer</i> <i>java.lang.Class</i> <i>double</i>	Like 2-arg cons, but takes a Class

ATTRIBUTES

ret. type	att. name	value	commit	description
java.util.ArrayList	AL	<ul style="list-style-type: none"> • one • two • three 	<input type="button" value="update"/>	AL att description
double	D	<input type="text" value="7.8"/>	<input type="button" value="update"/>	D att description
[D	DA	<ul style="list-style-type: none"> • 1.1 • 2.2 • 3.3 	<input type="button" value="update"/>	DA att description
int	H (WO)	<input type="text"/>	<input type="button" value="update"/>	H att description
java.util.HashMap	HM	<ul style="list-style-type: none"> • 3tres • 2ldos • 1luno 	<input type="button" value="update"/>	HM att description
java.lang.Integer	I (RO)	<input type="text" value="7"/>	<input type="button" value="update"/>	I att description
[Ljava.lang.String;	SA	<ul style="list-style-type: none"> • eins • zwei • drei 	<input type="button" value="update"/>	SA att description

OPERATIONS

HTML-over-HTTP Adaptor's MBean Dump page.

If you Display Constructors from the main page, you will be able to create new MBeans. Since this function just uses Java reflection, you can list the public constructors for any Java class in the classpath, but instantiation will fail if you try to construct a class that is not an MBean.

Create a Dynamic MBean

You can *use* Dynamic MBeans with any Adaptor the same way that you work with your Standard MBean. The only difference you will see is that the Adaptor will show meaningful description for Dynamic MBean classes and methods. (But see the next paragraph regarding constructor descriptions).

As explained in the Introduction to MBeans chapter, JMX Clients have access to all of the public constructors of your target MBean class. JMX Clients can use normal Java reflection to find all of the public constructors. However, if JMX Clients use JMX facilities to look at your constructors, they will only see the ones that you tell JMX about (to do this manually, you define entries in the MBeanInfo structure returned by one of your methods-- but hopefully you won't do it manually). This can only be done by the JMX Client if it already has an instance of your MBean. To make sure that all JMX Clients get a consistent view of your MBean, you should tell JMX about all of your public constructors. (This wasn't a consideration for Standard MBeans since they auto-publish their constructors). In addition, JMX Clients can only see *descriptions* of constructors if they interrogate the constructors using an existing MBean instance and if you have told JMX about this constructor, as described above.

To create a Dynamic MBean, you start with the target class that you want to expose, and implement the DynamicMBean interface according to the API (for javax.management.DynamicMBean). This is neither easy nor enjoyable.



Tip

I highly recommend that you try to avoid coding DynamicMBeans directly.

The Dynamic MBean design is bad, unnecessarily complex, and inherently difficult to maintain. You have to implement a bunch of methods which all return static information (unless you want to sabotage your application by changing the MBean interface at runtime). You have to assemble complex, nested structures containing redundant information.

If you have Sun's RI, then study Sun's example at **JMX_HOME** /examples/DynamicMBean/SimpleDynamic.java. To appreciate the extraordinary complexity and mess, search through there for "state" (do a case insensitive search). SimpleDynamic has a simple MBean Attribute (= getter/setter) for "state". Of course you must implement the get and set methods, but take a look at how much additional coding you need to write and maintain to expose these methods as an MBean Attribute.

If you are going to be coding Dynamic, Open or Model MBeans manually, you need to learn how to use the following classes (as well as many member classes used within these).

- javax.management.MBeanInfo
- javax.management.MBeanOperationInfo
- javax.management.MBeanAttributeInfo

Create a Model MBean

You have to assemble a MBeanInfo object for Model MBeans, just like for Dynamic MBeans, but there are some major differences. You don't touch the target (non-JMX) class to be exposed. You make a new JMX Client class (or use an existing one) to instantiate a ModelMBean (normally as implemented by RequiredModelMBean) which you use to expose any methods available to you (in this or other classes).

```
mBeanInfo = ...
    RequiredModelMBean mb = new RequiredModelMBean(mBeanInfo);
    mb.setManagedResource(targetObject, "targetReferenceString");
```

See `JMX_HOME` examples/ModelMBean/ModelAgent.java for an example.

Create a Dynamic MBean by subclassing AbstractDynamicMBean

Just follow the commented example of `tst/SampleADM.java`.

Wrappers for existing Java classes

If, for any reason at all, you want to expose methods of an existing class, but don't want to modify that class itself, make an MBean wrapper for it. Maybe you want to isolate the JMX portions, or maybe you don't have the source code for the target class, or maybe you want to expose a Java "interface" instead of a real Java "class".

If you don't need description capabilities, then you can make a Standard MBean which is a subclass of the original class. (When I say *target class*, I mean the original class that you want to expose). Say *Oclass* is the target class, then you could make *Mclass*, a subclass of *Oclass*, for your Standard MBean class. *Mclass* would have the MBean interface *MclassMBean*.

For a DynamicMBean wrapper, I highly recommend that you use `AbstractDynamicMBean` instead of doing it manually. Just follow the commented example of `tst/SampleWrapper.java`.

Notice that you have to code your MBean constructors. Pay attention to the comments.

Using AbstractDynamicMBean for MBeans that subclass something else

It isn't that difficult to take an existing (or new) class that is a subclass of something else and make it into a DynamicMBean without losing the convenience of `AbstractDynamicMBean`.

You modify your target class to implement the `DynamicMBean` interface, but the `DynamicMBean` methods will just delegate to calls to a nested `AbstractDynamicMBean` object that you will code.



Note

Here we are discussing a delegate for instrumentation-side target class. This is different from MX4J delegation stubs, which are JMX Client-side stubs for MBeans.

The first part, implementing `DynamicMBean`, is trivial if you name your nested `AbstractDynamicMBean` object "delegate". Just modify your class declarationline to "implement `AbstractDynamicMBean`", and copy this boilerplate into your class: `ADMDelegation.txt`.

Then you need to code up your `AbstractDynamicMBean` delegate object and take care a few other details. Study the example closely. `tst/DelegatedADM.java`.

Chapter 7. JMX Client Implementation

JMX Clients work with MBeans and/or MBeanServerConnections. An MBeanServerConnection is an interface. The underlying object could be a remote Connector connection to a MBeanServer, or it could be a local MBeanServer. The JMX Client itself is often (usually) also an MBean.

There are a few different ways for a JMX Client to get an MBeanServer/MBeanServerConnection handle. Most commonly, a local MBean would implement javax.management.MBeanRegistration, and would get access to an MBeanServer in that way. Any remote JMX Client or MBean will have access to an MBeanServerConnection. An MBeanServerConnection could also be passed as a constructor or method argument. (An MBeanServer could also be passed this way, but always use an MBeanServerConnection where possible).

From the client-programmer's perspective, the most important functional difference between an MBeanServer and an MBeanServerConnection is, there is a method MBeanServer.registerMBean(), and there is no corresponding method for MBeanServerConnection. Since a remote JMX Client may be running in a different JVM, it is generally impossible to register an existing, local object. Therefore, MBeanServerConnections must use the method createMBean() to both instantiate and register an MBean object on the server-side.

An example of a Local JMX Client could be a class to provide persistence for MBeans. It would store or retrieve based on notification methods or method calls from the Agent or some other JMX Client class.

Sun JMX users need **JMXRMT_HOME/jmxremote_optional.jar** if using a Generic remote protocol such as jmxmp.

If you are going to be coding JMX Clients, you need to learn how to use the following classes from the API.

- javax.management.MBeanServerConnection
- javax.management.ObjectName
- javax.management.MBeanInfo
- javax.management.MBeanOperationInfo
- javax.management.MBeanAttributeInfo

In particular, look over all of the methods in MBeanServerConnection. Many of them are essential to typical uses of a JMX Client, including the methods createMBean, getAttribute, setAttribute, and invoke, which is how clients invoke MBean constructors and methods. (You can alternatively use MX4J delegation stubs, in a similar manner to Axis SOAP delegation stubs).

In this document, there are examples of JMX Clients in the [Distributed Services and Connectors](#) chapter and the [Agent Implementation](#) chapter.

Most of the MBean lookup methods use an ObjectName structure. An ObjectName represents an MBean Name or a wildcard for MBean Name(s). I find JMX Client code easier to understand if you do your normally work with Strings and convert to/from ObjectNames when you need to. For example

```
mBeanServerConnection.someMethod(new ObjectName("dom:key=val1,key=val2"));
```

Wildcards work just as you would want them to.

```
mBeanServerConnection.queryMBeans(  
    new ObjectName("do:*:*"), null);
```

Behind the scenes there are awkward structures to assemble these queries. You would think that with all the attention Sun pays to them, that they would be powerful, but they only support simple, restricted queries. These work fine for

the simple purpose of looking up MBeans. You can work with these structures yourself, but if you really wanted to do complex regex work, you'd be better off to use a real regex facility rather than this half-assed, inelegant implementation. Just pass null as the second argument to `MBeanServerConnection.queryMBeans()` and `.queryNames()`.

To keep the Agent example simple, I used the 2-argument `createMBean` call. To invoke a constructor with argument(s) for an underlying MBean, you have to pass those arguments to JMX. In many cases, the classes are in classloader scope, and it would be much safer and convenient to specify a `Class` array (just like some `java.lang.Class` methods use), but JMX always requires a `String` array to specify the constructor signature. Instead of `{ String.class com.admc.util.AClass.class }`, you have to use `{ "java.lang.String", "AClass" }`. Note that, besides loss of elegance and consistency, you always have to specify the package (i.e., you can't rely on default package names or imports). Here's an example instantiating of an MBean instantiation in `JmxHtmlAdaptor` that uses a 2-argument constructor.

```
ObjectName htmlAdaptorName = new ObjectName(
    "JmxHtmlAdaptor:class=com.admc.jmx.JmxHtmlAdaptor,id=1");
...
Object[] oa2 = { new Boolean(true), new Boolean(false) };
String[] sa2 = { "boolean", "boolean" };
mbc.invoke(htmlAdaptorName, "start", oa2, sa2);
```

Chapter 8. Distributed Services and Connectors

The following phrases all describe the same thing.

- Connectors
- JMX Distributed Services
- JMX Remote API

I recommend that you use Java 1.4 or later if you will be running a JMXConnector. If you run into problems running connectors with 1.3, I don't consider that my problem.

A Remote JMX Client is just like a Local JMX Client except that it must instantiate a Connector client and connect it up to a Connector server to get a MBeanServerConnection. It then uses the MBeanServerConnection in the same way as a Local JMX Client. Remote JMX applications can not work with an MBeanServer object, only an MBeanServerConnection.

The Agent must run a Connector Server using the same protocol that you will use on the client side. If your server serves transport "rmi", then your client(s) need to use protocol "rmi".

RMI and JMXMP URLs

Sample JMXMP Server URL:
service:jmx:jmxmp://0.0.0.0:2004

This simply specifies to have the JMXConnectorServer listen to port 2004 using the JMXMP protocol. 0 . 0 . 0 . 0 is the IP address IPADDR_ANY, which means answer connections to the specified port on any IP addresses for this host.



Note

With Sun's JMX Remote JMX RI, the hostname/ip-addr segment is *not* used to limit the target IP address/name, as it should. It must just match any valid IP address/name for this host (including localhost or 127.0.0.1), and it will then listen to IPADDR_ANY. You will have to do custom coding (or use an IP filtering or firewall product) to limit the listen addresses.

Sample RMI Server URL:
service:jmx:rmi:///jndi:rmi://localhost:2003/jmxdemo

The *rmi://* part specifies that the JMX server will use the RMI protocol, and instead of running its own listener, it will advertize on the RMI server at the specified location. The given host/ip address and port identify the end-point at which to reach an RMI server which is already running. The *jmxdemo* part specifies that the this program will register this service with the key *jmxdemo* on the RMI server.

Sample JMXMP Client URL:
service:jmx:jmxmp://localhost:2004

This is as straight-forward as you can get. The client will attempt to connect to a JMXConnectorServer on port 2004 on localhost and will speak JMXMP protocol.

Sample RMI Client URL:
service:jmx:rmi:///jndi:rmi://localhost:2003/jmxdemo

The *rmi://* part specifies to connect to the RMI server at the specified host and port, and look for the service with the given JNDI name, *jmxdemo* in this case. It will then use the JMX RMI protocol to speak to the JMXConnectorServer which registered that service.

I definitely do not recommend that you use an RMI connector (for reasons discussed below). Nevertheless, most of our examples will use RMI because any self-respecting JMX developer should be conversant with RMI connectors. If you're not going to use an RMI connector, you should at least understand why. The only real-world reason to use RMI, in my opinion, is that you have so many JMX servers which are so volatile that you need dynamic service lookup; or that you are prohibited from using any class outside of the base J2SE (even if it is made by Sun). If you use any protocol other than RMI, you will need to add some jar to your classpath, since J2SE does not contain classes implementing any other protocol than RMI. For JMXMP, this simply means adding `jmxremote_optional.jar` from Sun's JMX Remote RI (available here [<http://java.sun.com/products/JavaManagement/download.html>]).

Here's a simple remote client (also available at `ConnectorClient.java`). It just displays information about the specified MBean, but you can copy and paste the relevant portions to make any application of yours a JMX client; or use this as a starting point. (I'll present a Connector Server shortly). As you can see, there is very little code other than argument parsing and TLS setup (ignore the TLS part for now, if you are just learning about Distributed Services and Connectors right now).

Example 8.1. ConnectorClient.java

```

/*
 * @(#) $Id: ConnectorClient.java 2011 2009-01-09 17:07:43Z blaine $
 *
 * Copyright 2004-2009 Axis Data Management Corp.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

import javax.management.remote.JMXConnector;
import javax.management.MBeanServerConnection;
import javax.management.ObjectName;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;
import javax.management.JMException;
import java.util.Map;
import java.util.HashMap;
import java.io.IOException;
import javax.management.remote.JMXAuthenticator;

/**
 * Dumps info about the specified MBean from the JMXConnectorServer at
 * the specified URL.
 */
public class ConnectorClient {
    static MBeanServerConnection mbsc = null;
    static public void main(String[] sa) throws IOException, JMException {
        boolean tlsMode = false;
        String urlString = null;
        String beanId = null;

        /* Command-line argument parsing */
        if (sa.length == 3 && sa[0].equals("--tls")) {
            tlsMode = true;
            urlString = sa[1];
            beanId = sa[2];
        } else if (sa.length == 2) {

```

```

    urlString = sa[0];
    beanId = sa[1];
}
if (urlString == null) {
    System.err.println(
        "SYNTAX:  java [--tls] ConnectorClient JMXServiceURL beanspec\n"
        + "Sample invocations:\n      "
        + "service:jmx:rmi:///jndi/rmi://saturn.acme.com:9999/jndi_id\n"
        + "      service:jmx:jmxmp://neptune.acme.com:9999\n"
        + "(A JMXMP impl. required in classpath for jmxmp service.\n"
        + "Try Sun's free jmxremote_optional.jar).");
    System.exit(2);
}

Map env = new HashMap();
if (tlsMode) {
    // TLS Mode setup.

    // Note that the settings below are conditional, so you can
    // override then with "java -Djavax...=Y... ConnectorClient..."
    // It's definitely not safe to use -D to set passwords, though,
    // but it's useful for prototyping.
    if (System.getProperty("javax.net.ssl.trustStore") == null)
        System.setProperty("javax.net.ssl.trustStore",
            "server-cert.store");
    if (System.getProperty("javax.net.ssl.keyStorePassword") == null)
        System.setProperty("javax.net.ssl.keyStorePassword",
            "pwdClstore");
    // Comment out the lines above if the server isn't requiring
    // a cert for your client.
    if (System.getProperty("javax.net.ssl.keyStore") == null)
        System.setProperty("javax.net.ssl.keyStore", "client1.store");

    /* The method above is the simplest (IMO therefore the best)
     * method if your application doesn't use certs for any other
     * purpose.  You should use the instance-based TLS configuration
     * method if your app uses certs for any other purpose in a
     * single instantiation (i.e., it could fetch web pages over
     * https, or be a TLS Soap client, etc., or it could run connect
     * to multiple TLS JMXConnectorServers).
     *
     * The instance-based method allocates a SSLSocketFactory
     * based on the SSLContext instance which you instantiate and
     * configure, so you can configure multiple SSLSocketFactories
     * with different SSLContext instances.  This all applies to
     * any standard JSSE TLS application, but for JMX, you
     * associate the allocated SSLSocketFactory to the Connector
     * with:
     *
     * env.put("jmx.remote.tls.socket.factory", yourFactory);
     */

    env.put("jmx.remote.profiles", "TLS");
    //env.put("jmx.remote.tls.enabled.protocols", "TLSv1");
    //env.put("jmx.remote.tls.enabled.cipher.suites",
        //"SSL_RSA_WITH_NULL_MD5");
    // Most users will probably want to use the default TLS
    // protocols and suites.
}

JMXConnector c =
    JMXConnectorFactory.connect(new JMXServiceURL(urlString), env);
// If you aren't setting a profile or any other options, you can use
// null for the second connect() parameter, instead of an empty list.
try {
    mbsc = c.getMBeanServerConnection();
}

```

```

// For this example, I chose to not expose the Adaptor as an
// MBean, which is sometimes a good thing to do for security.
// Therefore, I use it as a normal Java Object.
System.err.println("Info on '" + beanId + "' is:");
javax.management.MBeanAttributeInfo[] aa =
    mbsc.getMBeanInfo(new ObjectName(beanId)).getAttributes();
for (int i = 0; i < aa.length; i++)
    System.err.println(aa[i].getName());
} finally {
    if (c != null) c.close();
}
System.exit(0);
}
}

```

RMI Connector Details

The RMI protocol is required by all JMX Remote implementations, but it requires use of a running rmi registry server, and indirection (most simply by JNDI).

If you start an JMXConnectorServer with an rmi URL, you will need to start an rmi registry server if one is not already running on the host and port specified at the end of your URL. This is easily done by running

```
rmiregistry 2003
```

Where 2003 is the port (by default it'll serve all the host addresses on the server). The program *rmiregistry* comes with Java (Sun's JDKs, at least).

For some reason, Sun put a lot of effort into adding an extra level of indirection into the RMI method. Besides the extra configuration necessary, you actually have to run extra processes for clients to locate your RMI servers. Instead of specifying a URL that points right to your JMX service, your clients must use a lookup service like JNDI, SLP, etc. and JMX access must be made through the lookup service. Most people will never work with a JMX infrastructure large enough to justify the complexity needed for this indirection, but you will need to understand it nonetheless. (We call this phenomenon *over-engineering*).

JMXMP Connector Details

In contrast to RMI, Sun's generic JMXMP Connector is simple and intuitive. Just compare the rmi URL and the jmxmp URL above. The JMXMP Connector doesn't come with J2SE, but all you have to do is download the latest JMX Remote RI from <http://java.sun.com/products/JavaManagement/download.html>, pull the file `jmxremote_optional.jar` from the distribution, and put it in the classpath of your server and client classes. This JMXMP Connector just routes serialized Java objects over a TCP connection. There are two great things about jmxmp. Both server and client just specify the server address and port-- no extra server (like JNDI) is needed. Another great benefit is, JMXMP supports TLS in an intuitive and powerful way, as covered in the [TLS](#) chapter.

A Flexible JMXConnectorServer Agent

Now for the server side of the Connector. Just like most JMXConnectorServer programs, ours controls access to an MBeanServer and is therefore a JMX Agent, hence I've named this program `ConnectorServerAgent`. This is the agent that is used by the demo site web app at <http://admc.com/jmxhtml/jmx> (where it runs under the Java security manager). You can download this source code from `ConnectorServerAgent.java`. If you run this class as-is, it will require the classes `SampleStd.class` and `SampleStdMBean.class` to be in your runtime classpath. As the JavaDocs in the code explain, you can subclass `ConnectorServerAgent` to serve any MBeans that you want to. If you want to play with the rest of the sample MBeans from this Howto, or any of your own MBeans which use `AbstractDynamicMBean`, you'll also need the `AbstractDynamicMBean` classes in your classpath. (Probably easiest to just put <http://admc.com/dist/admcjmx-adaptors.jar> into the classpath). The length of this program is only due to TLS support and to make it scalable (so you can easily use it for your own purposes).

Example 8.2. ConnectorServerAgent.java

```

/*
 * @(#) $Id: ConnectorServerAgent.java 2011 2009-01-09 17:07:43Z blaine $
 *
 * Copyright 2004-2009 Axis Data Management Corp.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import javax.management.MBeanServerFactory;
import javax.management.MBeanServer;
import javax.management.ObjectName;
import javax.management.remote.JMXConnectorServerFactory;
import javax.management.remote.JMXServiceURL;
import javax.management.JMException;
import java.util.HashMap;
import java.util.Map;
import javax.management.ReflectionException;
import java.io.File;

/**
 * Sample Agent that runs a JMXConnectorServer using the specified URL.
 *
 * Subclass and override method loadBeans() to initialize with your own
 * MBeans (or none); and override isRequireClientAuth() if you want to
 * run with TLS but without Client certs.
 *
 * It's useful for debugging (and learning) to be able to run this program
 * in the foreground. If you don't ever want it to run in the foreground,
 * then override serve() to run this.serve() in a Thread (or similar).
 */
public class ConnectorServerAgent {
    private boolean tlsMode = false;
    private String urlString = null;
    private Map env = new HashMap();

    static public void main(String[] sa)
        throws IOException, JMException {
        boolean tlsMode = false;
        String urlString = null;

        /* Command-line argument parsing */
        if (sa.length == 2 && sa[0].equals("--tls")) {
            tlsMode = true;
            urlString = sa[1];
        } else if (sa.length == 1) {
            urlString = sa[0];
        }
        if (urlString == null) {
            System.err.println(
                "SYNTAX: java [--tls] ConnectorServerAgent JMXServiceURL\n"
                + "JMXServiceURL examples:\n"
            );
        }
    }
}

```

```

        + "service:jmx:rmi:///jndi/rmi://localhost:9999/jndi_id\n"
        + "    service:jmx:jmxmp://0.0.0.0:9999\n"
        + "(A JMXMP impl. required in classpath for jmxmp service.\n"
        + "Try Sun's free jmxremote_optional.jar).\n"
        + "(RMI URLs require an RMI registry to be running at the "
        + "specified address/port).");
    System.exit(2);
}
new ConnectorServerAgent(urlString, tlsMode).serve();
}

protected ConnectorServerAgent(String urlString, boolean tlsMode) {
    this.urlString = urlString;
    this.tlsMode = tlsMode;
}

/**
 * Override this class to load your own Beans upon startup, or override
 * with a no-op method to load no beans upon startup (in which case
 * clients will need to add Beans).
 */
protected void loadBeans(MBeanServer beanServer) throws JMException {
    try {
        beanServer.createMBean("tst.SampleStd", new ObjectName("a:b=c"));
    } catch (ReflectionException re) {
        throw new JMException(
            "The sample MBean class 'tst.SampleStd' is not in your classpath.\n"
            + "Either fix your classpath, or subclass "
            + getClass().getName() + '.'');
    }
}

/**
 * Initializes the Agent and runs the JMXServerConnector.
 */
protected void serve() throws JMException, IOException {
    if (tlsMode) setupTls();

    MBeanServer mBeanServer = MBeanServerFactory.createMBeanServer();
    loadBeans(mBeanServer);

    (JMXConnectorServerFactory.newJMXConnectorServer(
        new JMXServiceURL(urlString), env, mBeanServer)).start();

    System.out.println(
        "If you're running this server in the foreground, you "
        + "can stop it with Ctrl-C.");
}

/**
 * TLS Mode setup.
 */
protected void setupTls() {
    // Note that the settings below are conditional, so you can
    // override then with "java -Djavax...=Y... ConnectorServerAgent..."
    // It's definitely not safe to use -D to set passwords, though,
    // but it's useful for prototyping.
    if (System.getProperty("javax.net.ssl.trustStore") == null)
        System.setProperty("javax.net.ssl.trustStore",
            "client1-cert.store");
    if (System.getProperty("javax.net.ssl.keyStorePassword") == null)
        System.setProperty("javax.net.ssl.keyStorePassword",
            "pwdSstore");
    if (System.getProperty("javax.net.ssl.keyStore") == null)
        System.setProperty("javax.net.ssl.keyStore", "server.store");

    /* The method above is the simplest (IMO therefore the best)

```

```

* method if your application doesn't use certs for any other
* purpose. You should use the instance-based TLS configuration
* method if your app uses certs for any other purpose
* (i.e., it could fetch web pages over https, or be a TLS
* Soap client, etc., or it could run multiple TLS
* JMXConnectorServers).
*
* The instance-based method allocates a SSLSocketFactory
* based on the SSLContext instance which you instantiate and
* configure, so you can configure multiple SSLSocketFactories
* with different SSLContext instances. This all applies to
* any standard JSSE TLS application, but for JMX, you
* associate the allocated SSLSocketFactory to the Connector
* with:
*
* env.put("jmx.remote.tls.socket.factory", yourFactory);
*/

env.put("jmx.remote.profiles", "TLS");
//env.put("jmx.remote.tls.enabled.protocols", "TLSv1");
//env.put("jmx.remote.tls.enabled.cipher.suites",
// "SSL_RSA_WITH_NULL_MD5");
// Most users will probably want to use the default TLS
// protocols and suites.
env.put("jmx.remote.tls.need.client.authentication",
        Boolean.toString(isRequireClientAuth()));
// Comment out the line above if you don't want to require
// clients to have their own certs.

if ((new File("access.properties")).isFile())
    env.put("jmx.remote.x.access.file", "access.properties");
// IF file "access.properties" is present in $PWD (from where server
// is started), it must have keys of permitted client cert subjects,
// and values of "readwrite" or "readonly".
// N.b. You MUST ESCAPE all spaces, colons, and equal signes in the
// subject with backslashes!
// Example record:
// CN=proto\ client\ 1,OU=RND,O=Fake\ Corp.,C=US readwrite
}

/**
 * Only used if running with TLS mode.
 *
 * If you want to run TLS mode without Client certs, just override
 * this class and override this method to return false.
 *
 * @returns true (unless this method is overridden).
 */
protected boolean isRequireClientAuth() {
    return true;
}
}

```

For the convenience of UNIX users, you can download my startup script from `runagent.ksh` or script from `runagent.bash`. You will have to fix the file paths for your environment. You can also modify it to run other programs as daemons. Invoke it like `nohup ./runagent.ksh` or `nohup ./runagent.ksh` in order to disassociate it from your login shell.

Chapter 9. TLS

Also known as "SSL"

This isn't the place for a general description of TLS or of security. I intend to cover what I think is the most useful paradigm for secure JMX: TLS over JMXMP protocol. JMXMP does not come with J2SE, but it is available with the addition of a simple jar file to your classpaths. For these efforts, you get a protocol which, according to Sun, is more secure; and just as importantly, it is more simple and maintainable than the JMX RMI protocol. (The Distributed Services and Connectors explains JMXMP in general, and how to set the classpath.

TLS Considerations

Encryption

If you are running TLS, you must use a client and/or a server certificate, and the TCP/IP pipe will be encrypted. Enough said.

Password Authentication

Password authentication is secure over a TLS connection because the password itself can't be observed by outside parties due to the encryption. The problem with using password authentication with JMX is that it, besides typical password maintenance chores, it takes considerable custom coding of callbacks and profiles. (This is a limitation of Sun's implementation of the Connectors. They just didn't design a developer-friendly means to accommodate passwords). For this reason, my general recommendation is to use the authenticated certificate itself for authorization, instead of a password (there are, of course, user cases where this just won't work, like where an end-user credential is passed from a trusted application). See the example programs in Sun's JMX Remote RI if you must do password authentication. The rest of this document is about using certificate key pairs for authentication.

Certificates

Certificate authentication works like it does in most Java TLS apps. By default, the peer's certificate is accepted as authenticated if it is trusted by an approved certificate authority (CA). I recommend a traditional server cert for the server application which will eliminate IP spoofing and man-in-the-middle attacks. In addition, I recommend user client certificates to authenticate and authorize clients (because the certs are easy to maintain, and passwords are especially high maintenance with JMX Connectors). Requiring client certs is often called "client authentication" in this sphere. The keystores containing the private keys for the certs should be protected just like passwords are protected.

Authenticating and Authorizing the Server to the Client

The traditional means, used by web browsers doesn't work too good in this case. Web browsers check the client-requested server name against the authenticated cert returned by the server. This can easily be done with normal SSL socket coding, but they apparently forgot about this need when designing JMX Connector API. The authenticator callback only works on the server side, so a client program would have to work around the API to dig out the server principal details. This does not make for an easily maintainable or a portable client program. The work around for this is to specifically "trust" all of your client's servers' certificates, instead of all CA-trusted certs. To do this, you just put the public certificate into a new key store, and set this key store as the client's trust keystore. For intra-company applications, self-signed certificates work great for the server certificates.

Authenticating and Authorizing the Client to the Server

There are two great strategies to secure access to the server without using any passwords. Both require client certificates.

One strategy is the exact inverse of the previous subsection: The server keeps a trust keystore containing the public certs of all allowed clients. Self-signed certificates work great for this. But if you (or a friend!) are handy with OpenSSL, you can have a more scalable and elegant solution with some additional setup work. What I do is to make a self-signed CA key pair for each server application with OpenSSL, and use that to certify each client key-pair (as opposed to self-signing the client key-pairs). If the clients are not within your own organization, you could use a commercial CA to ensure integrity. The benefit to this is, the trust store used by the server contains just the CA public certificate. The server will accept all keys signed by the CA cert. If the need arises, OpenSSL can be used to expire individual certs prematurely, etc. (You could also use the following item as a simpler alternative to exclude some existing certs).

The alternative strategy is for the server to be more liberal *authenticating* client certificates, and then to decide who to authorize based on the Subjects of individual certs. With some environments, client applications could be responsible for obtaining their own commercial certificates. In others, you could have an application CA just like in the previous item. Your server is set up to use default trust or a custom trust keystore accordingly. After the server successfully authenticates a client based on that trust setup, the cert Subject is looked up in a plain properties file to see whether to grant read access or read-write access (the default being NO access).

As long as you require client certs, you can code your own callback to perform any test that you want against the client's certificate, but in most cases an equality test against the cert Subject should work.

TLS Examples

The shell scripts `storesetup.sh` and `gencerts.sh` set up the stores needed to run the example programs out-of-the-box. Run them in this order. They are Bourne-compatible and should work as-is with Bash or Korn. These use the *self-signed* strategy described above. When you run `ConnectorServerAgent` or `ConnectorClient` in TLS mode, the key store files must reside in the same directory that you run Java from.

Here, once again is `ConnectorServerAgent.java` from the previous chapter.

```

/*
 * @(#) $Id: ConnectorServerAgent.java 2011 2009-01-09 17:07:43Z blaine $
 *
 * Copyright 2004-2009 Axis Data Management Corp.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import javax.management.MBeanServerFactory;
import javax.management.MBeanServer;
import javax.management.ObjectName;
import javax.management.remote.JMXConnectorServerFactory;
import javax.management.remote.JMXServiceURL;
import javax.management.JMException;
import java.util.HashMap;
import java.util.Map;
import javax.management.ReflectionException;
import java.io.File;

/**

```

```

* Sample Agent that runs a JMXConnectorServer using the specified URL.
*
* Subclass and override method loadBeans() to initialize with your own
* MBeans (or none); and override isRequireClientAuth() if you want to
* run with TLS but without Client certs.
*
* It's useful for debugging (and learning) to be able to run this program
* in the foreground. If you don't ever want it to run in the foreground,
* then override serve() to run this.serve() in a Thread (or similar).
*/
public class ConnectorServerAgent {
    private boolean tlsMode = false;
    private String urlString = null;
    private Map env = new HashMap();

    static public void main(String[] sa)
        throws IOException, JMXException {
        boolean tlsMode = false;
        String urlString = null;

        /* Command-line argument parsing */
        if (sa.length == 2 && sa[0].equals("--tls")) {
            tlsMode = true;
            urlString = sa[1];
        } else if (sa.length == 1) {
            urlString = sa[0];
        }
        if (urlString == null) {
            System.err.println(
                "SYNTAX: java [--tls] ConnectorServerAgent JMXServiceURL\n"
                + "JMXServiceURL examples:\n    "
                + "service:jmx:rmi:///jndi/rmi://localhost:9999/jndi_id\n"
                + "    service:jmx:jmxmp://0.0.0.0:9999\n"
                + "(A JMXMP impl. required in classpath for jmxmp service.\n"
                + "Try Sun's free jmxremote_optional.jar).\n"
                + "(RMI URLs require an RMI registry to be running at the "
                + "specified address/port).");
            System.exit(2);
        }
        new ConnectorServerAgent(urlString, tlsMode).serve();
    }

    protected ConnectorServerAgent(String urlString, boolean tlsMode) {
        this.urlString = urlString;
        this.tlsMode = tlsMode;
    }

    /**
     * Override this class to load your own Beans upon startup, or override
     * with a no-op method to load no beans upon startup (in which case
     * clients will need to add Beans).
     */
    protected void loadBeans(MBeanServer beanServer) throws JMXException {
        try {
            beanServer.createMBean("tst.SampleStd", new ObjectName("a:b=c"));
        } catch (ReflectionException re) {
            throw new JMXException(
                "The sample MBean class 'tst.SampleStd' is not in your classpath.\n"
                + "Either fix your classpath, or subclass "
                + getClass().getName() + '.');
        }
    }

    /**
     * Initializes the Agent and runs the JMXServerConnector.
     */
    protected void serve() throws JMXException, IOException {

```

```

if (tlsMode) setupTls();

MBeanServer mBeanServer = MBeanServerFactory.createMBeanServer();
loadBeans(mBeanServer);

(JMXConnectorServerFactory.newJMXConnectorServer(
    new JMXServiceURL(urlString), env, mBeanServer)).start();

System.out.println(
    "If you're running this server in the foreground, you "
    + "can stop it with Ctrl-C.");
}

/**
 * TLS Mode setup.
 */
protected void setupTls() {
    // Note that the settings below are conditional, so you can
    // override then with "java -Djavax...=Y... ConnectorServerAgent..."
    // It's definitely not safe to use -D to set passwords, though,
    // but it's useful for prototyping.
    if (System.getProperty("javax.net.ssl.trustStore") == null)
        System.setProperty("javax.net.ssl.trustStore",
            "client1-cert.store");
    if (System.getProperty("javax.net.ssl.keyStorePassword") == null)
        System.setProperty("javax.net.ssl.keyStorePassword",
            "pwdSstore");
    if (System.getProperty("javax.net.ssl.keyStore") == null)
        System.setProperty("javax.net.ssl.keyStore", "server.store");

    /* The method above is the simplest (IMO therefore the best)
     * method if your application doesn't use certs for any other
     * purpose. You should use the instance-based TLS configuration
     * method if your app uses certs for any other purpose
     * (i.e., it could fetch web pages over https, or be a TLS
     * Soap client, etc., or it could run multiple TLS
     * JMXConnectorServers).
     *
     * The instance-based method allocates a SSLSocketFactory
     * based on the SSLContext instance which you instantiate and
     * configure, so you can configure multiple SSLSocketFactories
     * with different SSLContext instances. This all applies to
     * any standard JSSE TLS application, but for JMX, you
     * associate the allocated SSLSocketFactory to the Connector
     * with:
     *
     * env.put("jmx.remote.tls.socket.factory", yourFactory);
     */

    env.put("jmx.remote.profiles", "TLS");
    //env.put("jmx.remote.tls.enabled.protocols", "TLSv1");
    //env.put("jmx.remote.tls.enabled.cipher.suites",
        //"SSL_RSA_WITH_NULL_MD5");
    // Most users will probably want to use the default TLS
    // protocols and suites.
    env.put("jmx.remote.tls.need.client.authentication",
        Boolean.toString(isRequireClientAuth()));
    // Comment out the line above if you don't want to require
    // clients to have their own certs.

    if ((new File("access.properties")).isFile())
        env.put("jmx.remote.x.access.file", "access.properties");
    // IF file "access.properties" is present in $PWD (from where server
    // is started), it must have keys of permitted client cert subjects,
    // and values of "readwrite" or "readonly".
    // N.b. You MUST ESCAPE all spaces, colons, and equal signes in the
    // subject with backslashes!

```

```

    // Example record:
    // CN\=proto\ client\ 1,OU\=RND,O\=Fake\ Corp.,C\=US  readwrite
}

/**
 * Only used if running with TLS mode.
 *
 * If you want to run TLS mode without Client certs, just override
 * this class and override this method to return false.
 *
 * @returns true (unless this method is overridden).
 */
protected boolean isRequireClientAuth() {
    return true;
}
}

```

This time, study the TLS parts. The most important difference is setting the TLS *profile*-- that requires TLS mode for connections. To use TLS, you must set up the needed keystores, then invoke the server with the optional `--tls` argument. You can see from the code what system properties you can set to change behavior without coding. Comments in the code explain how you can subclass `ConnectorServerAgent` for other changes, like to load your own selection of MBeans upon startup. The `jmx.remote.x.access.file` setting provides the only easy way to perform tests on authenticated certs, but you could also reject certs from being authenticated by implementing a `jmx.remote.authenticator` authenticator.

`ConnectorClient.java` is trivial compared to the server. All it does is enable TLS by enabling the TLS *profile*, then setting the JSSE system properties, if they were not already set by some other means. Clients can't use the JMX Remote API to set up an authenticator, like servers can.

A few gotchas

Both key store files, and the optional `jmx.remote.x.access.file` are read in from the filesystem directly, not from the classpath.

Be very careful about the keys in the JMX environment Map. Unused keys are silently ignored. So, mistyping one character could cause drastic changes without the system giving any useful diagnostics.

Some of the TLS failure error messages generated by JSSE and JMX are terribly misleading. If you get a very obtuse TLS error message, make sure that your keystores reside where they should, are readable, etc.

If you are going to use System Properties to set up your JSSE settings (as opposed to instance-based settings), as described in the source code, use the same exact password for the records within a keystore as for the containing keystore itself.

There is no need to set the `trustStorePassword` property for normal use. You don't need a password just to read public information from a key store, and most applications (including JMX applications) will not modify their trust keystore at runtime.

Chapter 10. Protocol Adaptors

A Protocol Adaptor is just a JMX Client that provides a facade to non-JMX programs to access MBeans. An adaptor running locally will normally get a reference to the MBeanServer by implementing the `javax.management.MBeanRegistration` interface. Adaptors that run remotely can use their `MBeanServerConnection`.

My console Adaptor (`com.admc.jmx.JmxStreamAdaptor`) and Http/html Adaptor (`com.admc.jmx.JmxStreamAdaptor`) are described in the Instrumentation chapter. The JMX Resources chapter of this document has links to these, as well as to my web application JMX adaptor (war format).

Sun's RI includes a HTML Adaptor from their JDMK (a commercial JMX product of Sun's). MX4J comes with a very good HTTP adaptor. It's a very good product, but be aware that (a) it requires access to Java XML and XSLT libraries, and (b) it can not run over a Connector (i.e., it has to run as a local JMX Client). The Appendix of this document has an example of a very simple program to run MXJ's HttpAdaptor.

Chapter 11. Agent Implementation

An Agent is the Local JMX Application that instantiates and manages the MBeanServer.

There are several things that an Agent should do, and lots of stuff that it may do.

Essential Agent responsibilities

Instantiate an MBeanserver

```
mBeanServer = MBeanServerFactory.createMBeanServer();
```

Instantiate some MBeans and load them into the MBeanServer like

```
mBeanServer.registerMBean(this, new ObjectName("Jamama:type=JamamaJMXAgent"));
```

AND/OR

```
mBeanServerConnection.createMBean("tst.SampleStd", new ObjectName("a:b=c"));
```

Some of these MBeans may be Protocol Adaptors. You could also use an MLet file to load MBeans (or implementation-specific load-by-config-file methods). See the API for *MLet*.

Load and initialize Connector(s)

```
(JMXConnectorServerFactory.newJMXConnectorServer(url, new JMXServiceURL("service:jmx:jmxmp://localhost:2004"), mBeanServer)).start();
```

(As noted in the *Distributed Services and Connectors* chapter, if you use an rmi service instead of jmxmp, you will need to run an rmiregistry server in order for clients to be able to access your ConnectorServer).

See the *Distributed Services and Connectors* chapter for an example of a flexible Agent which you can use as the basis for your own Agent.

Appendix A. Running MX4J's HttpAdaptor

Below is an agent that sets up MX4j's HttpAdaptor to serve HTML, then runs it. Setup is trivial with Java 1.5 or later. I ran this adaptor for years with Java 1.4, but if you run Java older than 1.5, you will need to make sure that your CLASSPATH includes a JMX implementation and several XML and XSLT libraries (contact me if you want an exact list which works).

To run with Java 1.5 or newer, just make sure that the MX4JHttpAdaptor class and mx4j-tools.jar from the MX4J distribution are in your CLASSPATH, then run `java MX4JHttpAdaptor`.

You can download this source code from `MX4JHttpAdaptor.java`.

```
/*
 * @(#) $Id: MX4JHttpAdaptor.java 2011 2009-01-09 17:07:43Z blaine $
 *
 * Copyright 2004-2009 Axis Data Management Corp.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

import mx4j.tools.adaptor.http.HttpAdaptor;
import javax.management.MBeanServerFactory;
import javax.management.MBeanServer;
import javax.management.MBeanServerConnection;
import javax.management.ObjectName;
import mx4j.tools.adaptor.http.ProcessorMBean;
import mx4j.tools.adaptor.http.XSLTProcessor;

/**
 * Runs MX4J's HttpAdaptor with an XSLT Processor.
 * This results in a generic HTML over HTTP JMX Management Console.
 *
 * Syntax: java MX4JHttpAdaptor [PORT] (port defaults to 8112)
 */
public class MX4JHttpAdaptor {
    /**
     * Runs as a standalone Java process.
     * Serves its own HTTP, so no http container needed.
     * Also has its own JMX Agent, so no JMX ConnectorServer is needed.
     */
    static public void main(String[] sa) throws Exception {
        // Must use an MBeanServer (a.o.t. a MBeanServerConnection, since
        // the method HttpAdaptor.setProcessor() is not JMX-exposed.
        MBeanServer mbs = MBeanServerFactory.createMBeanServer();

        ObjectName name = new ObjectName(
            "mx4j:class=mx4j.tools.adaptor.http.HttpAdaptor,id=1");
        // Make our own Java object so we can run adaptor.setProcessor() below.
        mx4j.tools.adaptor.http.HttpAdaptor adaptor =
            new mx4j.tools.adaptor.http.HttpAdaptor(
                ((sa.length == 0) ? 8112 : Integer.parseInt(sa[0])),
                "0.0.0.0");
        // Previous line sets the listen port and hostname.
        // "0.0.0.0" listens to all IP addresses. You can use a specific
    }
}
```

Running MX4J's HttpAdaptor

```
// host name or IP address if you wish.

mbs.registerMBean(adaptor, name);
mbs.invoke(name, "start", null, null);
adaptor.setProcessor(new XSLTProcessor());
/* De-comment this block to run this in foreground.
System.err.println("Adaptor started. Hit ENTER to exit completely.");
System.in.read();
mbs.invoke(name, "stop", null, null);
System.out.println("Exiting Mx4j.main()");
*/
}
}
```

MX4J - MBean View - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

http://admc.com:8112/mbean?objectname=dom%3Aname

MX4J/Http Adaptor JMX Management Console

Server view **MBean view** Timers Monitors Relations MLet About

MBean Description

Attributes

Name	Description	Type
AL	AL att descript.	java.util.ArrayList
D	D att descript.	double
DA	DA att descript.	ID
H	H att descript.	int
HM	HM att descript.	java.util.Hash Map
I	I att descript.	java.lang.Integer
SA	SA att descript.	Array of java.lang.String

Operations

Name	Return type	Description
get3	[java.lang.String;	3 op descript.
Parameters	id Name Description	Class
	0	java.lang.String
get22	float	22 op descript.
Parameters	id Name Description	Class
	0	int
classOk	boolean	classOk op descript.
Parameters	id Name Description	Class
	0	java.lang.String
	1	java.lang.Class
times2	void	times2 op descript.

Constructors

Class	Description
tst.SampleADM	Instantiates w/ given Integer and double
Parameters	id Name Description
	0

Viewing a SampleADM MBean instance with MX4J Http Adaptor as described

For the convenience of UNIX users, you can download my startup script from `runmx4j.ksh` or script from `runmx4j.bash` (depending on your preference for Korn or Bash). You will have to fix the file paths for your environment. You can also modify it to run other programs as daemons. Invoke it like `nohup ./runmx4j.ksh` or `nohup ./runmx4j.bash` in order to disassociate it from your login shell.