

# Java Application Monitoring and Management

A practical guide to using JMX



Allard Buijze  
Software Architect

## Table of Contents

Introduction .....	3
About the author .....	3
Why you should bother .....	4
JMX Structure .....	4
Choosing your MBeans .....	5
Types of MBeans.....	5
Common examples of MBeans.....	6
System MBeans .....	7
Exposing your MBeans .....	8
Registering your MBeans with the MBean Server.....	9
Defining the MBean Name .....	9
Finding an MBean server .....	10
Using the Spring Framework.....	10
Provide external access to your MBean server.....	11
Enabling connectors.....	11
Security.....	12
Using the Spring Framework.....	13
Connecting to your application .....	14
Jconsole .....	14
JManage.....	14
Programmatically.....	14
Conclusion .....	15

## Introduction

In this document, I describe how JMX can be used to monitor applications and why this is important.

This document is primarily intended for application developers and software architects that work on applications with high uptime and/or load requirements. Some sections will also be interesting for project managers and requirement analysts that work on this type of project. Both functional and technical aspects of application monitoring are covered.

First, the use case for JMX is specified. It provides the cases in which JMX provides most benefit for your application. Next, I give a high level overview of the components that make up the JMX architecture. Following that chapter, I provide some guidelines on how to make a sensible choice of which components to expose. How to expose those components is explained in the fourth chapter. Finally, chapter 5 provides some information on how to connect monitoring tools to your application.

Although many concepts are available in earlier versions of Java, this document assumes Java 6. Where applicable, I will specifically mention the differences of Java 6 and earlier versions.

### About the author

Allard Buijze is a Software Architect at JTeam. In that role, he investigates different technologies and methodologies to find out how they can help develop better code with less effort. One way to test these methodologies is to apply them in real projects or proof-of-concepts. However, he strongly believes that sharing knowledge and visions is the best way to make progress. No developer is able to do anything on his own. By sharing knowledge, developers hand over what they know to the next generation of developers, giving them the opportunity to move on and learn more. “You can only grab something if your hands are empty.”

## Why you should bother

Today's applications have very high requirements on both uptime and performance. Applications are expected to be available 24 hours a day and 7 days a week to thousands or even millions of users. The result of application downtime is something we can often hardly oversee. For some businesses, a few minutes outage will result in multi-million euro claims, while others might be lucky and get away with it. The bottom line is that application outage leads to loss of revenue.

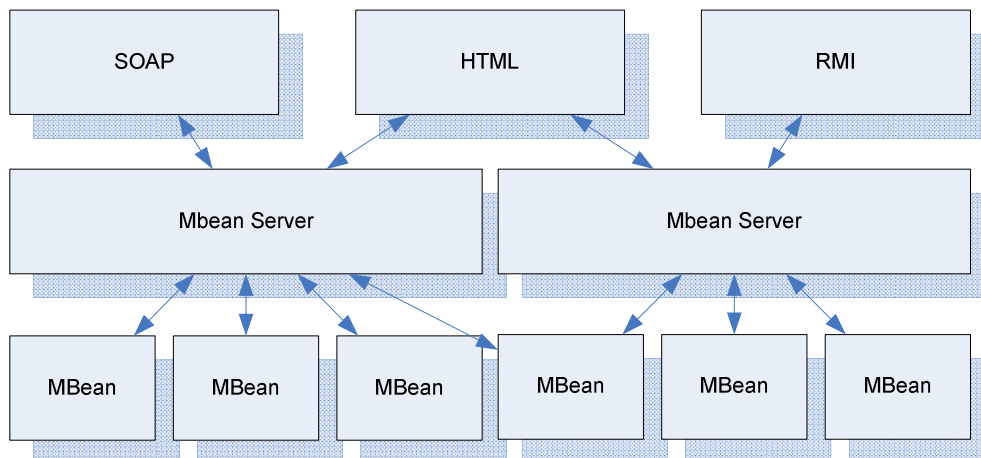
Furthermore, the internet is a hostile and opaque environment. Publishing an application to the public is both an opportunity and a risk. You can offer a service to potentially millions of users all over the world. However, if these millions of users use your application at the same time, your application is not likely to be able to cope with that demand.

In such environment, it is very important to monitor the state and activities of a server. Being able to do so continuously is very important if your business depends on the application.

JMX offers a uniform interface for application management. Monitoring tools can use this interface to monitor the application state, as well as change certain settings. By properly selecting the information you wish to expose, you provide monitoring tools with the ability to monitor your application's health. For example, you could expose information about the state of a database connection, such as number of queries processed and response time statistics. Those statistics give a good indication of the health of the connection. Similar statistics could be created for other integration points and key services.

## JMX Structure

JMX consists of three main layers: the connection layer, the MBean Servers and the MBeans.



The MBean (Management Bean) is the exposed unit in JMX. In its simplest form, an MBean is just a POJO (plain old java object), with getters, setters and possibly some other methods.

The properties of the MBean are exposed as attributes, while methods can be invoked by monitoring tools.

In a typical JMX environment, several MBeans are exposed via a single MBean server. Depending on the type of application, it could be desirable to register different MBeans on different MBean servers.

Each MBean server is externally exposed via one or more connectors. The default JMX connector is RMI, but any connector could be used. Connectors exist for the most common protocols, such as SOAP and HTTP. The MBean server is easily accessible programmatically, making it possible to create your own connectors with little effort. For more details about using connectors, see “Exposing your MBeans”.

MBeans can be registered and removed from an MBean Server at any time. However, application startup is typically the time when MBeans are registered. Again, registration is done programmatically, allowing you to implement a registration strategy with reasonably little effort.

Since Java 5, an instance of the MBean Server is always present: the “Platform MBean Server”. Typically, application containers use this MBeanServer instance to expose their MBeans. It is good practice to provide your own MBeans in the same MBeanServer, although some use cases might require you to do otherwise.

## Choosing your MBeans

The state of an application is formed by the available object instances and their field values. The values of these fields may give an indication on the current state and activities of your application. By exposing these fields through JMX, you provide administrators the ability to see what is going on within the application.

## Types of MBeans

There are four important concepts in JMX that allow you to find out what is happening inside a running application. Each MBean you expose deals with one or more of these concepts.

### State

The first is state: the actual current value of fields that provide infrastructural details. Examples of such value are the current log level, the number of connections waiting in a connection pool and the number of tasks waiting for execution. Often, a monitoring client can monitor state values automatically and send a notification if a value reaches a certain threshold.

### Events

Another concept is the event. A monitoring tool can subscribe to events raised by specific MBeans. When an event occurs, all registered listeners are notified. Events are useful when the application itself knows that something important has happened that might affect the operational status of the application. An example of such an event is the rejection of a Task in a TaskExecutor because the queue is full.

### *Method invocation*

Method invocations are another concept implemented in JMX. It is possible to invoke methods remotely on MBeans. This is useful when the application state has to be modified remotely, without the need of restarting the application. Examples of use cases for method invocations are modifying log levels and modifying the number of threads in a TaskExecutor.

### *Statistics*

Although it is derived from the first, the last concept deserves to be mentioned separately: statistics. Statistics provide information about the state of the application over a predefined time span. Examples are the number of tasks executed in the last hour and their average, minimum and maximum execution time.

### *Common examples of MBeans*

It is impossible to provide a checklist for each application that lists all MBeans that should be exposed, simply because each application is different. The application developer should know what is going on, what the key infrastructural classes are and which metrics give an indication of the health of an application.

However, a number of infrastructure classes that are used in a wide variety of applications are worthwhile mentioning.

### *Connection pools*

The most common example of a connection pool is the database connection pool. No matter what type of connection is in the connection pool, they all share the same basic principle: the number of connections in the pool should match the need of your application, at any moment in time.

The number of free connections inside the pool, as well as the number of threads waiting to obtain one is an extremely valuable metric that gives good insight in the performance of your configuration. When at peak times, only 10% of your connections are used, your connection pool is most likely too big. This means memory and processing power is spent on maintaining connections that are not used anyway. The opposite, however, is a bigger problem. When all available connections are used, and multiple threads are waiting for one, the response times are affected.

Being able to modify the minimum and maximum number of connections, as well as the cleanup timeout for unused connections gives you the ability to tweak the connection pool to cope with the demands of the application.

### *Thread pools and executor services*

Just like connection pools, thread pools and executor services provide the same type of valuable information. When comparing the number of active threads, passive threads and waiting tasks, you have a good insight in the performance impact of your configuration.

Executor services are often used in combination with queues. Some types of queues allow a virtually unlimited number of elements (only limited by the heap size), while others have a

fixed maximum number of elements. Regardless of the type of queue you use, it is valuable to expose the number of elements in the queue as a JMX attribute. Also, consider sending a notification when the queue reaches a certain threshold, such as 80% of queue size. This allows application maintenance to investigate problems before they become critical.

### *Circuit Breakers*

The circuit breaker is a stability pattern that typically blocks calls to unresponsive systems. When a circuit breaker changes state, it typically means that a resource your application relies on is unavailable. Because correctly implemented circuit breakers do not open (which is the state when errors occur) based on a single incident, the state of a circuit breaker is valuable information.

The state change of a circuit breaker is also a useful event to send out a JMX notification for. It allows you to report of fix the resource before users pick up the phone, or worse, just start disappearing.

### *Hibernate Session Factory*

An often-overlooked feature of Hibernate is the `StatisticsService`, which is an MBean provided by Hibernate. All you need to do is instantiate it, set the session factory you use in it and expose it in your MBeanServer.

The `StatisticsService` provides a lot of information, from number of queries executed, their average execution time, to the actual query that took the longest to execute. Some of the information provided by the `StatisticsService` is valuable for application monitoring, while others provide information that allow performance tuning of your application.

### *OSGi container*

OSGi is a technology that is gaining ground in the enterprise Java world. If you use an OSGi container, providing the ability to see the state of bundles as well as start, stop, and updating them is useful. If you can't communicate with the OSGi container, there is no way you can deploy bundles at runtime, which is what OSGi was designed for.

## System MBeans

Java comes with a number of MBeans that expose important parts of the JVM state, such as memory usage information and garbage collection. These beans are automatically registered to the platform MBean server. However, if you have special configuration requirements or use multiple MBean servers, you might be required to register these MBeans explicitly.

These MBeans are accessible using the `ManagementFactory` static factory object. Below is a list of the available MBeans via the `ManagementFactory`. To get a details description of what they do, check their javadoc. These beans are quite extensively documented.

- `ClassLoaderMXBean`
- `MemoryMXBean`
- `ThreadMXBean`
- `RuntimeMXBean`

- CompilationMXBean
- OperatingSystemMXBean
- MemoryPoolMXBean
- MemoryManagerMXBean
- GarbageCollectorMXBean

## Exposing your MBeans

Once you have chosen which object instances provide valuable information about the state of your application, you have to make sure these instances conform to the JMX specification.

The specification requires that all MBean instances consist of an interface, whose name ends on MBean or (since JDK 6) MXBean<sup>1</sup> and an implementation, whose name equals the interface name without the MBean or MXBean suffix.

The interface declares which parts of the bean should be exposed through JMX. Methods that conform to the java bean spec, the well-known getters and setters, define which attributes should be exposed. If a setter method is missing for an attribute, the attribute will be exposed as read only. Any other method will be exposed as a method.

The interface looks as follows. Note that the MXBean suffix was chosen here. When you intend to deploy on java 5 environments, you may use the MBean suffix instead.

```
package com.example.mbeans;

public interface HelloMXBean {

    public void sayHello();
    public int add(int x, int y);

    public String getName();

    public int getAge();
    public void setAge(int age);
}
```

The implementation for this interface looks as follows:

```
package com.example.mbeans;

public class Hello implements HelloMXBean {
    // attributes not shown for brevity

    public void sayHello() {
        System.out.println("hello, world");
    }
}
```

<sup>1</sup> MXBeans are available since java 6 and provide the ability to expose composite information, i.e. multiple properties that are wrapped in a java bean. MBeans can only pass values that the client is able to handle (typically only the primitives and basic objects).

```
public int add(int x, int y) {
    return x + y;
}

public String getName() {
    return this.name;
}

public int getAge() {
    return this.age;
}

public void setAge(int age) {
    this.age = age;
}
}
```

If you wish to emit notifications from this MBean, you should also implement the `NotificationBroadcaster` interface. The easiest way to do this is by extending the `NotificationBroadcasterSupport` class, which provides the basic functionality required by the interface.

Now you can use the `send(Notification n)` method to send notifications to whoever is listening.

Although not strictly required, it is good practice to register the types of notifications a class can emit. To do this, either override the `getNotificationInfo()` method, or use a constructor accepting `MBeanNotificationInfo` instances to register them at construction time.

## Registering your MBeans with the MBean Server

Once you have selected your Beans, they have to be registered with the MBean Server. To do this, you need two things, an instance of the bean itself, and an `ObjectName` instance, which represents the name under which the MBean is registered.

### Defining the MBean Name

The name of an MBean consists of a domain and a number of name-value pairs. Depending on the type of client you use, the domain and name-value pairs depend how the hierarchy of MBeans is displayed. In addition, these properties allow for querying MBeans that have specific values.

Consider the following name:

```
mydomain:type=Connection,name=MasterDatabase
```

This name registers an MBean in the domain “mydomain” with two properties: `type` and `name` with the values “Connection” and “MasterDatabase”, respectively. The general structure for an MBean name is as follows:

```
<domain>:<name>=<value> ( , <name>=<value> )*
```

The domain is separated from the property list by a colon. Therefore, the colon itself cannot be part of the domain name. A property definition consists of a name and a value, separated by an equals sign. When multiple properties are defined, they separated from each other by a comma.

### Finding an MBean server

In order to register MBean with an MBean server, you need to obtain a reference to that MBean server. The recommended way is to register all beans with the Platform MBean Server. This is considered as the main MBean Server, and should be your default choice when registering beans.

A reference to the Platform MBean Server can be obtained by calling:

```
ManagementFactory.getPlatformMBeanServer();
```

If your application architecture requires you to find another MBeanServer instance, you can use the following code to obtain a reference to a list of all MBean Servers known to the JVM:

```
MBeanServerFactory.findMBeanServer(String agentId)
// Use null as agentId to obtain a list of all MBean Servers.
```

If you want to create a new MBean Server, you can use the following code:

```
MBeanServerFactory.createMBeanServer();
```

Note that you are required to expose your MBean server beyond the JVM limits yourself when you create your own MBean server.

### Using the Spring Framework

Spring provides the possibility to create MBeans out of any spring managed bean. This means that you are not required to create an interface for each object you wish to expose as an MBean. Spring takes care of that dynamically.

There are two ways declare your MBeans through the Spring Framework. You can use annotations or specify them explicitly in the Spring application context.

#### Using Annotations

When annotations have your preference, there are a number of annotations of use to you. They are all located in the package `org.springframework.jmx.export.annotation`.

- `@ManagedResource` is a type level annotation, that indicates a spring bean should be exposed as an MBean. You can provide the ObjectName and several other properties using the annotation's attributes. The ObjectName defaults to using the package as domain and the simple class name as the value of the name property.
- `@ManagedOperation` can be used to mark a method to be accessible as a JMX operation.
- `@ManagedOperationParameters` is used as a method level annotation to provide optional documentation for the method parameters.

- `@ManagedAttribute` is a method level annotation that exposes a JMX attribute. The method should be either a getter or a setter. The name of the annotated method defines the name of the attribute. The availability of a getter and/or setter defines whether the attribute can only be read, or also written to.
- `@ManagedNotifications` can be used to document the types of notification this MBean can emit. This annotation is only used for providing meta-data. It does not cause the MBean to automatically emit notifications.

To tell Spring that it should look for these annotations, you should register an `AnnotationMBeanExporter`. Note that using the component-scanning feature of Spring will already cause your MBeans to be detected and exported automatically.

### Declaratively in application context

Another way to expose Spring managed beans as MBeans is using the `MBeanExporter`. The `MBeanExporter` allows you to specify the names of the beans you wish to expose, as well as the `ObjectName` that should be used to expose them, in the application context.

Below is an example configuration of an `MBeanExporter`:

```
<bean id="platformMbeanServer"
      class="java.lang.management.ManagementFactory"
      factory-method="getPlatformMBeanServer"/>

<bean class="org.springframework.jmx.export.MBeanExporter">
  <property name="server" ref="platformMbeanServer"/>
  <property name="beans">
    <map>
      <entry key="someName" value-ref="someBean"/>
      <entry key="someOtherName" value-ref="someOtherBean"/>
    </map>
  </property>
</bean>
```

The role of the bean names `platformMbeanServer` is to force the `MBeanExporter` to use the platform `MBeanServer`. The default option (when the property “server” is not provided) is to autodetect an existing `MBeanServer`, which will normally result in exactly the same behavior.

### Provide external access to your MBean server

Typically, you want MBeans to be available outside of your JVM. The connector is the component that exposes your `MBeanServer` instance to the outside world.

#### Enabling connectors

There are numerous possibilities to create connectors. A commonly chosen way to do this is through RMI, since it only requires JVM startup options. However, there are tools that enable you to expose your server through HTML or SOAP. Since programmatically accessing the `MBeanServer` instance is as easy as a single call to a static method, it is relatively easy to create your own connectors. Do keep in mind that existing monitoring tools might not be able to connect to custom connectors.

Most, if not all, monitoring tools are able to connect to a remote MBeanServer using RMI. To enable the RMI connector for your platform MBean Server, you should specify the following JVM startup parameters:

```
-Dcom.sun.management.jmxremote
```

This creates a RMI connector that is available for local access. To access your connector from remote servers, use the following parameter:

```
-Dcom.sun.management.jmxremote.port=9004
```

In addition to publishing a RMI connector for local access, setting this property publishes an additional RMI connector in a private read-only registry at the specified port using a well-known name, "jmxrmi".

### Security

By default, the RMI connector uses password authentication. These default values look for the password file "jmxremote.password" and access file "jmxremote.access", both located in "JRE\_HOME/lib/management/". Note that the `jmxremote.password` file does not exist by default. The locations of the password and access file can be overridden using the following system properties:

```
com.sun.management.jmxremote.password.file=pw_file_location  
com.sun.management.jmxremote.access.file=xs_file_location
```

To configure JMX security, there are several configuration options that can either be passed in via command line properties or via the `management.properties` file, located in the "JRE\_HOME/lib/management/" folder. To specify another location of the `management.properties` file, use the `com.sun.management.config.file` property.

For a more extensive explanation of using and configuring the RMI connector, visit <http://java.sun.com/javase/6/docs/technotes/guides/management/agent.html>.

To disable authentication altogether use the following system property:

```
com.sun.management.jmxremote.authenticate=false
```

Note that this is highly discouraged in production environments, since it gives anybody with a JMX client access to your application infrastructure.

The most secure way of exposing your MBeans through RMI is using an SSL connection with client authentication. The system properties required for this are as follows:

```
com.sun.management.jmxremote.registry.ssl=true  
com.sun.management.jmxremote.ssl=true  
com.sun.management.jmxremote.ssl.need.client.auth=true  
javax.net.ssl.keyStore=keystore_location  
javax.net.ssl.keyStorePassword=keystore_password  
javax.net.ssl.trustStore=truststore_location
```

```
javax.net.ssl.trustStorePassword=truststore_password
```

The first option protects the RMI registry using SSL. This feature has been introduced in Java 6. The second option is actually the system default, making it optional to specify.

The third property specifies that this SSL connection needs client authentication. The client must be able to encrypt the connection using a key for which there is a certificate in the truststore.

The `javax.net.ssl` properties tell the application which certificates to secure the connection with (keystore), and which client certificates to trust (truststore). These settings are described in the JSSE (Java Secure Socket Extension) specification. These stores are usually password protected.

### Using the Spring Framework

Another option for creating connectors is from your own application. Spring provides the `ConnectorServerFactoryBean`, which allows you to create additional connectors for your MBean Servers declaratively. The most important properties in this `FactoryBean` are `serviceUrl` and `server`.

The `serviceUrl` defines the URL where the service is published. The format of service URL's is defined by the JMX specification:

```
service:jmx:protocol:[host[:port]][url-path]
```

Allowed values for `protocol` are the supported protocols as defined in the `JMXConnectorServerFactory` class. By default, the RMI and IIOP protocols are supported. The `jmxremote-optional.jar` library, which can be downloaded from the Sun website, contains support for additional protocols, such as JMXMP, SASL and TLS.

For example, the Spring configuration for a JMXMP connector would look as follows:

```
<bean class="org.springframework.jmx.support.ConnectorServerFactoryBean">
  <property name="server" ref="platformMbeanServer"/>
  <property name="serviceUrl" value="service:jmx:jmxmp://localhost:9005"/>
</bean>
```

To expose a connector using the RMI protocol, replace the service URL with `service:jmx:rmi:///jndi/rmi://localhost:9005/jmxrmi`

It is important to configure proper security before deploying the application in any production environment. The `environment` property allows you to pass in arbitrary properties to configure the connector. These properties differ per connector and should be included in their documentation.

## Connecting to your application

At this point, your application exposes some MBeans through one or more connectors. Now it is time to configure some tools to connect to that application and monitor it.

### Jconsole

Jconsole is a monitoring tool included in the Sun JDK. It provides a nice graphical user interface to access your MBeans. System MBean information is graphically displayed in graphs, giving a good overview of the state of memory of the monitored application. Jconsole is located in the /bin folder of your jdk root.

By default, jconsole supports the RMI and IIOP connectors. Since the JMXMP, SASL and TLS connectors are provided in an optional jar, you will have to tell jconsole to look in that jar too.

Start jconsole with the following command to specify a folder containing jar files that should be on jconsole's path:

```
jconsole -J-Djava.endorsed.dirs=directory
```

If the jmxremote-optional.jar file is located in that folder, you have the ability to connect to your application using a JMXMP connector.

If you have configured SSL security for your connector, some additional properties are required:

```
-J-Djavax.net.ssl.trustStore=truststore  
-J-Djavax.net.ssl.trustStorePassword=trustword
```

If client authentication is also required, provide these properties too:

```
-Djavax.net.ssl.keyStore=keystore  
-Djavax.net.ssl.keyStorePassword=password
```

### JManage

Jconsole is a nice tool to access MBeans, but it is rather a developer's tool. It is not suitable to monitor the state of multiple applications, since it does not reconnect after disconnecting and is not able to warn you when certain attribute values exceed a certain threshold.

The marketplace offers a wide variety of application that can do monitoring of applications and that will warn you through email or some other mechanism when predefined conditions are (or are not) met. JManage is an open source application what provides this ability.

More information about Jmanage is found at [www.jmanage.org](http://www.jmanage.org).

### Programmatically

If you have built a custom monitoring application or have some other reason to connect to your application through JMX programmatically, there are some classes to help you connect to any JMX connector.

`JMXConnectorFactory.connect(JMXServiceUrl serviceUrl)` will connect to a remote JMX connector using the service URL specified. The static `newPlatformMXBeanProxy(...)` method in the `ManagementFactory` class can be used to create proxies for remote MBeans. This allows you to call MBeans as if they resided in the local JVM. To obtain the `MBeanServerConnection`, call `getMBeanServerConnection()` on a `JMXConnector` instance.

Spring allows you to create an `MBeanServerConnection` declaratively, using the `MBeanServerConnectionFactoryBean`. The configuration of this factory bean is straightforward.

Here is an example:

```
<bean
  class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">
  <property name="serviceUrl" value="service:jmx:jmxmp://localhost:9005"/>
</bean>
```

The `MBeanServerConnection` interface itself provides easy access to JMX functionality. In fact, the `MBeanServer` interface extends this interface.

## Conclusion

Companies often financially rely on their applications. In these cases, monitoring the state of applications is crucial to be able to react to relevant events quickly. Good monitoring allows problems to be detected before they become critical.

In this document, I have given guidelines to find the objects that may provide valuable information about the health of an application. Next, I have shed some light on how these instances can be exposed as MBeans in a relatively non-intrusive way either programmatically or using the Spring Framework. Finally, I have shown how to gain remote access to those MBeans with `JConsole` or from your own application.