JAVA MANAGEMENT EXTENSIONS FOR BUSINESS APPLICATIONS

Dănuţ-Octavian SIMION
Athenaeum University, Economic Informatics Department,
Bucharest, Romania
danut_so@yahoo.com

Abstract.
The paper presents the Java Management Extensions (JMX) which was designed to address the management needs of business applications written for the Java platform and to be compatible with existing management standards, such as the Simple Network Management Protocol (SNMP), which is the standard for management of enterprise networks. It was also designed so that instrumentation of resources to put them under the control of a management application. The JMX architecture enables Java applications (or systems) to become manageable and so the resources become available for management.

Keywords: JMX, MBeans, Java Virtual Machine (JVM), management applications, UML diagram.

1. Introduction

Java Management Extensions (JMX) is the result of the Java Community Process (JCP) and Java Specification Request (JSR) 3 designed to deal with the growth of large-scale distributed applications. Mission-critical business applications have evolved from a sequence of programs running on a single computer to business components running on different machines scattered throughout a network. The difficulty of managing distributed systems has increased along with the complexity of those systems. When considering a management solution for enterprise applications, some issues arise like which management solution is best for the application, what standards should a management solution follow and how much effort is required to enable the components of the application to be managed [4], [5], [1]. JMX was designed so that instrumentation of

resources to put them under the control of a management application. A resource is any entity in the system that needs to be monitored and/or controlled by a management application, so resources that can be monitored and controlled are called manageable.

2. JMX Architecture

JMX architecture contains three levels. The level closest to the application is called the instrumentation level. This level consists of four approaches for instrumenting application and system resources to be manageable (making them managed beans, or MBeans), as well as a model for sending and receiving notifications. The middle level of the JMX architecture is called the agent level. This level contains a registry for handling manageable resources (the MBean server) as well as several agent services, which themselves are MBeans and thus are manageable. The combination of an instance of the MBean server, its registered MBeans, and any agent services in use within a single Java Virtual Machine (JVM) is typically referred to as a JMX agent. The third level of the JMX architecture is called the distributed services level. This level contains the middleware that connects JMX agents to applications that manage them (management applications) [2], [1], [3]. This middleware is broken into two categories: protocol adaptors and connectors. Through a protocol adaptor, an application such as a web browser can connect to one or more JMX agents and manage the MBeans that are registered within it.

2.1. The Instrumentation Level

JMX instrumentation level is the level that should be of most concern to developers, because this level prepares resources to be manageable. Fig. 1 shows the two areas of concern for the instrumentation level of the JMX architecture:

- Application resources, or even the application itself;
- The instrumentation strategy that is used to instrument application resources.

An application resource that is to be manageable through JMX must provide information about five of its features:

- Attributes, which contain the state of the resource;
- Constructors, which are used by management applications and other JMX agents to create instances of the resource;

- Operations, which may be invoked by a management application or other JMX agent to cause the resource to perform some action;
- Parameters to constructors and operations;
- Notifications, which are emitted by the resource and sent via the JMX notification infrastructure to any interested agents.

The combination of these pieces of information, or metadata - about a resource's features is known as its management interface. It is through this interface alone that a management application or other JMX agent may interact with a resource [4], [5]. For example if a resource called *ExampleResource* is available, than it has the following attributes:

- Version: the version of the *ExampleResource*;
- ProcessingTime: the number of milliseconds of processing time that have been consumed by this instance of *ExampleResource*;
- NumberOfExceptions: the total number of exceptions that have been thrown by this instance of *ExampleResource* in the course of its processing.

An implementation of the state of *ExampleResource* would look like Example 1.

Example 1: Attributes of a resource

```
public class ExampleResource {
// Version : read-only
private String version_ex = "1.0.1";
public String getVersion() {
return version_ex;
}
// ProcessingTime : read-only
private long processingTime_ex;
public long getProcessingTime() {
return processingTime_ex;
}
// NumberOfExceptions : read-write
private short numberOfExceptions_ex;
public short getNumberOfExceptions() {
return numberOfExceptions_ex;
}
public void setNumberOfExceptions(short value) {
numberOfExceptions_ex = value;
} }
```

This example demonstrates the fundamentals of instrumenting the attributes of a resource according to the JavaBeans state pattern. Each attribute is backed by a private member variable, so that the part of the resource's state represented by that attribute cannot be accessed directly [5], [3]. All attributes in this example are readable and have corresponding getters. The

NumberOfExceptions attribute is writable, and it provides a setter for that purpose.

Standard MBeans

Standard MBeans are the simplest type of MBean to code. It is necessary to define the MBean interface as a Java interface and implement that interface on the resource MBean. If the instrument is *ExampleResource* (from Example 1) as a standard MBean, it must be defined a Java interface like this:

```
public interface ExampleResourceMBean {
// Version : read-only
public String getVersion();
// ProcessingTime : read-only
public long getProcessingTime();
// NumberOfExceptions : read-write
public short getNumberOfExceptions();
public void setNumberOfExceptions(short value);
}
```

After that it must be implemented the interface on the *ExampleResource* class:

```
public class ExampleResource implements ExampleResourceMBean {
// ……
}
```

The name assigned to this interface is important (it must be the name of the class that implements it), followed by MBean. That is the instrumentation code that must be written to make *ExampleResource* capable of being managed. To add a method, reset(), to reset the state of the ProcessingTime and NumberOfExceptions attributes. It must be added this method to the MBean interface, as the following code:

```
public interface ExampleResourceMBean {
// Version : read-only
public String getVersion();
// ProcessingTime : read-only
public long getProcessingTime();
// NumberOfExceptions : read-write
public short getNumberOfExceptions();
public void setNumberOfExceptions(short value);
// reset() operation
public void reset();
}
```

After that it must be implemented the method on the *ExampleResource* class, as shown in Example 2.

Example 2 : The ExampleResource managed bean

```
public class ExampleResource {
// Version : read-only
private String version_ex = "1.0.1";
public String getVersion() {
return version_ex;
}
// ProcessingTime : read-only
private long processingTime_ex;
public long getProcessingTime() {
return processingTime_ex;
}
// NumberOfExceptions  : read-write
private short numberOfExceptions_ex;
public short getNumberOfExceptions() {
return numberOfExceptions_ex;
}
public void setNumberOfExceptions(short value) {
numberOfExceptions_ex = value;
}
public void reset() {
processingTime_ex = 0;
setNumberOfExceptions(0);
}
//  ……
}
```

The metadata required of every MBean is created automatically by the JMX infrastructure for standard MBeans. Before an MBean can be managed, it must be registered with a JMX agent. When a standard MBean is registered, it is inspected, and metadata placeholder classes are created and maintained by the JMX agent on behalf of the MBean [4], [2]. The Java reflection API is used to discover the constructors on the MBean class, as well as other features. The attribute and operation metadata comes from the MBean interface and is verified by the JMX agent.

Dynamic MBeans

In the case of standard MBeans, the JMX agent creates the metadata that describes the features of a resource. In contrast, the developer must provide the metadata that describes a resource as a dynamic MBean. Dynamic MBeans implement a JMX interface called DynamicMBean that contains methods that allow the JMX agent to discover the management interface of the resource at runtime [5], [1].The DyamicMBean interface is defined in Example 3.

Example 3 : the DynamicMBean interface

```
package javax.management;
```

```
public interface DynamicMBean {
public Object getAttribute(String attribute)
throws AttributeNotFoundException, MBeanException,
ReflectionException;
public void setAttribute(Attribute attribute)
throws AttributeNotFoundException, InvalidAttributeValueException,
MBeanException, ReflectionException;
public AttributeList getAttributes(String[] attributes);
public AttributeList setAttributes(AttributeList attributes);
public Object invoke(String actionName, Object params[], String
signature[])
throws MBeanException, ReflectionException;
public MBeanInfo getMBeanInfo();
}
```

To describe the management interface of a resource as a dynamic MBean, it must be described five fundamental pieces of metadata that correspond to its five fundamental features: constructors, attributes, parameters, operations and notifications. As shown in Fig. 2, these five pieces of metadata are described through instances of MBeanConstructorInfo, MBeanAttributeInfo, MBeanParameterInfo, MBeanOperationInfo and MBeanNotificationInfo. The parameters that are passed to a constructor or operation must also be described through the JMX metadata class MBeanParameterInfo. Once all the metadata for an MBean has been described through these classes, it is contained in a single metadata class – MbeanInfo - that describes the MBean interface [2], [1]. The JMX agent uses the getMBeanInfo() method of the DynamicMBean interface to obtain this MBeanInfo object in order to discover the management interface of a dynamic MBean. Once the management interface has been described, the JMX agent uses the other methods of DynamicMBean to retrieve and set attribute values and invoke operations on the MBean.

Open MBeans

Using the standard, dynamic, or model MBean instrumentation approaches allows to describe MBean features (attributes, constructors, parameters, operations, and notifications) that are one of the following types:

- A fundamental Java type, such as boolean, char, long, or float, through its corresponding JDK wrapper - Boolean, Char, Long, or Float;
- A string, as java.lang.String;
- An array of fundamental types or strings.

Open MBeans were designed in an effort to make MBeans accessible to the widest possible range of management applications. It can be used complex types on the management interface of standard, dynamic, and model MBeans. For a management application to correctly interpret the state of those types, the classes representing those types must be available to the management application. The result is a coupling between the management application and the resources it manages, compromising the maintainability of the underlying managed resources [4], [2], [1]. To describe an open MBean attribute, it must be used the OpenMBeanAttributeInfo interface, which is implemented by a support class called OpenMBeanAttributeInfoSupport. Each support class extends its dynamic MBean counterpart. OpenMBeanAttributeInfoSupport extends MBeanAttributeInfo.

Model MBeans

The Model MBeans are the most powerful type of MBean. Instrumenting applications resources as model MBeans provides with the most features and flexibility of any of the MBean types that are fully specified by the JMX specification. Model MBeans work, includes a Descriptor class and the metadata classes that are used by resources instrumented as model MBeans [4], [5]. It also includes a RequiredModelMBean, a model MBean class that is required to be present in every JMX implementation. Model MBeans are dynamic MBeans and so use metadata to describe the features of the MBean. When a management application manages the MBean, it simply uses this information to call back into the resource [4], [2]. A second benefit of model MBeans is the feature set that comes along with them. Model MBeans have a rich set of features, including support for:

- **Automatic attribute-change notifications;**
- **Persistence of the MBean's state at predefined intervals;**
- **Logging of certain significant events in state changes of the MBean;**
- **Accessing MBean state from a cache to improve performance for attributes whose values have a relatively long freshness.**

A third benefit of model MBeans is that the resources that are instrumented do not require any code changes. A logical place for this code is in the resource itself, but JMX does not require this. Like all other MBean types, model MBeans must be created and registered with the MBean server, and, as with dynamic MBeans, the management interface of resource is exposed

through metadata classes [4], [1], [3]. Every compliant JMX implementation must ship a class called RequiredModelMBean. This class is instantiated and registered with the MBean server, and it implements several interfaces, including DynamicMBean. The key difference between instrumenting a resource as a model MBean versus a dynamic MBean is the Descriptor interface. A class implementing this interface describes certain properties of the MBean to the agent level of the JMX architecture, other MBeans, and management applications. Each descriptor contains one or more fields, which have corresponding Object values. In order to exploit the full power of a descriptor, the management application must know about model MBeans. If the management application knows nothing of model MBeans, it simply treats the MBean as it would any other. Example 4 shows the Descriptor interface.

Example 4 : the Descriptor interface

```
public interface Descriptor_ex extends java.io.Serializable {
public Object getFieldValue(String fieldName) throws
RuntimeOperationsException;
public void setField(String fieldName, Object fieldValue)
throws RuntimeOperationsException;
public String[] getFields();
public String[] getFieldNames();
public Object[] getFieldValues(String[] fieldNames);
public void removeField(String fieldName);
public void setFields(String[] fieldNames, Object[] fieldValues)
throws RuntimeOperationsException;
public Object clone() throws RuntimeOperationsException;
public boolean isValid() throws RuntimeOperationsException;
}
```

The Descriptor interface revolves around the idea of a field. A field is a name/value pair in which the name is a String that contains the name of the field and the value is an Object that is the value of the field. Fields have two uses. First, field values are used internally by the JMX implementation, for example, to determine when to retrieve the value of an attribute from the internal cache or when to invoke the getter for that attribute. The second use of Descriptor fields is to provide more information to the agent level or a management application about an MBean or one of its features. This feature can then be exploited by any agent or management application that is aware of model MBeans [2], [1]. None of the fields used by the agent level or management applications are required for JMX compliance and some of those fields have constraints on the possible values they may have.

3. Conclusions

In most business applications the manageable concept is very important because offers the better usage of Java objects, modules and other features. JMX architecture through different types of MBeans and services enhance the business logic of applications build on Java platform [4], [5]. At the base of JMX are UML diagrams that are showing the relationships between the MBeans, metadata classes and the corresponding interfaces and so the programmer has better options in choosing the right type of the Java components. The most essential components of the JMX are the MBeans which are split in categories such as Standard MBeans, Dynamic MBeans, Open MBeans and Model MBeans and they are making business applications more flexible and more manageable regarding the logic of them [2], [5]. JMX is working with applications resources, instrumentation strategy, attributes, constructors and operations on Java platform and so objects are more easy to use and to change regarding the business logic of applications.

References

[1] Todd Bowker, "Superior app management with JMX", JavaWorld.com 2001

[2] Marc Fleury, Juha Lindfors, "Enabling Component Architectures with JMX", O'Reilly, 2001

[3] Brian Goetz, "Instrumenting applications with JMX", IBM-developerWorks, 2006

[4] Kreger, Harold, Ward, Williamson, "Java and JMX: Building Manageable Systems", Addison Wesley, 2002

[5] Qusay H.M., "Getting Started with Java Management Extensions (JMX): Developing Management and Monitoring Solutions", Sun Developer Network, 2004

URI: http://javaworld.com/javaworld/
URI: http://jmxracing.citymax.com/
URI: http://servletsuite.com/servlets/jmxtag.htm
URI: http://pub.admc.com/howtos/jmx/