Craig Walls
Norman Richards

XDoclet IN ACTION

MANNING

For online information and ordering of this and other Manning books, go to www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact:

Special Sales Department
Manning Publications Co.
209 Bruce Park Avenue
Greenwich, CT 06830
Fax: (203) 661-9018
email: orders@manning.com

Manning Publications Co.
209 Bruce Park Avenue
Greenwich, CT 06830

Copyeditor: Tiffany Taylor
Typesetter: Denis Dalinnik
Cover designer: Leslie Haimes

ISBN 1-932394-05-2

Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – VHG – 07 06 05 04 03

brief contents

9 XDoclet and JMX

This chapter covers

- Generating MBean interfaces
- Working with mlets
- JBossMX services and XMBeans
- Describing MX4J MBeans

After one look at this planet any visitor from outer space would say "I want to see the manager."

—William S. Burroughs (1914–1997)

Instrumenting systems for management and reporting is an often-overlooked aspect of software development. For example, suppose that after you deployed the web-log application you started in chapter 3, you decided that you need to be able to expose some simple management functions (such as deleting a blog) as well as some server health functions (such as reporting the amount of available memory).

Developing management interfaces is often a reaction to a problem rather than a proactive movement to prevent (or at least predict) problems. A probable reason for this attitude toward software management is that writing management code typically isn't as much fun as writing the main application code. Furthermore, it's usually hard to write a good software-management solution. When something is difficult and not fun, it tends to not get done.

All that changed with the introduction of Java Management Extensions (JMX). JMX makes it simple to instrument an application for management. But, as with many Java technologies, redundancy exists between the artifacts that make up a JMX management interface. And, as is the theme of this book, redundancy translates into an opportunity for XDoclet to take on some of the development work.

In this chapter, we'll look at some of these redundancies and explain how XDoclet can handle them for you. As we do so, you'll develop a management interface for the web-log application that will enable you to report on how the application is being used.

9.1 A quick JMX overview

JMX is an optional extension to the Java 2 Standard Edition that defines the tools, specifications, services, and APIs necessary for developing management and monitoring solutions in Java. Using JMX, it's possible to instrument applications (both new and legacy systems) for management, monitoring, and configuration. (You can download JMX and read more about it at http://java.sun.com/products/JavaManagement.)

The key component of an application that is instrumented for management using JMX is the MBean (short for Managed Bean). An *MBean* is a JavaBean that exposes certain methods that define the management interface. There are three types of MBeans:

- *Standard MBeans* expose managed operations using a fixed interface. They're good when the management interface is unlikely to change.

- *Dynamic MBeans* are useful when the managed resource's API changes frequently. By exposing the management interface via a metadata class, you shield the MBean's client from the ever-changing management interface.
- *Model MBeans* take the notion of dynamic MBeans a step further. Instead of defining the MBean's management interface using a metadata class, you declare the management interface in an external resource (perhaps an XML file). The MBean server then uses the external resource to create the management interface on the fly at runtime.

We'll focus much of our attention in this chapter on how to use XDoclet to generate MBean interfaces and other artifacts associated with MBean development; we won't dwell on the details of JMX. To learn more about JMX, we recommend that you read *JMX in Action*.[1]

9.2 Preparing the build for JMX generation

The `<jmxdoclet>` task is the key to generating artifacts associated with JMX. Like other XDoclet tasks, `<jmxdoclet>` has several subtasks that indicate specific types of artifacts that should be generated. Table 9.1 describes the subtasks of `<jmxdoclet>`.

Table 9.1 Subtasks of `<jmxdoclet>` and the types of files they generate

Subtask	Description
`<mbeaninterface>`	Generates a managed operation interface for standard MBeans
`<mlet>`	Generates an mlet descriptor file
`<jbossxmbean>`	Generates a deployment descriptor for JBossMX model MBeans (XMBeans)
`<jbossxmldoc>`	Generates documentation for an MBean in DocBook format
`<jbossxmlservicetemplate>`	Generates JBossMX service deployment descriptors
`<mx4jdescription>`	Generates an MBean description class for deployment in MX4J

The first thing you need to do is create a new build file specific to JMX generation. Listing 9.1 shows `build-jmxgen.xml`, the build file that uses `<jmxdoclet>` and its subtasks to generate artifacts for MBeans.

[1] Benjamin G. Sullins and Mark B. Whipple, *JMX in Action* (Greenwich, CT: Manning, 2002).

Listing 9.1 The build-jmxgen.xml build file used to generate JMX artifacts

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

<project name="Blog" default="generate-mock" basedir=".">
  <path id="xdoclet.lib.path">
    <fileset dir="${lib.dir}" includes="*.jar"/>
    <fileset dir="${xdoclet.lib.dir}" includes="*.jar"/>
  </path>

  <target name="generate-jmx">
    <taskdef name="jmxdoclet"
        classname="xdoclet.modules.jmx.JMXDocletTask"
        classpathref="xdoclet.lib.path"/>

    <jmxdoclet destdir="${gen.src.dir}" mergeDir="${merge.dir}">
      <fileset dir="${src.dir}">
        <include name="**/jmx/*.java"/>
      </fileset>

      <mbeaninterface/>
      <mlet/>
    </jmxdoclet>
  </target>
</project>
```

Generate interfaces for standard MBeans

Generate mlet deployment files

Notice that the `<fileset>` you use limits the scope of `<jmxdoclet>` to classes in packages whose name ends with `.jmx`. The build will run faster because `<jmxdoclet>` won't have to process all classes looking for `@jmx.mbean` tags.

Next you must call `build-jmxgen.xml` from your main build file. An `<ant>` task is needed in the `generate` target of `build.xml`:

```
<target name="generate">
  <ant antfile="build-ejbgen.xml" target="generate-ejb"/>
  <ant antfile="build-webgen.xml" target="generate-web"/>
  <ant antfile="build-jmxgen.xml" target="generate-jmx"/>
</target>
```

With the build files ready for working with JMX, let's get started by having XDoclet generate interfaces for your MBeans.

9.3 Generating MBean interfaces

Suppose you want to expose some of the management functions of the web-log application as managed operations through JMX. The `BlogFacade` EJB that you wrote in chapter 3 already implements the functions you'd like to expose as

managed operations. The quickest way to do this is to write a standard MBean that wraps the `BlogFacade` EJB. Listing 9.2 shows such an MBean.

Listing 9.2 Standard MBean that wraps BlogFacade to expose management functions

```
package com.xdocletbook.jmx;

import java.util.List;
import com.xdocletbook.blog.interfaces.BlogFacade;
import com.xdocletbook.blog.interfaces.BlogFacadeHome;
import com.xdocletbook.blog.util.BlogFacadeUtil;

/**
 * @jmx.mbean                                          ❶ Declare this as an MBean
 *     name="BlogManager"                                named BlogManager
 */
public class BlogManager implements BlogManagerMBean {
  BlogFacadeHome home;

  public BlogManager() throws Exception {
    home = BlogFacadeUtil.getHome();
  }

  /**
   * @jmx.managed-operation                                        ◁
   */
  public String getBlogName(String blogId) throws Exception {
    BlogFacade facade = home.create();
    return facade.getBlogSimple(blogId).getName();
  }

  /**                                                              ❷ Include these
   * @jmx.managed-operation                                       methods in
   */                                                             the MBean
  public void deleteBlog(String blogId) throws Exception {        interface
    home.create().deleteBlog(blogId);
  }

  /**
   * @jmx.managed-operation                                        ◁
   */
  public List getAllBlogs() throws Exception{
    BlogFacade facade = home.create();
    return facade.getAllBlogs();
  }
}
```

Standard MBeans are composed of two parts:

- An interface that declares methods to be exposed as managed operations
- A class that implements the interface, fleshed out with code that performs the operations

`BlogManager.java` defines the implementation class for the `BlogManager` MBean. Notice that we haven't shown you the interface it implements; that's because you'll let XDoclet generate the interface.

You tag `BlogManager` at the class level with `@jmx.mbean` to identify this class to XDoclet as an MBean named `BlogManager` ❶. Then you tag each method with `@jmx.managed-operation` to tell XDoclet to include these methods in the generated MBean interface ❷.

When you run the build and the `<mbeaninterface>` subtask processes `BlogManager.java`, the following interface is generated:

```
package com.xdocletbook.jmx;

/**
 * MBean interface.
 */
public interface BlogManagerMBean {

  java.lang.String getBlogName(java.lang.String blogId)
      throws java.lang.Exception;

  void deleteBlog(java.lang.String blogId)
      throws java.lang.Exception;

  java.util.List getAllBlogs() throws java.lang.Exception;

}
```

To kick off the `BlogManager` MBean, write `JMXStartupListener.java` (listing 9.3), a context listener that will be deployed with the web-log application and will start up the MBean server and register the `BlogManager` MBean with the MBean server. It also starts the HTML adapter so that you can view and invoke the `BlogManager` MBean through a web browser.

Listing 9.3 JMXStartupListener.java starts JMX services within a web container.

```
package com.xdocletbook.blog.listener;
import javax.management.MBeanServer;
import javax.management.MBeanServerFactory;
import javax.management.ObjectName;
```

```
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import com.sun.jdmk.comm.HtmlAdaptorServer;
import com.xdocletbook.jmx.BlogManager;

/**
 * @web.listener
 */
public class JMXStartupListener implements ServletContextListener {
  private MBeanServer server;
  private HtmlAdaptorServer htmlAdaptor;

  public void contextDestroyed(ServletContextEvent arg0) {
    System.out.println("STOPPING MBEAN SERVER");
    htmlAdaptor.stop();
  }

  public void contextInitialized(ServletContextEvent arg0) {
    System.out.println("STARTING MBEAN SERVER");

    server = MBeanServerFactory.createMBeanServer("BlogAgent");

    try {
      startBlogManager();
      startHtmlAdaptor();
    } catch (Exception e) {
      System.err.println("Exception:  "+e);
    }
  }

  private void startBlogManager() throws Exception {
    server.registerMBean(new BlogManager(),
        new ObjectName("BlogAgent:name=blogManager"));
  }

  private void startHtmlAdaptor() throws Exception {
    htmlAdaptor = new HtmlAdaptorServer();
    htmlAdaptor.setPort(9092);

    server.registerMBean(htmlAdaptor,
        new ObjectName("Server:name=HtmlAdaptor"));
    htmlAdaptor.start();
  }
}
```

Add this class as a listener in web.xml

Register BlogManager MBean with MBean server

Register and start HTML Adaptor

With the HTML Adaptor started, you can point your web browser to http://localhost:9092 and see the agent view, as shown in figure 9.1.

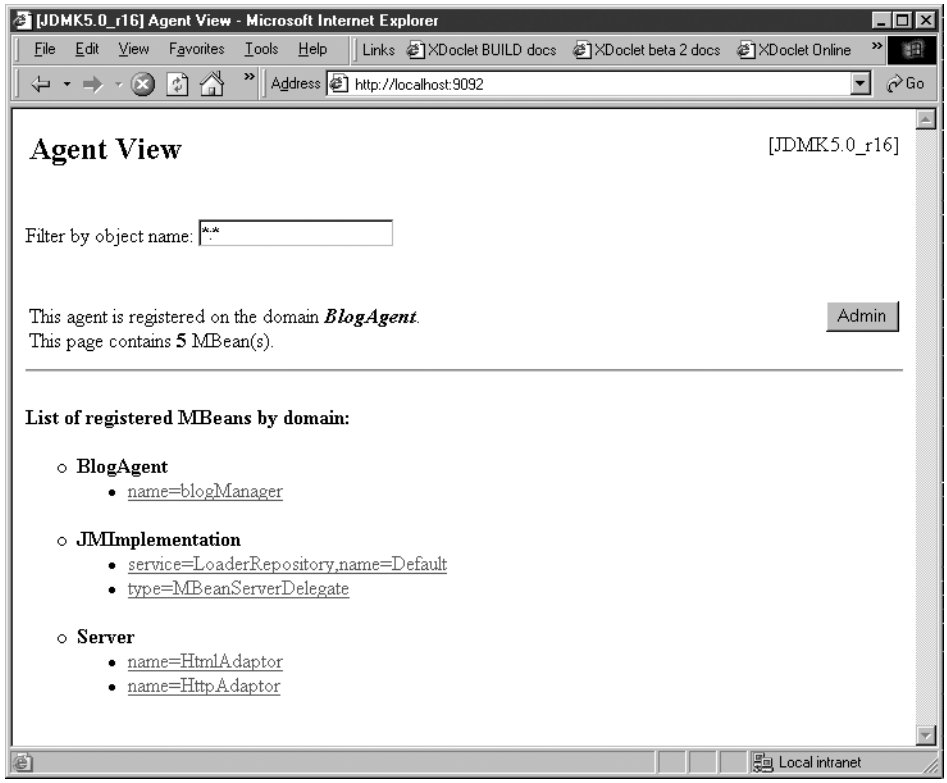

Figure 9.1 The agent view shows the list of registered MBeans by domain.

Clicking on the name=blogManager link under the BlogAgent domain, you can access the `BlogManager` MBean through its HTML interface, as shown in figure 9.2.

From the HTML interface, you can interact with the `BlogManager` MBean and manage your application.

9.4 Generating mlet files

Suppose that while your application is running, you encounter some performance problems and suspect that there's a memory leak. To keep an eye on the memory consumption, you write a `MemoryMonitor` MBean (listing 9.4).

But suppose you don't want to permanently deploy the `MemoryMonitor` MBean with your application. Furthermore, it'd be great to deploy this bean on the go without having to restart the application. How can you do this?

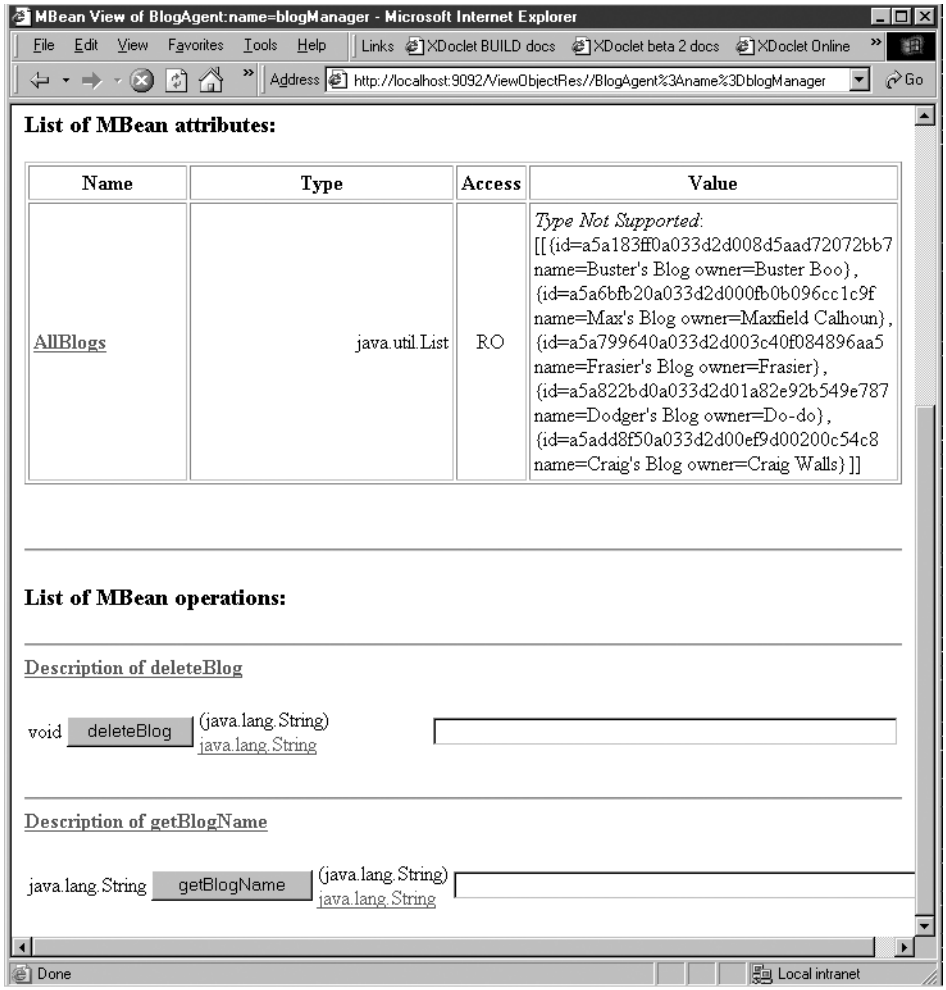

Figure 9.2 The `BlogManager` MBean's HTML interface as presented by the HTML Adaptor

The JMX specification provides a mechanism for hot deployment of MBeans through the mlet (short for management applet) service. The mlet service downloads and installs MBeans from remote locations using an XML-like mlet file that describes the MBean.

Listing 9.4 The MemoryMonitor MBean keeps a watchful eye on available memory.

```
package com.xdocletbook.jmx;

/**
 * @jmx.mbean
 *     name="Memory:name=memoryMonitor"
 *
 * @jmx.mlet-entry                              Add this MBean
 *     archive="MemoryMonitorMBean.jar"         to the mlet file
 */
public class MemoryMonitor implements MemoryMonitorMBean{
  public MemoryMonitor() {
  }

  /**
   * @jmx.managed-attribute
   */
  public long getAvailableMemory() {
    return Runtime.getRuntime().freeMemory();
  }

  /**
   * @jmx.managed-attribute
   */
  public long getTotalMemory() {
    return Runtime.getRuntime().totalMemory();
  }

  /**
   * @jmx.managed-attribute
   */
  public long getMaxMemory() {
    return Runtime.getRuntime().maxMemory();
  }
}
```

You use the `<mlet>` subtask of `<jmxdoclet>` to have XDoclet generate the mlet deployment file for you:

```
<jmxdoclet destdir="${gen.src.dir}" mergeDir="${merge.dir}">
  <fileset dir="${src.dir}">
    <include name="**/jmx/*.java"/>
  </fileset>

  <mbeaninterface/>
  <mlet/>
</jmxdoclet>
```

When the `<mlet>` subtask processes `MemoryMonitor.java`, the mlet deployment file, named `mbeans.mlet`, is generated:

```
<!--
  Generated file - Do not edit!
-->

<MLET NAME="Memory:name=memoryMonitor"
    CODE="com.xdocletbook.jmx.MemoryMonitor"
    ARCHIVE="MemoryMonitorMBean.jar">
</MLET>
```

MBean name

MBean implementation class

JAR file containing the MBean

If you've ever written a Java applet, the format of the `mbeans.mlet` file should be familiar. The `<MLET>` tag works for MBeans in a way similar to how the `<APPLET>` tag works for applets.

The `<mlet>` subtask uses meta-information declared with the `@jmx.mbean` and `@jmx.mlet-entry` tags to generate the mlet deployment file. In the case of `MemoryMonitor.java`, the `name` attribute of `@jmx.mbean` maps to the `NAME` attribute of `<MLET>`. And you use the `archive` attribute of `@jmx.mlet-entry` to declare that the agent can download the `MemoryMonitor` MBean as a JAR file called `MemoryMonitorMBean.jar`.

Optionally, you can instruct the agent to download the MBean one class file at a time by changing the `@jmx.mlet-entry` tag, replacing the `archive` attribute with the `codebase` attribute:

```
/**
 * @jmx.mlet-entry
 *     codebase="mlets/memory"
 */
public class MemoryMonitor implements MemoryMonitorMBean {
...
}
```

This change results in the following `mbeans.mlet` deployment file being generated:

```
<!--
  Generated file - Do not edit!
-->

<MLET NAME="Memory:name=memoryMonitor"
    CODE="com.xdocletbook.jmx.MemoryMonitor"
    CODEBASE="mlets/memory">
</MLET>
```

The `CODEBASE` attribute indicates that the MBean's class files can be downloaded from the `mlet/memory` path relative to the path of the `mbeans.mlet` file.

9.4.1 Deploying the mlet using the mlet service

To try out the `MemoryMonitor` MBean, you need to make sure the mlet service is registered in the MBean server. `MletAgent` (listing 9.5) starts the `HtmlAdaptorServer` and registers the mlet service to enable hot deployment of MBeans.

Listing 9.5 MletAgent.java registers the mlet service.

```
package com.xdocletbook.jmx;
import javax.management.MBeanServer;
import javax.management.MBeanServerFactory;
import javax.management.ObjectName;
import com.sun.jdmk.comm.HtmlAdaptorServer;

public class MletAgent {
  public static void main(String[] args) throws Exception {
    MBeanServer server =
        MBeanServerFactory.createMBeanServer("MletAgent");
    HtmlAdaptorServer htmlAdaptor = new HtmlAdaptorServer();
    htmlAdaptor.setPort(9092);

    ObjectName htmlName =
        new ObjectName("MletAgent:name=HtmlAdaptor");
    ObjectName mletName = new ObjectName("MletAgent:name=mlet");

    server.registerMBean(htmlAdaptor, htmlName);
    server.createMBean("javax.management.loading.MLet", mletName);

    htmlAdaptor.start();
  }
}
```

Register mlet service MBean

Once `MletAgent` is running, navigate to the mlet service by visiting http://localhost:9092 and then clicking the name=mlet link. If you scroll down the page, you'll see the `getMBeansFromURL` operation. Enter the URL to the `mbeans.mlet` file into the text field, as shown in figure 9.3—in our case, it's in the root of the D: drive.

Finally, click the getMBeansFromURL button to deploy the MBean. If the operation is successful, you should see a page that looks like figure 9.4.

Now let's look at what the `MemoryManager` MBean can tell you. Navigate back to the list of deployed agents by clicking the Back to Agent View link. You should now see a name=memoryMonitor link under the Memory domain. Clicking it yields the screen you see in figure 9.5.

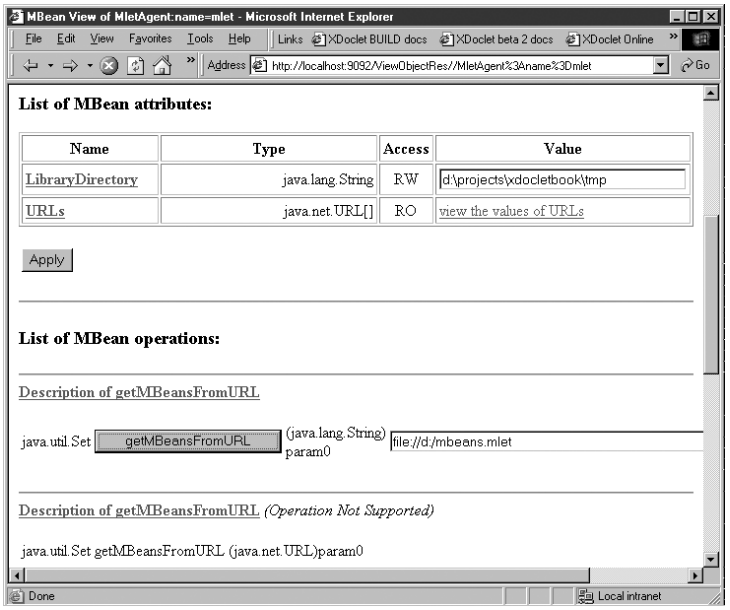

Figure 9.3 Dynamically deploying the `MemoryMonitor` MBean using the mlet service

Now you can watch the `AvailableMemory` attribute to gain insight on memory usage. Changing the reload period (near the top) to a nonzero value will

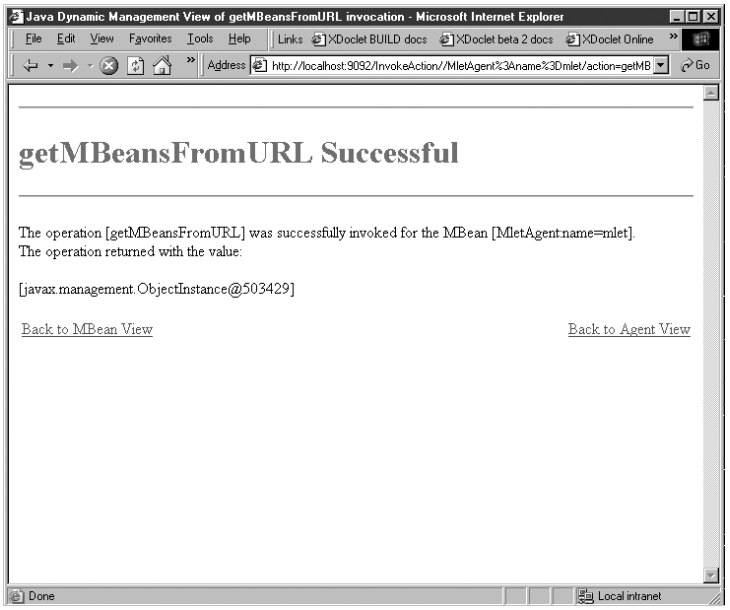

Figure 9.4 The `MemoryMonitor` MBean is successfully deployed.

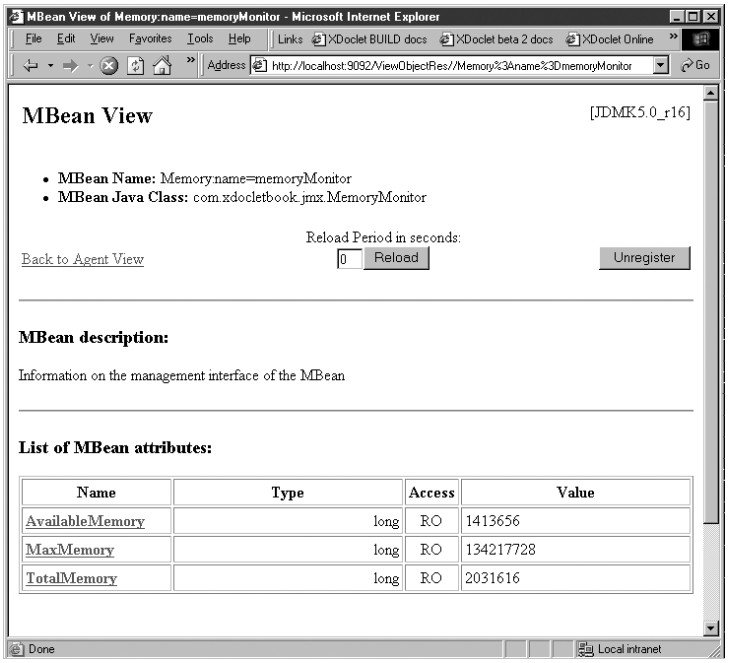

Figure 9.5 Monitoring memory usage with the `MemoryMonitor` MBean

cause the screen to be reloaded periodically, making it easier to monitor memory usage.

So far, you've seen how XDoclet can help generate MBean interfaces and mlet deployment files. Now let's look at how XDoclet can help you generate artifacts that support extensions to the JMX specification, starting with JBossMX and a special implementation of a model MBean called an XMBean.

9.5 Working with MBean services in JBossMX

Arguably the most well known JMX application is the JBoss application server. At JBoss's core is JBossMX, a JMX server that hosts a collection of MBean services that make up the application server functionality. As a result of this architecture, JBoss is not only a great application server but also instrumented for management through JMX.

To see JBoss's JMX instrumentation in action, make sure JBoss is running and then point your browser to http://{hostname}:8080/jmx-console. Figure 9.6 shows the JBoss JMX console that lists all the MBean services deployed in JBossMX.

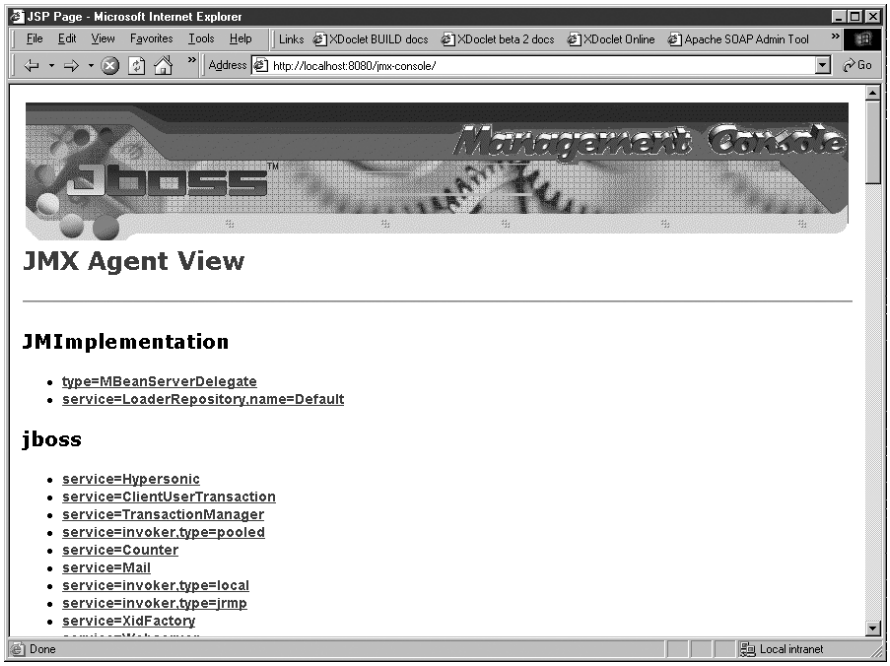

Figure 9.6 The JBoss JMX console lets you manage JBoss services through a web interface.

In addition to the MBean services that make up the JBoss application server, you can also deploy your own MBean services in JMX. Let's look at how XDoclet can help you turn the `MemoryMonitor` MBean into an MBean service suitable for deployment in JBossMX.

9.5.1 Creating JBossMX services

JBossMX services are deployed in a service archive (SAR) file. A SAR file is nothing more than a JAR file with an XML service deployment descriptor in the JAR's `META-INF` directory.

The SAR file you'll create, `memory.sar`, needs to include the following files:

- `/com/xdocletbook/jmx/MemoryMonitor.class`—The MBean implementation
- `/com/xdocletbook/jmx/MemoryMonitorMBean.class`—The MBean interface
- `/META-INF/jboss-service.xml`—The service deployment descriptor file

You wrote `MemoryMonitor.java` in the last section, and the `<mbeaninterface>` subtask generates `MemoryMonitorMBean.java` for you. Once compiled, these will give

you the two class files that your SAR file needs. But where do you get the service deployment descriptor?

Fortunately, you can add the `<jbossxmlservicetemplate>` subtask to your build to have the deployment descriptor generated:

```
<jmxdoclet destdir="${gen.src.dir}" mergeDir="${merge.dir}">
...
  <jbossxmlservicetemplate destdir="${jboss.service.dir}"
      servicefile="jboss"
  />
</jmxdoclet>
```

You set the `destdir` attribute so that the service descriptor will be generated in a convenient path for later JARing up (or SARing up, to be exact). The value of the `servicefile` attribute is prepended to `-service.xml` to determine the name of the service descriptor file. In this case, the file will be named `jboss-service.xml`. Multiple service descriptor files can be generated by duplicating the `<jbossxmlservicetemplate>` subtask within the build and specifying a different value for the `servicefile` attribute.

The `<jbossxmlservicetemplate>` subtask alone won't generate the service descriptor file. It works in tandem with the `@jboss.service` class-level tag. So, you add this tag to your MBean:

```
/**
 * @jmx.mbean
 *     name="Memory:name=memoryMonitor"
 *     description="Memory Monitor MBean"
...
 * @jboss.service servicefile="jboss"
...
 */
public class MemoryMonitor implements MemoryMonitorMBean {
...
}
```

Pay special attention to the `servicefile` attribute of `@jboss.service`. It's no coincidence that in this example, its value matches that of `<jbossxmlservicetemplate>`'s `servicefile` attribute in the build file. Because it's possible to have multiple service descriptor files generated by XDoclet, there must be a mechanism to indicate which service descriptor file a particular MBean should be part of. This is accomplished by matching `@jboss.service`'s `servicefile` attribute to the `servicefile` attribute of a particular `<jbossxmlservicetemplate>` subtask.

Furthermore, it's also possible for a service to contain multiple MBeans. You can do this by setting the `servicefile` attribute of `@jboss.service` in several MBeans to all match the same `<jbossxmlservicetemplate>` in the build.

When the `<jbossxmlservicetemplate>` subtask processes `MemoryMonitor.java`, the following `jboss-service.xml` file is generated:

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

<service>

  <mbean code="com.xdocletbook.jmx.MemoryMonitor"
         name="Memory:name=memoryMonitor"
         >
    <!--Memory Monitor MBean-->
  </mbean>
</service>
```

Clearly, the `name` attribute of `<mbean>` in `jboss-service.xml` is received from the `name` attribute of the `@jmx.mbean` tag in `MemoryMonitor.java`.

Deploying an MBean service to JBossMX

Now that you have your service descriptor file, all that's left to do is to JAR (or SAR) it up along with the MBean's class files. To do this, add the following target to the `build-package.xml` file:

```
<target name="build-sar">
  <mkdir dir="${jboss.sar.dir}"/>

  <jar jarfile="${jboss.sar.dir}/memory.sar">
    <fileset dir="${build.dir}" includes="**/jmx/Memory*.class"/>
    <metainf dir="${jboss.service.dir}"/>
  </jar>
</target>
```

When the build is run, a file named `memory.sar` is created in the directory specified by the `${jboss.sar.dir}` variable. To deploy the memory service, copy this SAR file into the deploy directory for the JBoss server. If you're successful, JBoss should report something like this to its console:

```
[MainDeployer] Starting deployment of package:
    file:/D:/jboss-3.0.4/server/default/deploy/memory.jar
[MainDeployer] Deployed package:
    file:/D:/jboss-3.0.4/server/default/deploy/memory.jar
```

Now, point your web browser to the JBoss management console to see the memory monitor service deployed. It should be visible near the top under the Memory header, as shown in figure 9.7.

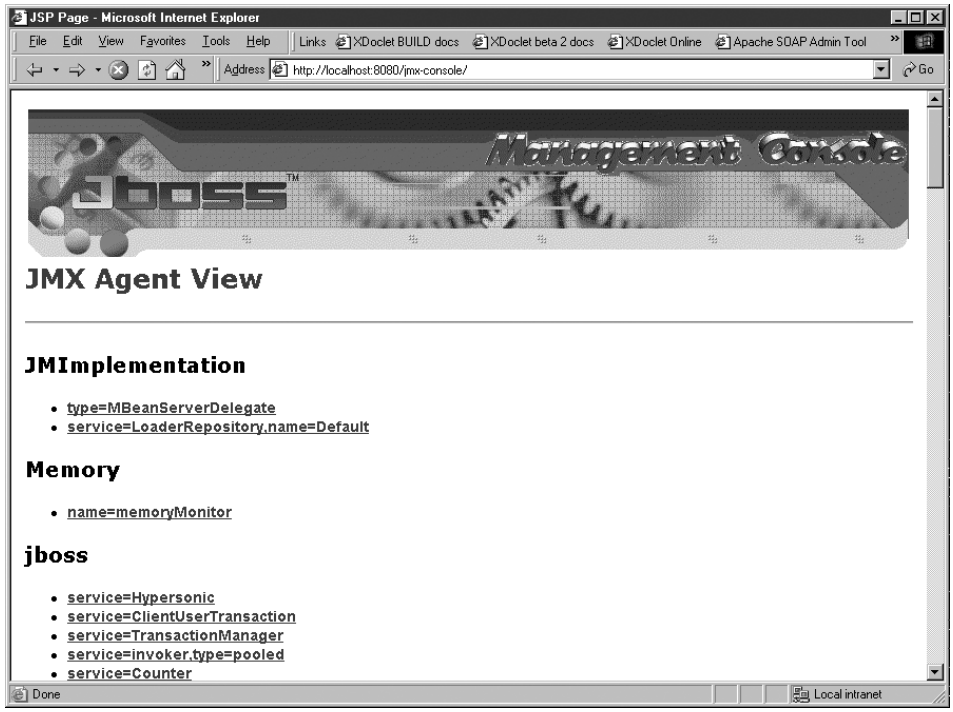

Figure 9.7 The `memoryMonitor` service deployed

To test the memory monitor MBean, click on the name=memoryMonitor link. You should be greeted with a vaguely familiar page showing the current memory levels (figure 9.8).

Now that you've seen how to deploy a standard MBean as a service in JBossMX, let's look at how you can use XDoclet to write model MBean definitions in XML for deployment in JBossMX.

9.5.2 Generating XML for JBossMX model MBeans

What if the management interface for an MBean is likely to change quite often? Using standard MBeans, you'd have to change the MBean interface and then recompile. Model MBeans make it possible to define the MBean's management interface in an external (non-code) entity so that the interface can be changed on the fly. Furthermore, model MBeans can define metadata that describes the MBean and its operations and attributes for improved usability.

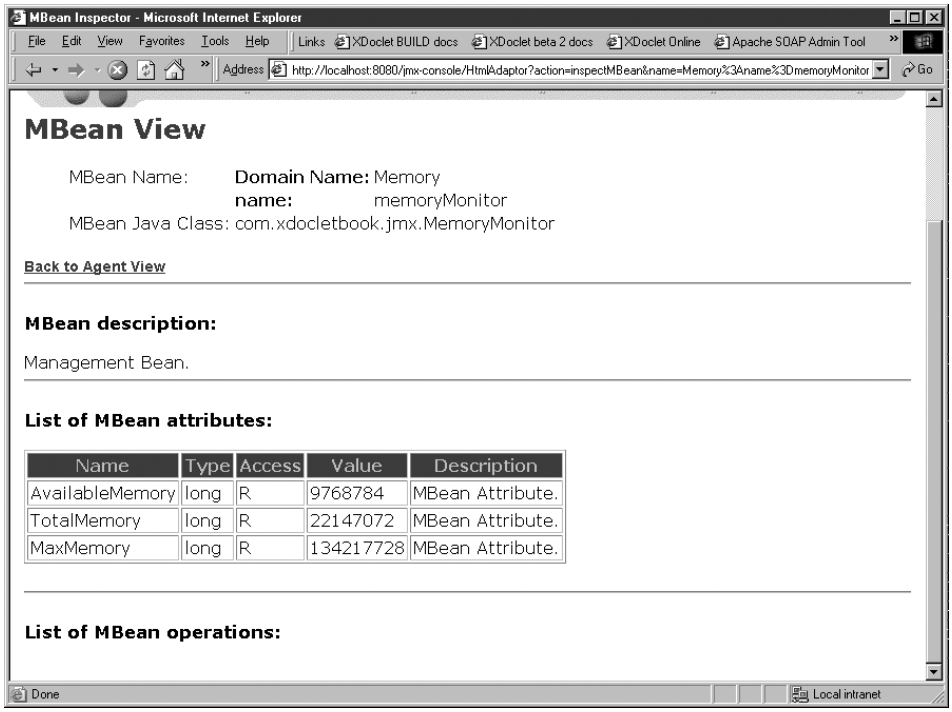

Figure 9.8 Accessing the `memoryMonitor` MBean through the JBoss management console

XMBeans, first made available in JBoss version 3.2.0, are a special type of model MBean whose interface is defined in an XML file. Just as XDoclet generates Java interfaces for standard MBeans, it can also generate XML descriptors that define the management interface of XMBeans.

Let's change the memory monitor MBean to be an XMBean instead of a standard MBean. You start by adding `the <jbossxmbean>` subtask to `<jmxdoclet>` in the build file to have XDoclet generate XMBean interface descriptor files:

```
<jmxdoclet destdir="${gen.src.dir}" mergeDir="${merge.dir}">
...
  <jbossxmbean destdir="${jboss.xmbean.dir}" />
</jmxdoclet>
```

The `<jbossxmbean>` subtask generates a management interface descriptor file for any class that is tagged at the class-level with the `@jboss.xmbean` tag. So, you tag `MemoryMonitor.java` at the class level with `@jboss.xmbean`:

```
/**
 * @jmx.mbean
 *     name="Memory:name=memoryMonitor"
 *     description="Memory Monitor MBean"
...
 *
 * @jboss.xmbean
 */
public class MemoryMonitor implements MemoryMonitorMBean {
...
}
```

When the build is run and `MemoryMonitor.java` is processed by `<jbossxmbean>`, the following XMBean descriptor, `MemoryMonitor.xml`, is created:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mbean PUBLIC "-//JBoss//DTD JBOSS XMBEAN 1.0//EN"
    "http://www.jboss.org/j2ee/dtd/jboss_xmbean_1_0.dtd">

<mbean>
  <description>Memory Monitor MBean</description>
  <descriptors>
    <persistence/>
  </descriptors>
  <class>com.xdocletbook.jmx.MemoryMonitor</class>

  <!--attributes-->
  <attribute access="read-only" getMethod="getAvailableMemory">

    <description>Amount of free memory available</description>
    <name>AvailableMemory</name>
    <type>long</type>
    <descriptors>
      <persistence/>

    </descriptors>
  </attribute>
  <attribute access="read-only" getMethod="getTotalMemory">

    <description>(no description)</description>
    <name>TotalMemory</name>
    <type>long</type>
    <descriptors>
      <persistence/>

    </descriptors>
  </attribute>
  <attribute access="read-only" getMethod="getMaxMemory">

    <description>(no description)</description>
    <name>MaxMemory</name>
    <type>long</type>
```

```
    <descriptors>
      <persistence/>

    </descriptors>
  </attribute>

<!--artificial attributes-->

<!--operations -->

<!--artificial operations-->

<!--notifications -->

</mbean>
```

Notice that `<jbossxmbean>` pulled together information from other `@jmx.*` tags to help generate the XMBean descriptor file. In this example, `<jbossxmbean>` takes advantage of the `description` and `access` attributes of `@jmx.managed-attribute` when defining the `<attribute>` declarations in `MemoryMonitor.xml`.

Another side effect of tagging `MemoryMonitor.java` with `@jboss.mxbean` is that a reference to `MemoryMonitor.xml` is included in the `jboss-service.xml` service descriptor file:

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

<service>

  <mbean code="com.xdocletbook.jmx.MemoryMonitor"
         name="Memory:name=memoryMonitor"
         xmbean-dd="com/xdocletbook/jmx/MemoryMonitor.xml"
         >
    <!--Memory Monitor MBean-->
  </mbean>
</service>
```

To include `MemoryMonitor.xml` in the SAR file, add an additional `<fileset>` that includes the XMBean directory to the `<jar>` task you used before:

```
<target name="build-sar">
  <mkdir dir="${jboss.sar.dir}"/>

  <jar jarfile="${jboss.sar.dir}/memory.sar">
    <fileset dir="${build.dir}" includes="**/jmx/Memory*.class"/>
    <fileset dir="${jboss.xmbean.dir}"
        includes="**/jmx/Memory*.xml"/>
    <metainf dir="${jboss.service.dir}"/>
  </jar>
</target>
```

After you redeploy `memory.sar` to JBoss, look at the JMX Console in JBoss again (figure 9.9). You should see the same MBean as before, except that this time the MBean and its attributes have descriptions other than the default descriptions you saw earlier. Also, you'll notice that the MBean implementation class is now `org.jboss.mx.modelmbean.XMBean`, indicating that this is an XMBean and not a standard MBean.

The nice thing about deploying MBeans as XMBeans is that you can alter the management interface without recompiling. You simply deploy a new XML descriptor file, and when JBossMX picks it up, your MBean service will have a new interface.

Now that you've seen how to use XDoclet to write MBeans for JBossMX, let's look at how XDoclet can help you develop MBeans for MX4J, another implementation of the JMX specification.

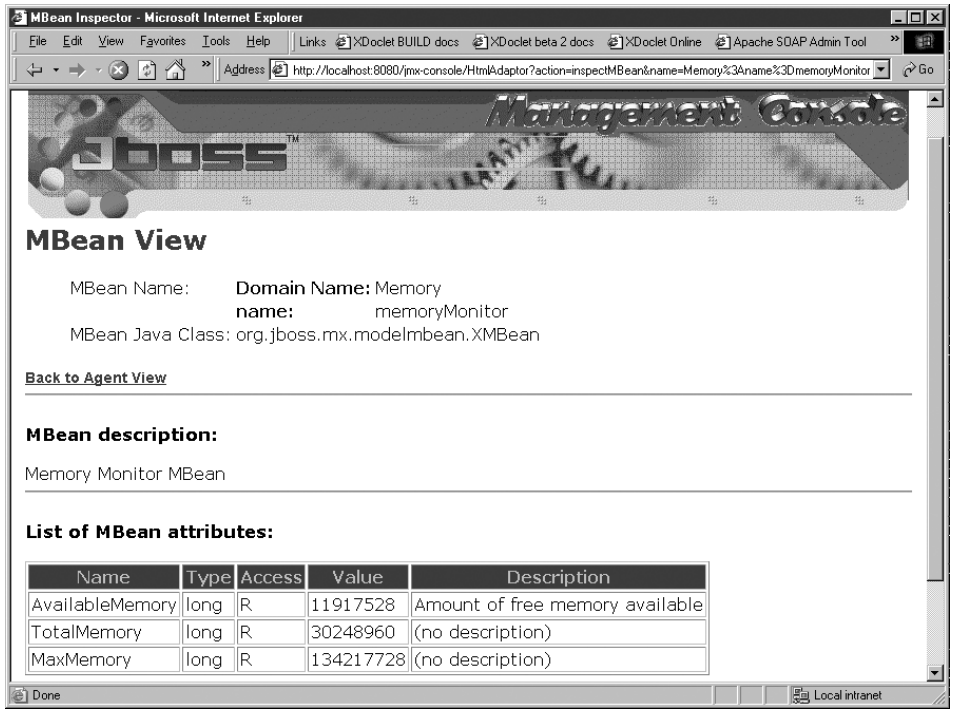

Figure 9.9 The `memoryMonitor` MBean deployed as an XMBean

9.6 Generating MBean description classes for MX4J

Suppose you're going to deploy your MBeans into MX4J. MX4J is another open-source (Apache-style license) implementation of JMX. Its most notable application is that it serves as the basis for Jakarta Tomcat's management interface. You can download and read more about MX4J at http://mx4j.sourceforge.net.

Your beans will work fine as-is in MX4J. However, with MX4J you also have the option of describing MBeans and their managed operations with MBean description classes.

Using reflection, the JMX agent can retrieve information about managed operations, attributes, constructors, and notifications. However, it can't retrieve information that's important for the user of the managed application, such as operation, attribute, and parameter descriptions; parameter names; and so forth. MBean description classes enable you to customize the descriptions surrounding an MBean's interface for increased usability.

An MBean description class is any class that either implements `mx4j.MBeanDescription` or extends `mx4j.MBeanDescriptionAdaptor` and is named with the same fully qualified name of the MBean class ending with `MBeanDescription`.

These are the steps you must take to have XDoclet generate MBean description classes for your MBeans:

1. Add the `<mx4jdescription>` subtask to your build file.
2. Tag your MBean class file with the appropriate `@jmx.*` tags and `description` attributes.
3. Run the build.
4. Deploy the MBean in an MX4J agent.

9.6.1 Preparing the build for MX4J

By adding the `<mx4jdescription>` subtask under `<jmxdoclet>` in your build file, you can have XDoclet write MBean description classes for you:

```
<jmxdoclet destdir="${gen.src.dir}" mergeDir="${merge.dir}">
  <fileset dir="${src.dir}">
    <include name="**/jmx/*.java"/>
  </fileset>

  <mbeaninterface/>
  <mlet/>
  <mx4jdescription/>
</jmxdoclet>
```

9.6.2 Tagging MBeans for MX4J

The `<mx4jdescription>` subtask uses the `description` attribute of the following tags to create an MBean descriptor class for each class it processes that's tagged with `@jmx.mbean`:

- `@jmx.mbean`
- `@jmx.managed-constructor`
- `@jmx.managed-constructor-parameter`
- `@jmx.managed-attribute`
- `@jmx.managed-operation`
- `@jmx.managed-operation-parameter`

As you'll see, `<mx4jdescription>` also uses the `name` and `position` attributes of `@jmx.managed-constructor-parameter` and `@jmx.managed-operation-parameter` when describing parameters.

In the event that the MBean (or any of its managed operations and attributes) is not tagged with the description attributes of these tags, then the generated MBean description class will still be generated, but it will give default values for descriptions.

Let's revisit the `BlogManager` MBean. Listing 9.6 shows the new `BlogManager` class, updated with meta-information that `<mx4jdescription>` can use when generating an MBean descriptor class for `BlogManager`.

Listing 9.6 BlogManager.java, updated to be more descriptive for MX4J

```
package com.xdocletbook.jmx;

import java.util.List;

import com.xdocletbook.blog.interfaces.BlogFacade;
import com.xdocletbook.blog.interfaces.BlogFacadeHome;
import com.xdocletbook.blog.util.BlogFacadeUtil;

/**
 * @jmx.mbean
 *    name="BlogManager"
 *    description="Some basic administrative functions for blogs."
 */
public class BlogManager implements BlogManagerMBean {
  BlogFacadeHome home;

  public BlogManager() throws Exception {
    home = BlogFacadeUtil.getHome();
  }
```

Describe MBean

```
  /**
   * @jmx.managed-operation
   *     description="Retrieve a blog's name given its ID."
   *
   * @jmx.managed-operation-parameter
   *     description="The ID of the blog to be looked up."
   *     name="blogId"
   *     position="1"
   */
  public String getBlogName(String blogId) throws Exception {
    BlogFacade facade = home.create();
    return facade.getBlogSimple(blogId).getName();
  }

  /**
   * @jmx.managed-operation
   *     description="Delete a blog."
   *
   * @jmx.managed-operation-parameter
   *     description="The ID of the blog to be deleted."
   *     name="blogId"
   *     position="1"
   */
  public void deleteBlog(String blogId) throws Exception {
    home.create().deleteBlog(blogId);
  }

  /**
   * @jmx.managed-operation
   *     description="Retrieve a list of all blogs."
   */
  public List getAllBlogs() throws Exception{
    BlogFacade facade = home.create();
    return facade.getAllBlogs();
  }
}
```

Describe managed operations

Describe parameters to the managed operations

Describe managed operations

Describe parameters to the managed operations

Describe managed operations

9.6.3 Running the build

When `<mx4jdescription>` processes this new `BlogManager.java`, it generates `BlogManagerMBeanDescription.java`:

```
/*
 * Generated file - Do not edit!
 */
package com.xdocletbook.jmx;
import mx4j.MBeanDescriptionAdapter;
import java.lang.reflect.Method;
import java.lang.reflect.Constructor;
```

```
/**
 * MBean description.
 */
public class BlogManagerMBeanDescription
    extends MbeanDescriptionAdapter {

  public String getMBeanDescription() {
    return "Some basic administrative functions for blogs.";
  }

  public String getConstructorDescription(Constructor ctor) {
    String name = ctor.getName();

    return super.getConstructorDescription(ctor);
  }

  public String getConstructorParameterName(Constructor ctor,
    int index) {

    return super.getConstructorParameterName(ctor, index);
  }

  public String getConstructorParameterDescription(Constructor ctor,
      int index) {

    return super.getConstructorParameterDescription(ctor, index);
  }

  public String getAttributeDescription(String attribute) {

      return super.getAttributeDescription(attribute);
  }

  public String getOperationDescription(Method operation) {
    String name = operation.getName();

    if (name.equals("getBlogName")) {
      return "Retrieve a blog's name given its ID.";
    }
    if (name.equals("deleteBlog")) {
      return "Delete a blog.";
    }
    if (name.equals("getAllBlogs")) {
      return "Retrieve a list of all blogs.";
    }

    return super.getOperationDescription(operation);
  }

  public String getOperationParameterName(Method method,
      int index) {
    String name = method.getName();

    if (name.equals("getBlogName")) {
      switch (index) {
```

```
        case 1:
          return "blogId";
      }
    }
    if (name.equals("deleteBlog")) {
      switch (index) {
         case 1:
           return "blogId";
      }
    }

    return super.getOperationParameterName(method, index);
  }

  public String getOperationParameterDescription(Method method,
      int index) {
    String name = method.getName();

    if (name.equals("getBlogName")) {
      switch (index) {
        case 1:
          return "The ID of the blog to be looked up.";
      }
    }
    if (name.equals("deleteBlog")) {
      switch (index) {
        case 1:
          return "The ID of the blog to be deleted.";
      }
    }

    return super.getOperationParameterDescription(method, index);
  }
}
```

9.6.4 Deploying the MBean into MX4J

To take advantage of this MBean descriptor class, you must deploy your MBean (and this class) within an MX4J agent. To do so, change `JMXStartupListener.java` to include MX4J's `HttpAdaptor` (MX4J's answer to Sun's `HtmlAdaptorServer`) among the services it starts (listing 9.7).

Listing 9.7 JMXStartupListener.java, modified to also start MX4J's HttpAdaptor

```
package com.xdocletbook.blog.listener;
import javax.management.MBeanServer;
import javax.management.MBeanServerFactory;
import javax.management.ObjectName;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import mx4j.adaptor.http.HttpAdaptor;
```

```
import com.sun.jdmk.comm.HtmlAdaptorServer;
import com.xdocletbook.jmx.BlogManager;

/**
 * @web.listener
 */
public class JMXStartupListener implements ServletContextListener {
  private MBeanServer server;
  private HttpAdaptor httpAdaptor;
  private HtmlAdaptorServer htmlAdaptor;

  public void contextDestroyed(ServletContextEvent arg0) {
    System.out.println("STOPPING MBEAN SERVER");
    httpAdaptor.stop();
    htmlAdaptor.stop();
  }

  public void contextInitialized(ServletContextEvent arg0) {
    System.out.println("STARTING MBEAN SERVER");

    server = MBeanServerFactory.createMBeanServer("BlogAgent");

    try {
      startBlogManager();
      startHtmlAdaptor();
      startHttpAdaptor();   <-- Register and start MX4J's HttpAdaptor
    } catch (Exception e) {
      System.err.println("Exception:  "+e);
    }
  }

  private void startBlogManager() throws Exception {
    server.registerMBean(new BlogManager(),
        new ObjectName("BlogAgent:name=blogManager"));
  }

  private void startHtmlAdaptor() throws Exception {
    htmlAdaptor = new HtmlAdaptorServer();
    htmlAdaptor.setPort(9092);

    server.registerMBean(htmlAdaptor,
        new ObjectName("Server:name=HtmlAdaptor"));
    htmlAdaptor.start();
  }

  private void startHttpAdaptor() throws Exception {
    httpAdaptor = new HttpAdaptor();
    httpAdaptor.setPort(9093);

    server.registerMBean(httpAdaptor,
        new ObjectName("Server:name=HttpAdaptor"));
    httpAdaptor.start();
  }
}
```

Register and start MX4J's HttpAdaptor

The `HttpAdaptor` listens on port 9093 for MX4J commands that come in the form of HTTP requests. For example, to retrieve a list of the MBeans available in the server, point your web browser to http://localhost:9093/server. The result should look something like Figure 9.10.

Notice that the results come back in XML format. Although this may be more difficult for human eyes to read, it makes it much easier to write applications that interface with MBeans through this interface.

Unfortunately, MX4J's `server` command doesn't tell you much about the `BlogManager` MBean. For example, you can't see any of the managed operations that are exposed via JMX. The `mbean` command allows you to drill down and see more details about a specific MBean. Point your browser to http://localhost:9093/mbean?objectname=BlogAgent:name=blogManager to see the MBean detail for the `BlogManager` MBean (figure 9.11).

From the MBean detail screen, you can see the purpose of the MBean description class. Notice that all the descriptions with which you tagged your MBean class are visible in the XML returned from the `mbean` command.

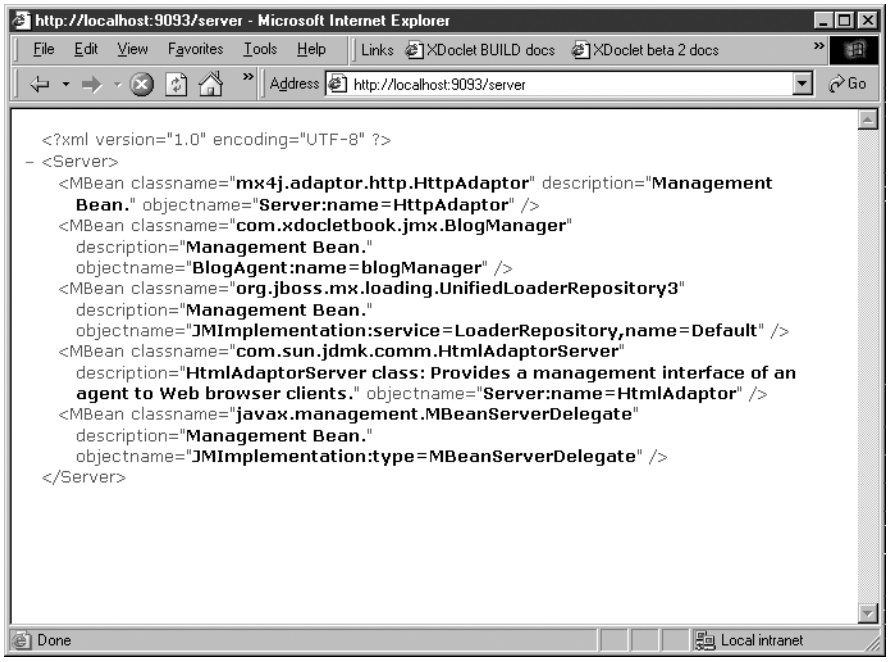

Figure 9.10 All the deployed MBeans as reported by MX4J's `server` command

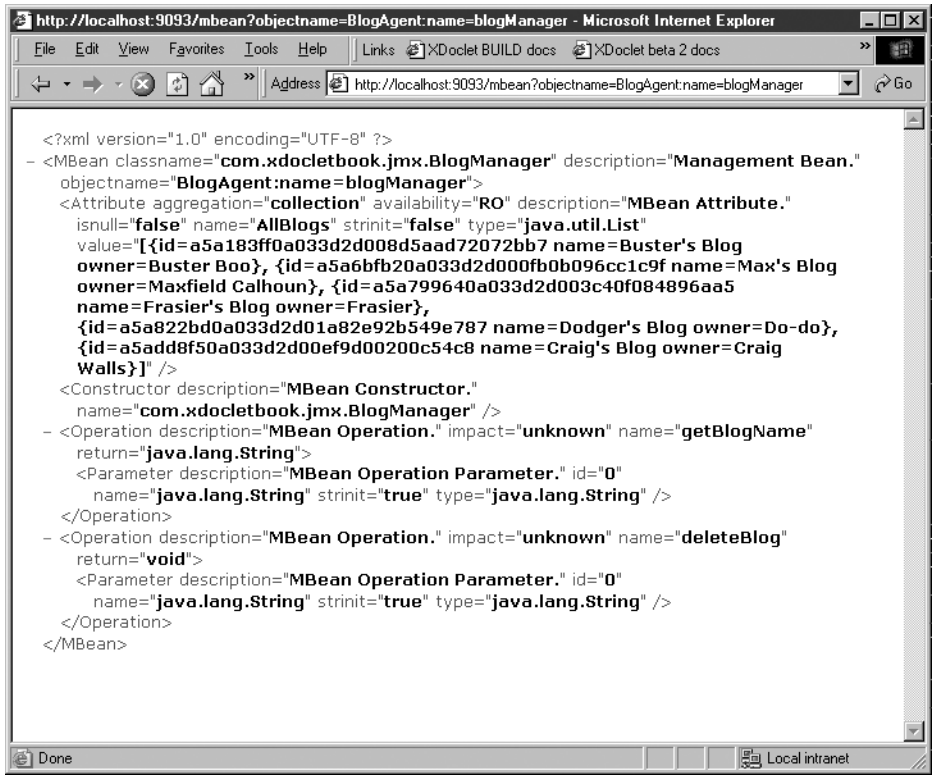

Figure 9.11 `BlogManager` **MBean details as seen through MX4J**

9.7 Summary

You've seen how JMX can make it easy to instrument an application for management. You've also seen how MBean development can be further simplified using XDoclet to generate MBean interfaces, mlet deployment files, and other artifacts associated with extensions to the JMX specification.

In the next chapter, we'll wrap up our discussion of the code generation modules that come with XDoclet by looking at how XDoclet can generate mock object classes that can be used to perform unit testing from the inside out.

JAVA

XDoclet IN ACTION

Craig Walls and Norman Richards

Are you tired of repeatedly writing essentially the same Java code? XDoclet will take the burden of common development tasks off your shoulders by writing code for you. XDoclet is a metadata driven, code generation engine for Java. It generates deployment descriptors, interfaces, framework classes and other utility classes your project requires from simple JavaDoc-style comments.

XDoclet in Action is a complete resource for XDoclet code generation. With many short code examples and a full-scale J2EE example, the book shows you how to use XDoclet with EJBs, Servlets, JMX, and other technologies. You'll also learn how to customize XDoclet beyond its out-of-the-box capabilities to generate code specific to your application. This book shows you how to write less code, how to keep your application components in sync, and how to keep your deployment, interface, utility and other information all in one place.

What's Inside

- Introduction to XDoclet
- Best practices and techniques
- How to customize XDoclet
- How to use XDoclet with
  - EJB
  - Servlets
  - Struts and WebWork
  - JDO
  - Hibernate
  - JMX
  - SOAP
  - MockObjects
  - JBoss and WebLogic
  - Eclipse and IDEA

Craig Walls, an XDoclet project committer, has been a software developer since 1994 and a Java fanatic since 1996. He lives in Dallas, Texas. **Norman Richards** has developed software for a decade and has been working with code generation techniques for much of that time. He is an avid XDoclet user and evangelist. Norman lives in Austin, Texas.

"This is the first serious XDoclet book, and the authors got it right!"

—Michael Yuan
JavaWorld Author

"XDoclet is the missing link for complex code generation ... This book will quickly teach you how to build and deploy J2EE projects with just a click."

—Daniel Brookshier, author of *JXTA: Java P2P Programming*

"I learned a lot reading it ... I was immediately filled with new ideas on how to use XDoclet. It's perfect."

—Rickard Öberg
inventor of EJBdoclet

"The patterns described make a lot of sense to me ... I've done things the hard way in the past."

—Nathan Egge
Argon Engineering

"... a fantastic job ... written clearly and effectively, with humor too.

—Erik Hatcher, co-author of *Java Development With Ant*

www.manning.com/walls

AUTHOR ONLINE Authors respond to reader questions

Ebook edition available

MANNING