WrapUpBeanWithBaseModelMBean

At the core of Modeler is the ModelMBean interface — for those unfamiliar with this, it allows a developer to wrap up any old Java class into an MBean which can be then managed via normal JMX means. In other words, you can take any class you want to export to JMX and without changing your class, without your class even "knowing" it's going to be JMX'd, you can do so — by the usage of a ModelMBean!

Unfortunately, for those of you who tried to use a ModelMBean — in fact same for DynamicMBean — the interfaces for creating these and defining the metadata for each bean can be quite clunky and awkward at times to use. I thought in the past, when working outside the Spring framework, that it would be so good to have a way to just say "here's my class, a standard Java class, not aligned to any JMX naming conventions or anything — please register this somehow with JMX so I can inspect data through JMX Console". This is possible through Commons Modeler.

In this "how to", I'm going to start with a very simple bean and show how to wrap it up using Commons Modeler.

For the purpose of this I'm going to use a (very) simple Java bean:

```
/**
 * Simple bean class with a few basic properties.
 *
 * @author Liviu Tudor http://about.me/liviutudor
 */
public class JacksMagicBean {
    private int     magicNumber;

    private String  magicWord;

    private boolean magicBoolean;

    public JacksMagicBean() {
        this(0, null, false);
    }

    public JacksMagicBean(int magicNumber, String magicWord, boolean magicBoolean) {
        this.magicNumber = magicNumber;
        this.magicWord = magicWord;
        this.magicBoolean = magicBoolean;
    }

    public int getMagicNumber() {
        return magicNumber;
    }

    public void setMagicNumber(int magicNumber) {
        this.magicNumber = magicNumber;
    }

    public String getMagicWord() {
        return magicWord;
    }

    public void setMagicWord(String magicWord) {
        this.magicWord = magicWord;
    }

    public boolean isMagicBoolean() {
        return magicBoolean;
    }

    public void setMagicBoolean(boolean magicBoolean) {
        this.magicBoolean = magicBoolean;
    }

    public void switchMagicBoolean() {
        magicBoolean = !magicBoolean;
    }

    public void addNumber( int number ) {
        this.magicNumber += number;
    }

    @Override
    public String toString() {
```

```
        return "JacksMagicBean [magicNumber=" + magicNumber + ", magicWord=" + magicWord + ", magicBoolean="
                + magicBoolean + "]";
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + magicNumber;
        result = prime * result + ((magicWord == null) ? 0 : magicWord.hashCode());
        result = prime * result + (magicBoolean ? 1 : 0);
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        JacksMagicBean other = (JacksMagicBean) obj;
        if (magicNumber != other.magicNumber) return false;
        if (magicBoolean != other.magicBoolean) return false;
        if (magicWord == null) {
            if (other.magicWord != null) return false;
        } else if (!magicWord.equals(other.magicWord)) return false;
        return true;
    }
}
```

As you can see, nothing tricky here – just a bunch of properties and a couple of methods. You could use a JMXBean I suppose to define an interface and have this JMX'd — but this introduces another interface and makes (to me at least) the code less readable.

You could of course used the ModelMBean class as it is — and after you've fallen asleep on your keyboard writing all the code to deal with the attribute information and the method metadata etc etc etc you'll get there `http://liviutudor.com/wp-includes/images/smilies/icon_biggrin.gif|😃|class="wp-smiley"`

Or you can use a piece of code like this (look at registerBean function):

```
MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
BaseModelMBean bmBean = new BaseModelMBean(JacksMagicalBean.class.getCanonicalName());
ObjectName name = new ObjectName(someStringName);
mbs.registerMBean(bmBean, name);
```

If you run the attached code and inspect it with *JConsole* (I've put a few "breakpoints" in the code, so the program stops waiting for user input in the console, giving you time to look at *JConsole*) you will see that auto-magically all the bean properties are exposed via JMX and the methods too! And if you take the instantiation of the platform MBeanServer out, you are left with only a couple of lines!

You are however still left with the task of creating the ObjectName instance — small task, I know, but the constructor throws a MalformedObjectNameException, which presents the inconvenience (when coding around it) of having to wrap it up in try/catch, even though you know for sure the naming used is perfectly valid! So would be good if we can shorten this sequence just a bit more and pass the responsibility of the whole ObjectName creation to a function that handles that too.

This is possible by using the Registry class. This is a wrapper ultimately for the MBeanServer but offers a few extra features — as to be expected. One of them, is a method which takes an object (instance of your bean you want to JMX) and a name to use and does all the work under the covers:

- create a ModelMBean to wrap up your bean
- the newly-created ModelMBean will introspect your bean and create all the metadata needed to made all the bean's properties and methods
- it then creates an ObjectName with the given name
- and finally registers the ModelMBean with the registry — which in turns means registering this with the platform MBeanServer

So if you look at the registerObject() method, you will see that it's very short — and calls simply just one function on the Registry class:

```
    private static void registerObject(final Registry registry, final Object obj, String oName) {
        try {
            registry.registerComponent(obj, oName, null);
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(4);
        }
    }
```

In fact apart from the call registry.registerComponent(), all the extra code in this function is just to prevent exception from bubbling up the call stack! One line of code for each of your beans to export it to JMX — not bad, huh? And here's what *JConsole* reports at the first "breakpoint" — which is after we register 2 sets of beans, one using the MBeanServer method, and one using Registry — the outcome being exactly the same.

And here's another feature that I did like in Registry: you can invoke a method on a set of beans in one go — by calling invoke() and passing it a list of object names! Might not sound like much, but imagine a scenario where you have a plugin-based system: 3rd parties can extend certain classes of yours and at some point you want to ensure all of these plugins get to a certain state — maybe you just want to initialize them, or reset them etc. You can of course build a whole listener/notification mechanism, or you can avoid that, and simply send the notifications in one go via JMX — by calling invoke()!

Finally, below is the source of samples program:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.lang.management.ManagementFactory;
import java.util.ArrayList;
import java.util.List;

import javax.management.MBeanServer;
import javax.management.ObjectName;

import org.apache.commons.modeler.BaseModelMBean;
import org.apache.commons.modeler.Registry;

/**
 * Sample entry point.
 *
 * @author Liviu Tudor http://about.me/liviutudor
 */
public class SamplesDriver {
    private static final int N_BEANS = 10;

    /**
     * Program entry point. Simply starts the Sun JDMK agent.
     *
     * @param args
     *            command-line parameters -- ignored
     */
    public static void main(String[] args) {
        // do the actual work
        MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
        // instantiate N_BEANS beans
        for (int i = 1; i < N_BEANS; i++) {
            registerBean(mbs, baseModelName(i), JacksMagicBean.class);
        }

        // retrieve the registry
        Registry reg = getRegistry();

        // an easier way of registering
        for (int i = 1; i < N_BEANS; i++) {
            registerObject(reg, new JacksMagicBean(), simpleObjectName(i));
        }

        // invoked method on all of the beans we created above
        List
```

Please note this article is a copy from my initial blog post here, slightly modified to include the complete sources here. Also I couldn't figure out how to include the screenshot (still a rookie with wiki sorry 🙂) so it does look slightly different to the original, but the contents is nevertheless the same.

One last note – the waitForUserInput() function has an if/else branch as it turns out System.console() can return null when run in most IDE's nowadays, as such, for those of you who want to run this in an IDE, you will not get a prompt – hence the if/else.