covers Spring 3.0

Spring IN ACTION

THIRD EDITION

Craig Walls

MANNING

Table of Contents

Managing Spring beans with JMX

This chapter covers

- Exposing Spring beans as managed beans
- Remotely managing Spring beans
- Handling JMX notifications

Spring's support for DI is a great way to configure bean properties in an application. But once the application has been deployed and is running, DI alone can't do much to help you change that configuration. Suppose that you want to dig into a running application and change its configuration on the fly. That's where *Java Management Extensions (JMX)* comes in.

JMX is a technology that enables you to instrument applications for management, monitoring, and configuration. Originally available as a separate extension to Java, JMX is now a standard part of the Java 5 distribution.

The key component of an application that's instrumented for management with JMX is the *MBean (managed bean)*. An MBean is a JavaBean that exposes certain methods which define the management interface. The JMX specification defines four types of MBeans:

- *Standard MBeans*—Standard MBeans are MBeans whose management interface is determined by reflection on a fixed Java interface that's implemented by the bean class.
- *Dynamic MBeans*—Dynamic MBeans are MBeans whose management interface is determined at runtime by invoking methods of the `DynamicMBean` interface. Because the management interface isn't defined by a static interface, it can vary at runtime.
- *Open MBeans*—Open MBeans are a special kind of dynamic MBean whose attributes and operations are limited to primitive types, class wrappers for primitive types, and any type that can be decomposed into primitives or primitive wrappers.
- *Model MBeans*—A model MBean is a special kind of dynamic MBean that bridges a management interface to the managed resource. Model MBeans aren't written as much as they are declared. They're typically produced by a factory that uses some metainformation to assemble the management interface.

Spring's JMX module enables you to export Spring beans as Model MBeans so that you can see inside your application and tweak the configuration—even while the application is running. Let's see how to JMX-enable our Spring application so that we can manage the beans in the Spring application context.

13.1 Exporting Spring beans as MBeans

There are several ways that we could use JMX to manage the beans within the Spitter application. In the interest of keeping things simple, let's start with a modest change to the `HomeController` to add a new `spittlesPerPage` property:

```
public static final int DEFAULT_SPITTLES_PER_PAGE = 25;

private int spittlesPerPage = DEFAULT_SPITTLES_PER_PAGE;

public void setSpittlesPerPage(int spittlesPerPage) {
  this.spittlesPerPage = spittlesPerPage;
}

public int getSpittlesPerPage() {
  return spittlesPerPage;
}
```

Previously, when `HomeController` called `getRecentSpittles()` on the `SpitterService`, it passed in `DEFAULT_SPITTLES_PER_PAGE`, which would result in at most 25 spittles being displayed on the home page. Now, rather than commit to that decision at build time, we're going to use JMX to leave that decision open to change at runtime. The new `spittlesPerPage` property is the first step to enabling that.

But on its own, the `spittlesPerPage` property can't enable external configuration of the number of spittles displayed on the home page. It's just a property on a bean, like any other property. What we'll need to do next is to expose the `HomeController` bean as an MBean. Then, the `spittlesPerPage` property will be exposed as the MBean's *managed attribute* and we'll be able to change its value at runtime.

Spring's `MBeanExporter` is the key to JMX-ifying beans in Spring. `MBeanExporter` is a bean that exports one or more Spring-managed beans as Model MBeans in an *MBean server.* An MBean server (sometimes called an *MBean agent*) is a container where MBeans live and through which the MBeans are accessed.

As illustrated in figure 13.1, exporting Spring beans as JMX MBeans makes it possible for a JMX-based management tool such as JConsole or VisualVM to peer inside a running application to view the beans' properties and invoke their methods.

The following `<bean>` declares an `MBeanExporter` in Spring to export the `homeController` bean as a Model MBean:

```
<bean id="mbeanExporter"
  class="org.springframework.jmx.export.MBeanExporter">
  <property name="beans">
    <map>
      <entry key="spitter:name=HomeController"
             value-ref="homeController"/>
    </map>
  </property>
</bean>
```

In its most straightforward form, `MBeanExporter` can be configured through its `beans` property by injecting a `Map` of one or more beans that you'd like to expose as model MBeans in JMX. The `key` of each `<entry>` is the name to be given to the MBean (composed of a management domain name and a key-value pair—`spitter:name=HomeController` in the case of the `HomeController` MBean). The value of the `entry` is a

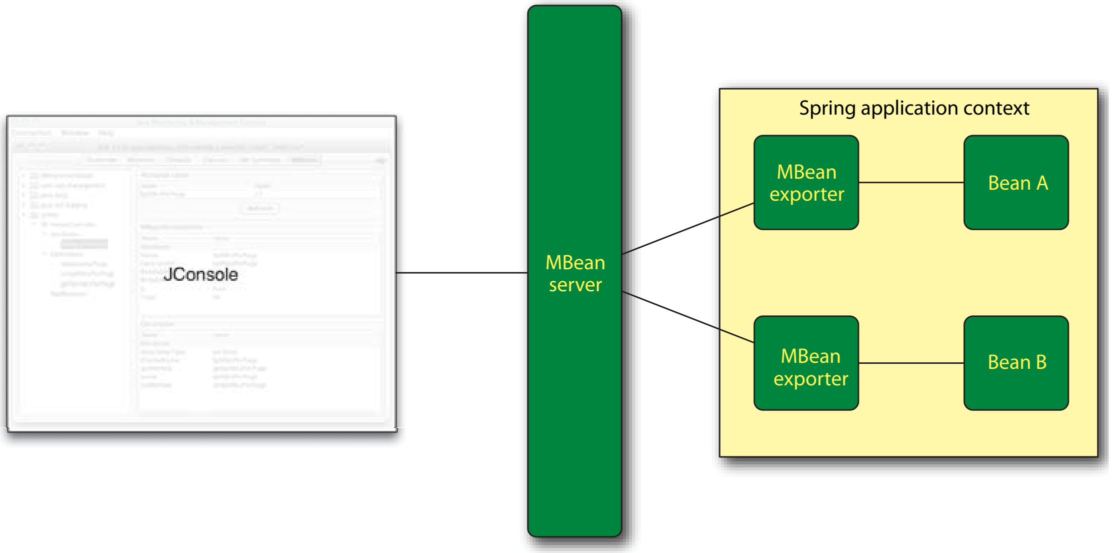


Figure 13.1 Spring's `MBeanExporter` exports the properties and methods of Spring beans as JMX attributes and operations in an MBean server. From there, a JMX management tool such as JConsole can look inside the running application.

From whence the MBean server?

As configured, `MBeanExporter` assumes that it's running within an application server (such as Tomcat) or some other context that provides an MBean server. But if your Spring application will be running standalone or in a container that doesn't provide an MBean server, you'll want to configure an MBean server in the Spring context. The `<context:mbean-server>` element can handle that for you:

```
<context:mbean-server />
```

`<context:mbean-server>` will create an MBean server as a bean within the Spring application context. By default, that bean's ID is `mbeanServer`. Knowing this, you can then wire it into `MBeanExporter`'s `server` property to specify which MBean server an MBean should be exposed through:

```
<bean id="mbeanExporter"
      class="org.springframework.jmx.export.MBeanExporter">
  <property name="beans">
    <map>
      <entry key="spitter:name=HomeController"
             value-ref="homeController"/>
    </map>
  </property>

  <property name="server" ref="mbeanServer" />
</bean>
```

reference to the Spring-managed bean that's to be exported. Here, we're exporting the `homeController` bean so that its properties can be managed at runtime through JMX.

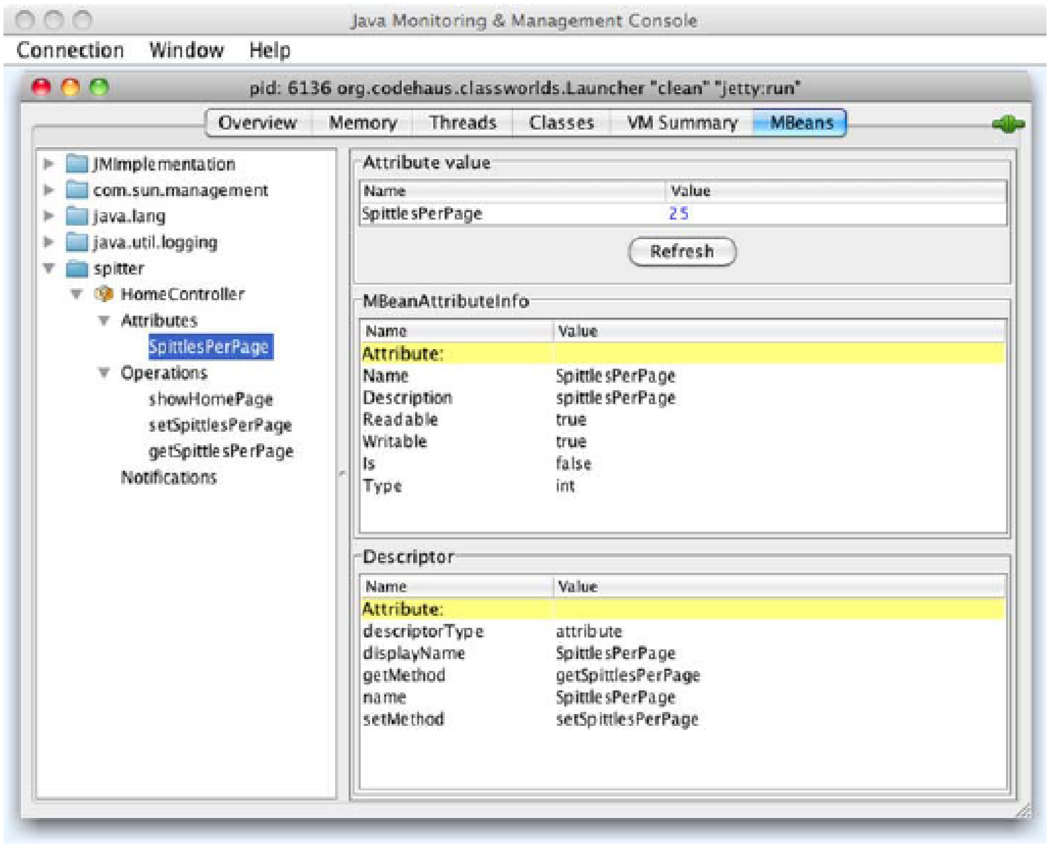
With the `MBeanExporter` in place, the `homeController` bean will be exported as a Model MBean to the MBean server for management under the name `HomeController`. Figure 13.2 shows how the `homeController` MBean appears when viewed through JConsole.

As you can see on the left side of figure 13.2, all public members of the `homeController` bean are exported as MBean operations and attributes. This is probably not what we want. All we really want to do is configure the `spittlesPerPage` property. We don't need to invoke the `showHomePage()` method or muck about with any other part of `HomeController`. Thus, we need a way to select which attributes and operations are available.

To gain finer control on an MBean's attributes and operations, Spring offers a few options, including

- Declaring bean methods to expose/ignore by name
- Fronting the bean with an interface to select the exposed methods
- Annotating the bean to designate managed attributes and operations

Let's try out each of these options to see which best suits our `HomeController` MBean. We'll start with selecting the bean methods to expose by name.



Name	Value
SpittlesPerPage	25

Name	Value
Attribute:	
Name	SpittlesPerPage
Description	spittlesPerPage
Readable	true
Writable	true
Is	false
Type	int

Name	Value
Attribute:	
descriptorType	attribute
displayName	SpittlesPerPage
getMethod	getSpittlesPerPage
name	SpittlesPerPage
setMethod	setSpittlesPerPage

Figure 13.2 **`HomeController` exported as an MBean and seen through the eyes of JConsole**

13.1.1 *Exposing methods by name*

An *MBean info assembler* is the key to constraining which operations and attributes are exported in an MBean. One such MBean info assembler is `MethodNameBasedMBeanInfoAssembler`. This assembler is given a list of names of methods to export as MBean operations. For the `HomeController` bean, what we want to do is export `spittlesPerPage` as a managed attribute. How can a method name–based assembler help us export a managed attribute?

Recall that per JavaBean rules (not necessarily Spring bean rules), what makes `spittlesPerPage` a property is that it has corresponding accessor methods names `setSpittlesPerPage()` and `getSpittlesPerPage()`. To limit our MBean's exposure, we'll need to tell `MethodNameBasedMBeanInfoAssembler` to only include those methods in the MBean's interface. The following declaration of a `MethodNameBasedMBeanInfoAssembler` bean singles out those methods:

```
<bean id="assembler"
        class="org.springframework.jmx.export.assembler.
        ➥MethodNameBasedMBeanInfoAssembler"
        p:managedMethods="getSpittlesPerPage,setSpittlesPerPage" />
```

The managedMethods property takes a list of method names. Those are the methods that will be exposed as the MBean's managed operations. Since those are property accessor methods, they'll also result in a spittlesPerPage managed attribute on the MBean.

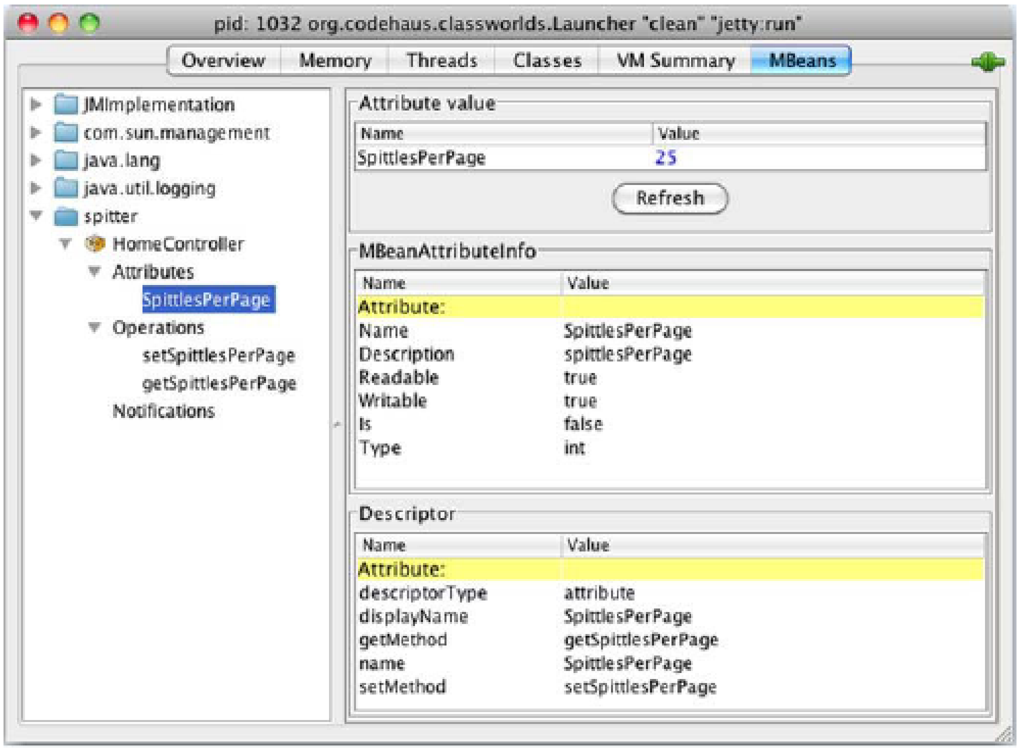
To put the assembler into action, we'll need to wire it into the MBeanExporter:

```
<bean id="mbeanExporter"
  class="org.springframework.jmx.export.MBeanExporter">
  <property name="beans">
    <map>
      <entry key="spitter:name=HomeController"
             value-ref="homeController"/>
    </map>
  </property>
  <property name="server" ref="mbeanServer" />

  <property name="assembler" ref="assembler"/>
</bean>
```

Now if we fire up the application, the HomeController's spittlesPerPage will be available as a managed attribute, but the showHomePage() method won't be exposed as a managed operation. Figure 13.3 shows what this looks like in JConsole.

Another method name–based assembler to consider is MethodExclusionMBean-InfoAssembler. This MBean info assembler is the inverse of MethodNameBasedMBean-



Name	Value
SpittlesPerPage	25

Name	Value
Attribute:	
Name	SpittlesPerPage
Description	spittlesPerPage
Readable	true
Writable	true
Is	false
Type	int

Name	Value
Attribute:	
descriptorType	attribute
displayName	SpittlesPerPage
getMethod	getSpittlesPerPage
name	SpittlesPerPage
setMethod	setSpittlesPerPage

Figure 13.3 After specifying which methods are exported in the HomeController MBean, the showHomePage() method is no longer a managed operation.

InfoAssembler. Rather than specifying which methods to expose as managed operations, MethodExclusionMBeanInfoAssembler is given a list of methods to *not* reveal as managed operations. For example, here's how to use MethodExclusion-MBeanInfoAssembler to keep showHomePage() out of consideration as a managed operation:

```
<bean id="assembler"
    class="org.springframework.jmx.export.assembler.
    ➥MethodExclusionMBeanInfoAssembler"
p:ignoredMethods="showHomePage" />
```

Method name–based assemblers are straightforward and easy to use. But can you imagine what would happen if we were to export several Spring beans as MBeans? After a while the list of method names given to the assembler would be huge. And there's also a possibility that we may want to export a method from one bean while another bean has a same-named method that we don't want to export.

Clearly, in terms of Spring configuration, the method name approach doesn't scale well when exporting multiple MBeans. Let's see if using interfaces to expose MBean operations and attributes would be any better.

13.1.2 Using interfaces to define MBean operations and attributes

Spring's InterfaceBasedMBeanInfoAssembler is another MBean info assembler that lets you use interfaces to pick and choose which methods on a bean get exported as MBean-managed operations. It's similar to the method name–based assemblers, except that instead of listing method names to be exported, you list interfaces that define the methods to be exported.

For example, suppose that you were to define an interface named HomeController-ManagedOperations like this:

```
package com.habuma.spitter.jmx;

public interface HomeControllerManagedOperations {
  int getSpittlesPerPage();
  void setSpittlesPerPage(int spittlesPerPage);
}
```

Here you've selected the setSpittlesPerPage() and getSpittlesPerPage methods as the operations that you want to export. Again, these accessor methods will indirectly export the spittlesPerPage property as a managed attribute. To use this assembler, you just need to use the following assembler bean instead of the method name–based assemblers from before:

```
<bean id="assembler"
      class="org.springframework.jmx.export.assembler.
      ➥InterfaceBasedMBeanInfoAssembler"
  p:managedInterfaces=
                "com.habuma.spitter.jmx.HomeControllerManagedOperations"
/>
```

The managedInterfaces property takes a list of one or more interfaces that serve as the MBean-managed operation interfaces—in this case, the HomeControllerManagedOperations interface.

What may not be apparent, but is certainly interesting, is that HomeController doesn't have to explicitly implement HomeControllerManagedOperations. The interface is there for the sake of the exporter, but we don't need to implement it directly in any of our code.

The nice thing about using interfaces to select managed operations is that we could collect dozens of methods into a few interfaces and keep the configuration of InterfaceBasedMBeanInfoAssembler clean. This goes a long way toward keeping the Spring configuration tidy even when exporting multiple MBeans.

Ultimately, those managed operations must be declared somewhere, whether in Spring configuration or in some interface. Moreover, the declaration of the managed operations represents a duplication in code—method names declared in an interface or Spring context and method names in the implementation. This duplication exists for no other reason than to satisfy the MBeanExporter.

One of the things that Java annotations are good at is helping to eliminate such duplication. Let's see how to annotate a Spring-managed bean so that it can be exported as an MBean.

13.1.3 *Working with annotation-driven MBeans*

In addition to the MBean info assemblers I've shown you thus far, Spring provides another assembler known as MetadataMBeanInfoAssembler that can be configured to use annotations to appoint bean methods as managed operations and attributes. I could show you how to use that assembler, but I won't. That's because wiring it up manually is burdensome and not worth the trouble just to be able to use annotations.

Instead, I'm going to show you how to use the <context:mbean-export> element from Spring's context configuration namespace. This handy element wires up an MBean exporter and all of the appropriate assemblers to turn on annotation-driven MBeans in Spring. All you have to do is use it instead of the MBeanExporter bean that we've been using:

```
<context:mbean-export server="mbeanServer" />
```

Now, to turn any Spring bean into an MBean, all we must do is annotate it with @ManagedResource and annotate its methods with @ManagedOperation or @ManagedAttribute. For example, here's how to alter HomeController to be exported as an MBean using annotations.

Listing 13.1 Annotating HomeController to be an MBean

```
package com.habuma.spitter.mvc;
import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jmx.export.annotation.ManagedAttribute;
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import com.habuma.spitter.service.SpitterService;

@Controller
@ManagedResource(objectName="spitter:name=HomeController") //

public class HomeController {

  ...

  @ManagedAttribute   //

  public void setSpittlesPerPage(int spittlesPerPage) {
    this.spittlesPerPage = spittlesPerPage;
  }

  @ManagedAttribute   //

  public int getSpittlesPerPage() {
    return spittlesPerPage;
  }
}
```

Export HomeController as MBean

Expose spittlesPerPage as managed attribute

The `@ManagedResource` annotation is applied at the class level to indicate that this bean should be exported as an MBean. The `objectName` attribute indicates the domain (`spitter`) and name (`HomeController`) of the MBean.

The accessor methods for the `spittlesPerPage` property are both annotated with `@ManagedAttribute` to indicate that it should be exposed as a managed attribute. Note that it's not strictly necessary to annotate both accessor methods. If you choose to only annotate the `setSpittlesPerPage()` method, then you'll still be able to set the property through JMX, but you won't be able to see what its value is. Conversely, annotating `getSpittlesPerPage()` will enable the property's value to be viewed as read-only via JMX.

Also note that it's possible to annotate the accessor methods with `@ManagedOperation` instead of `@ManagedAttribute`. For example:

```
@ManagedOperation
public void setSpittlesPerPage(int spittlesPerPage) {
  this.spittlesPerPage = spittlesPerPage;
}

@ManagedOperation
public int getSpittlesPerPage() {
  return spittlesPerPage;
}
```

This will expose those methods through JMX, but it won't expose the `spittlesPerPage` property as a managed attribute. That's because methods annotated with `@ManagedOperation` are treated strictly as methods and not as JavaBean accessors when it comes to exposing MBean functionality. Thus, `@ManagedOperation` should be reserved for exposing methods as MBean operations and `@ManagedAttribute` should be used when exposing managed attributes.

13.1.4 Handing MBean collisions

So far you've seen how to publish an MBean into an MBean server using several approaches. In all cases, we've given the MBean an object name that's made up of a management domain name and a key-value pair. Assuming that there's not already an MBean published with the name we've given our MBean, we should have no trouble publishing our MBean. But what happens if there's a name collision?

By default, `MBeanExporter` will throw an `InstanceAlreadyExistsException` should you try to export an MBean that's named the same as an MBean that's already in the MBean server. But you can change that behavior by specifying how the collision should be handled via the `MBeanExporter`'s `registrationBehaviorName` property or through `<context:mbean-export>`'s `registration` attribute.

There are three ways that an MBean name collision can be handled:

- Fail if an existing MBean has the same name (this is the default behavior)
- Ignore the collision and don't register the new MBean
- Replace the existing MBean with the new MBean

For example, if you're using `MBeanExporter`, you can configure it to ignore collisions by setting the `registrationBehaviorName` property to `REGISTRATION_IGNORE_EXISTING` like this:

```
<bean id="mbeanExporter"
  class="org.springframework.jmx.export.MBeanExporter">
  <property name="beans">
    <map>
      <entry key="spitter:name=HomeController"
             value-ref="homeController"/>
    </map>
  </property>
  <property name="server" ref="mbeanServer" />

  <property name="assembler" ref="assembler"/>
  <property name="registrationBehaviorName"
            value="REGISTRATION_IGNORE_EXISTING" />
</bean>
```

The `registrationBehaviorName` property accepts `REGISTRATION_FAIL_ON_EXISTING`, `REGISTRATION_IGNORE_EXISTING`, or `REGISTRATION_REPLACING_EXISTING`, each representing one of the three collision-handling behaviors available.

If you're using `<context:mbean-export>` to export annotated MBeans, then you'll use the `registration` attribute to specify collision-handling behavior. For example:

```
<context:mbean-export server="mbeanServer"
    registration="replaceExisting"/>
```

The `registration` attribute accepts `failOnExisting`, `ignoreExisting`, or `replaceExisting`.

Now that we've registered our MBeans using MBeanExporter, we'll need a way to access them for management. As you've seen already, we can use tools like JConsole to access a local MBean server to view and manipulate MBeans. But a tool such as JConsole doesn't lend itself to programmatic management of MBeans. How can we manipulate MBeans in one application from within another application? Fortunately, there's another way to access MBeans as remote objects. Let's explore how Spring's support for remote MBeans will enable us to access our MBeans in a standard way through a remote interface.

13.2 Remoting MBeans

Although the original JMX specification referred to remote management of applications through MBeans, it didn't define the actual remoting protocol or API. Consequently, it fell to JMX vendors to define their own, often proprietary, remoting solutions for JMX.

In response to the need for a standard for remote JMX, the Java Community Process produced *JSR-160*, the Java Management Extensions Remote API Specification. This specification defines a standard for JMX remoting, which at a minimum requires an RMI binding and optionally the *JMX Messaging Protocol (JMXMP)*.

In this section, we'll see how Spring enables remote MBeans. We'll start by configuring Spring to export our `HomeController` MBean as a remote MBean. Then we'll see how to use Spring to manipulate that MBean remotely.

13.2.1 Exposing remote MBeans

The simplest thing we can do to make our MBeans available as remote objects is to configure Spring's `ConnectorServerFactoryBean`:

```
<bean class=
    "org.springframework.jmx.support.ConnectorServerFactoryBean" />
```

`ConnectorServerFactoryBean` creates and starts a JSR-160 `JMXConnectorServer`. By default, the server listens for the JMXMP protocol on port 9875—thus, it's bound to `service:jmx:jmxmp://localhost:9875`. But we're not limited to exporting MBeans using only JMXMP.

Depending on the JMX implementation, you may have several remoting protocol options to choose from, including RMI, SOAP, Hessian/Burlap, and even IIOP. To specify a different remote binding for our MBeans, we just need to set the `serviceUrl` property of `ConnectorServerFactoryBean`. For example, if we wanted to use RMI for MBean remoting, we'd set `serviceUrl` like this:

```
<bean class="org.springframework.jmx.support.ConnectorServerFactoryBean"
    p:serviceUrl=
         "service:jmx:rmi://localhost/jndi/rmi://localhost:1099/spitter" />
```

Here, we're binding it to an RMI registry listening on port 1099 of the localhost. That means that we'll also need an RMI registry running and listening at that port. As you'll recall from chapter 10, `RmiServiceExporter` can start an RMI registry automatically for you. But in this case we're not using `RmiServiceExporter`, so we'll need to start an

RMI registry by declaring an `RmiRegistryFactoryBean` in Spring with the following `<bean>` declaration:

```
<bean class="org.springframework.remoting.rmi.RmiRegistryFactoryBean"
       p:port="1099" />
```

And that's it! Now our MBeans are available through RMI. But there's little point to doing this if nobody will ever access the MBeans over RMI. So let's now turn our attention to the client side of JMX remoting and see how to wire up a remote MBean in Spring.

13.2.2 Accessing remote MBeans

Accessing a remote MBean server involves configuring an `MBeanServerConnectionFactoryBean` in the Spring context. The following bean declaration sets up an `MBeanServerConnectionFactoryBean` that can be used to access the RMI-based remote server we created in the previous section:

```
<bean id="mBeanServerClient"
    class=
        "org.springframework.jmx.support.MBeanServerConnectionFactoryBean"
    p:serviceUrl=
        "service:jmx:rmi://localhost/jndi/rmi://localhost:1099/spitter"/>
```

As its name implies, `MBeanServerConnectionFactoryBean` is a factory bean that creates an `MBeanServerConnection`. The `MBeanServerConnection` produced by `MBeanServerConnectionFactoryBean` acts as a local proxy to the remote MBean server. It can be wired into a bean property just as if it were any other bean:

```
<bean id="jmxClient" class="com.springinaction.jmx.JmxClient">
  <property name="mbeanServerConnection" ref="mBeanServerClient" />
</bean>
```

`MBeanServerConnection` provides several methods that let us query the remote MBean server and invoke methods on the MBeans contained therein. For example, say that we'd like to know how many MBeans are registered in the remote MBean server. The following code snippet will print that information:

```
int mbeanCount = mbeanServerConnection.getMBeanCount();
System.out.println("There are " + mbeanCount + " MBeans");
```

And we may also query the remote server for the names of all of the MBeans using the `queryNames()` method:

```
java.util.Set mbeanNames = mbeanServerConnection.queryNames(null, null);
```

The two parameters passed to `queryNames()` are used to refine the results. Passing in `null` for both parameters indicates that we're asking for the names of all of the registered MBeans.

Querying the remote MBean server for bean counts and names is fun, but doesn't get much work done. The real value of accessing an MBean server remotely is found in accessing attributes and invoking operations on the MBeans that are registered in the remote server.

For accessing MBean attributes, you'll want to use the `getAttribute()` and `setAttribute()` methods. For example, to retrieve the value of an MBean attribute, you'd call the `getAttribute()` method like so:

```
String cronExpression = mbeanServerConnection.getAttribute(
    new ObjectName("spitter:name=HomeController"), "spittlesPerPage");
```

Similarly, changing the value of an MBean attribute can be done using the `setAttribute()` method:

```
mbeanServerConnection.setAttribute(
    new ObjectName("spitter:name=HomeController"),
    new Attribute("spittlesPerPage", 10));
```

If you'd like to invoke an MBean's operation, then the `invoke()` method is what you're looking for. Here's how you might invoke the `setSpittlesPerPage()` method on the `HomeController` MBean:

```
mbeanServerConnection.invoke(
    new ObjectName("spitter:name=HomeController"),
    "setSpittlesPerPage",
    new Object[] { 100 },
    new String[] {"int"});
```

And you can do dozens of other things with remote MBeans by using the methods available through `MBeanServerConnection`. I'll leave it to you to explore the possibilities.

But invoking methods and setting attributes on remote MBeans is awkward when done through `MBeanServerConnection`. Doing something as simple as calling the `setSpittlesPerPage()` method involves creating an `ObjectName` instance and passing several other parameters to the `invoke()` method. This isn't nearly as intuitive as a normal method invocation would be. For a more direct approach, we'll need to proxy the remote MBean.

13.2.3 Proxying MBeans

Spring's `MBeanProxyFactoryBean` is a proxy factory bean in the same vein as the remoting proxy factory beans we examined in chapter 10. But instead of providing proxy-based access to remote Spring-managed beans, `MBeanProxyFactoryBean` lets you access remote MBeans directly (as if they were any other locally configured bean). Figure 13.4 illustrates how this works.

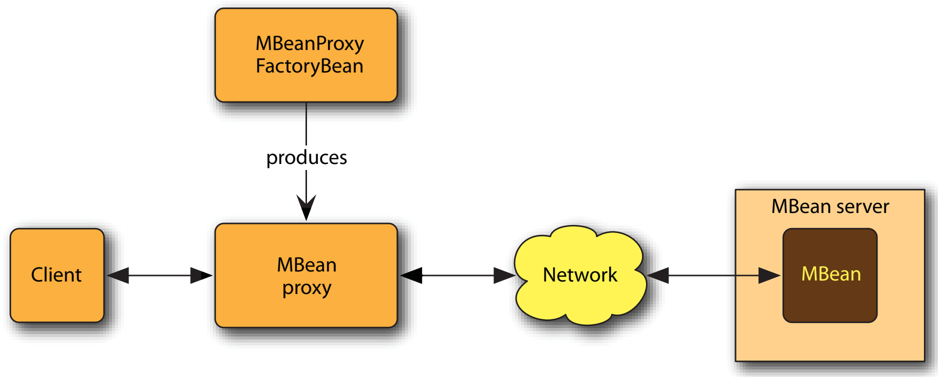


Figure 13.4 **`MBeanProxyFactoryBean` produces a proxy to a remote MBean. The proxy's client can then interact with the remote MBean as if it were a locally configured POJO.**

For example, consider the following declaration of `MBeanProxyFactoryBean`:

```
<bean id="remoteHomeControllerMBean"
      class="org.springframework.jmx.access.MBeanProxyFactoryBean"
      p:objectName="spitter:name=HomeController"
      p:server-ref="mBeanServerClient"
      p:proxyInterface=
        "com.habuma.spitter.jmx.HomeControllerManagedOperations" />
```

The `objectName` property specifies the object name of the remote MBean that's to be proxied locally. Here it's referring to the `HomeController` MBean that we exported earlier.

The `server` property refers to an `MBeanServerConnection` through which all communication with the MBean is routed. Here we've wired in the `MBeanServer-ConnectionFactoryBean` that we configured earlier.

Finally, the `proxyInterface` property specifies the interface that will be implemented by the proxy. In this case, we're using the same `HomeControllerManaged-Operations` interface that we defined in section 13.1.2.

With the `remoteHomeControllerMBean` bean declared, we can now wire it into any bean property whose type is `HomeControllerManagedOperations` and use it to access the remote MBean. From there, we'll be able to invoke the `setSpittlesPerPage()` and `getSpittlesPerPage()` methods.

We've now seen several ways that we can communicate with MBeans, and can now view and tweak our Spring bean configuration while the application is running. But thus far it's been a one-sided conversation. We've talked to the MBeans, but the MBeans haven't been able to get a word in edgewise. It's now time for us to hear what they have to say by listening for notifications.

13.3 Handling notifications

Querying an MBean for information is only one way of keeping an eye on the state of an application. But it's not the most efficient way to be informed of significant events within the application.

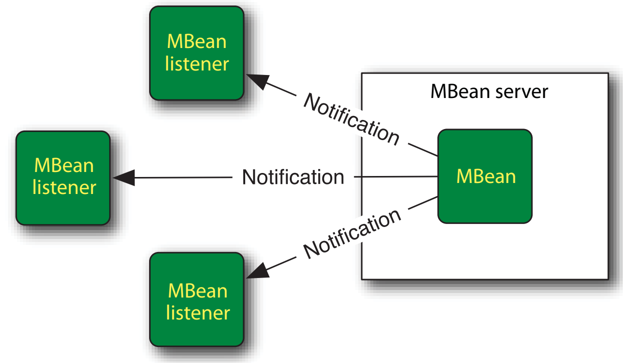


Figure 13.5 JMX notifications enable MBeans to communicate proactively with the outside world.

For example, suppose that the Spitter application were to keep a count of how many spittles have been posted. And suppose that you want to know every time the count has increased by one million spittles (for the 1,000,000th, 2,000,000th, 3,000,000th spittle, and so on). One way to handle this is to write some code that periodically queries the database, counting the number of spittles. But the process that performs that query would keep itself and the database busy as it constantly checks for the spittle count.

Instead of repeatedly querying the database to get that information, a better approach may be to have an MBean notify you when the momentous occasion takes place. JMX notifications, as illustrated in figure 13.5, are a way that MBeans can communicate with the outside world proactively, instead of waiting for some external application to query them for information.

Spring's support for sending notifications comes in the form of the `NotificationPublisherAware` interface. Any bean-turned-MBean that wishes to send notifications should implement this interface. For example, consider `SpittleNotifierImpl`.

Listing 13.2 Using a `NotificationPublisher` to send JMX notifications

```
package com.habuma.spitter.jmx;

import javax.management.Notification;
import org.springframework.jmx.export.annotation.ManagedNotification;
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.jmx.export.notification.NotificationPublisher;
import org.springframework.jmx.export.notification.NotificationPublisherAware;
import org.springframework.stereotype.Component;

@Component
@ManagedResource("spitter:name=SpitterNotifier")
@ManagedNotification(
        notificationTypes="SpittleNotifier.OneMillionSpittles",
        name="TODO")
public class SpittleNotifierImpl
    implements NotificationPublisherAware, SpittleNotifier {

  private NotificationPublisher notificationPublisher;

  public void setNotificationPublisher(
          NotificationPublisher notificationPublisher) {
```

Implement NotificationPublisherAware

Inject notification publisher

```
    this.notificationPublisher = notificationPublisher;
  }

  public void millionthSpittlePosted() {
    notificationPublisher.sendNotification(        <-- Send notification
            new Notification(
                    "SpittleNotifier.OneMillionSpittles", this, 0));
  }

}
```

As you can see, SpittleNotifierImpl implements NotificationPublisherAware. This isn't a demanding interface. It requires only that a single method be implemented: setNotificationPublisher.

SpittleNotifierImpl also implements a single method from the SpittleNotifier interface,[1] millionthSpittlePosted(). This method uses the NotificationPublisher that's automatically injected via the setNotificationPublisher() method to send a notification that another million spittles have been posted.

Once the sendNotification() method has been called, the notification is on its way to… hmm… it seems that we haven't decided who'll receive the notification yet. Let's set up a notification listener to listen to and react to the notification.

13.3.1 Listening for notifications

The standard way to receive MBean notifications is to implement the javax.management.NotificationListener interface. For example, consider PagingNotificationListener:

```
package com.habuma.spitter.jmx;
import javax.management.Notification;
import javax.management.NotificationListener;

public class PagingNotificationListener implements NotificationListener {
  public void handleNotification(Notification notification,
                                 Object handback) {
     // ...
  }
}
```

PagingNotificationListener is a typical JMX notification listener. When a notification is received, its handleNotification() method will be invoked to react to the notification. Presumably, PagingNotificationListener's handleNotification() method will send a message to a pager or cell phone about the fact that another million spittles have been posted. (I've left the actual implementation to the reader's imagination.)

The only thing left to do is register PagingNotificationListener with the MBeanExporter:

[1] For brevity's sake, I'm not showing the SpittleNotifier interface. But as you can imagine, its only method is millionthSpittlePosted.

```
<bean class="org.springframework.jmx.export.MBeanExporter">
  ...
  <property name="notificationListenerMappings">
    <map>
      <entry key="Spitter:name=PagingNotificationListener">
        <bean class="com.habuma.spitter.jmx.PagingNotificationListener" />
      </entry>
    </map>
  </property>
</bean>
```

`MBeanExporter`'s `notificationListenerMappings` property is used to map notification listeners to the MBeans that they'll be listening to. In this case, we've set up `PagingNotificationListener` to listen to any notifications published by the `Spittle-Notifier` MBean.

13.4 Summary

With JMX, you can open a window into the inner workings of your application. In this chapter, we saw how to configure Spring to automatically export Spring beans as JMX MBeans so that their details could be viewed and manipulated through JMX-ready management tools. We also saw how to create and use remote MBeans for times when those MBeans and tools are distant from each other. Finally, we saw how to use Spring to publish and listen for JMX notifications.

By now you've probably noticed that the number of remaining pages in this book is dwindling fast. Our journey through Spring is almost complete. But before we conclude, we have a few more quick stops to make along the way. In the next chapter, we'll explore a handful of Spring features that, although useful, haven't appeared in any chapter up until now, including how to use Spring to access objects in JNDI, send email, and schedule tasks.