Mastering Grails: Grails in the enterprise

Using Grails with JMX, Spring, and log4j

Skill Level: Introductory

Scott Davis (scott@aboutgroovy.com)
Editor in Chief
AboutGroovy.com

16 Dec 2008

In this installment of *Mastering Grails*, series author Scott Davis puts to rest any qualms about Grails' readiness for the enterprise. You'll see how to use Grails with enterprise-caliber libraries including the Java™ Management Extensions (JMX), Spring, and log4j.

I'm often asked if I think that Grails is enterprise-ready. The short answer is yes. But the long answer I typically give is: "Only if you feel that Spring and Hibernate (the underlying technologies that Grails is based on) are ready. Only if you feel that Tomcat or JBoss (or the Java Enterprise Edition (Java EE) application server you are using) is ready. Only if you feel that MySQL or PostgreSQL (or the database that you are using) is ready. Only if you feel that Java programming is enterprise-ready."

British Sky Broadcasting Group recently migrated its public-facing Web sites to Grails. They get 110 million hits a month. LinkedIn.com uses Grails for some of the commercial parts of its site. Tropicana Juice has a Web site in the United Kingdom that has been running on Grails for several years. Grails.org itself is written in Grails, supporting over 70,000 downloads a month. And SpringSource's recent acquisition of G2One (the company behind Groovy and Grails) should allay any lingering doubts as to whether Groovy and Grails are ready for the enterprise.

As exotic as Groovy can seem at times, it's important to remember that it is implemented in plain-old Java code. As different as Grails development is from other typical Java Web frameworks, you still end up with a Java EE-compliant WAR file.

In this article, you'll explore some enterprise-grade tools for monitoring and configuration. You'll learn how to instrument your Grails application with JMX. You'll get a brief introduction to Spring configuration in Grails. You'll also see how the log4j settings are initially specified in Config.groovy, and you'll learn how to adjust them dynamically with JMX.

About this series

Grails is a modern Web development framework that mixes familiar Java technologies like Spring and Hibernate with contemporary practices like convention over configuration. Written in Groovy, Grails give you seamless integration with your legacy Java code while adding the flexibility and dynamism of a scripting language. After you learn Grails, you'll never look at Web development the same way again.

Implementing JMX instrumentation

JMX has been around since 2000. It is one of the oldest JSRs — JSR 3, to be exact. As the Java language gained popularity on the server, the ability to tune and configure a live, running application remotely became a critical part of the platform. In 2004, Sun instrumented the JVM using JMX and shipped supporting tools such as JConsole with the Java 1.5 JDK.

JMX provides introspection on the JVM, the application server, and your classes all through a consistent interface. These various components are exposed to the management console via *managed beans* — MBeans for short.

For more background information on JMX, see "*Java theory and practice*: Instrumenting applications with JMX."

MBeans are like the various gauges, dials, and switches on the dashboard of your car. Sometimes the instruments are read-only, like your speedometer. Other times, like the accelerator, they are writable as well. But this dashboard metaphor breaks down a bit when you consider the fact that MBeans are meant to be managed remotely. Imagine flipping on the turn signal or changing the radio station in your car — remotely.

Enabling a local JMX agent

Local or remote?

For development and testing, running both the JMX agent and client locally is generally the easiest thing to do. In a real production environment, however, the benefits of JMX become apparent when you monitor agents remotely. JConsole takes up the same system resources (RAM, CPU cycles, and so on) that any other Java process does. This can be problematic, especially if the production server you are trying to monitor is already under duress. But more important, being able to monitor multiple servers from a single seat makes you the omniscient ruler of your digital domain.

Of course, monitoring your production servers remotely means securing them properly as well. You can set up password protection or ideally use public/private key authentication (see Resources).

To use JMX for monitoring, you must first enable it. In Java 5, you need to supply a couple of JMX-related flags to the JVM at run time. (In Java 6, these settings should already be in place, although you might choose to supply them anyway to override the defaults.) In JMX vernacular, you are setting up a JMX *agent*. Listing 1 shows the JVM parameters:

Listing 1. JVM parameters to enable JMX monitoring

```
-Dcom.sun.management.jmxremote
-Djava.rmi.server.hostname=localhost
```

Some tutorials advocate creating a global `JAVA_OPTS` environment variable to hold the JMX flags. Others have you type the flags in at the command line: `java -Dcom.sun.management.jmxremote -Djava.rmi.server.hostname=localhost someExampleClass`.

Both suggestions work, but neither is optimal for a production environment. I've found that it is best to set these values in the server's startup script. Having to remember to type these esoteric flags each time you need to restart a server is a fragile solution at best. And setting global variables like `CLASSPATH` and `JAVA_OPTS` should be avoided for two reasons: they add unnecessary additional configuration steps when you clone servers (a consistent startup script is much easier to copy among servers), and they force all Java processes on the same machine to share the same configuration. Yes, you can create a detailed checklist to remind yourself of these niggling configuration details, but documenting complexity is far less effective than removing complexity.

For UNIX®, Linux®, and Mac OS X systems, the Grails startup script is $GRAILS_HOME/bin/grails. Edit this file, adding the two `JAVA_OPTS` lines shown in Listing 2:

Listing 2. Enabling JMX monitoring in the Grails startup script for UNIX, Linux, and Mac OS X

```
#!/bin/sh
DIRNAME='dirname "$0"'
. "$DIRNAME/startGrails"

export JAVA_OPTS="-Dcom.sun.management.jmxremote"
export JAVA_OPTS="$JAVA_OPTS -Djava.rmi.server.hostname=localhost"
```

```
startGrails org.codehaus.groovy.grails.cli.GrailsScriptRunner "$@"
```

For Windows®, the Grails startup script is $GRAILS_HOME/bin/grails.bat. Add the two lines shown in Listing 3 to grails.bat before the call to startGrails.bat:

Listing 3. Enabling JMX monitoring in the Grails startup script for Windows

```


set JAVA_OPTS=-Dcom.sun.management.jmxremote
set JAVA_OPTS=%JAVA_OPTS% -Djava.rmi.server.hostname=localhost
```

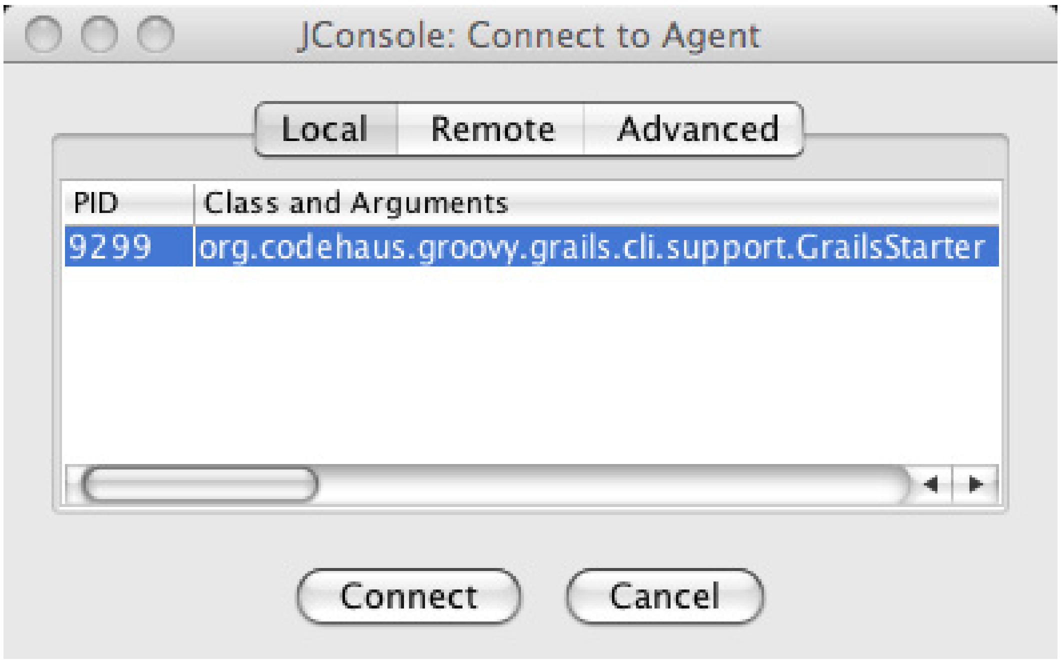
In both scripts, note that the first `JAVA_OPTS` variable assignment overrides the global environment variable if it exists. (The setting is overridden just for this single process — it doesn't reassign the global variable for the entire system.) I do this on purpose to keep global settings from inadvertently polluting my local settings. If you are dependent on global values already in place, be sure to include the existing variable at the start of the assignment as I did in the second line of Listings 2 and 3.

Now, start Grails by typing `grails run-app`. You won't see anything different in the console output, but your application server is now ready to monitor.

You use a JMX client to monitor JMX agents. This can be a desktop GUI like JConsole (included with Java 5 and above) or a Web UI (included with most servers like Tomcat and JBoss). You can even monitor an agent programmatically, as you'll see toward the end of the article.

Open a second command window and type `jconsole`. You should see Grails in the list of local JMX agents, as shown in Figure 1. Click on **Grails**, and then click the **Connect** button.

Figure 1. JConsole listing local JMX agents

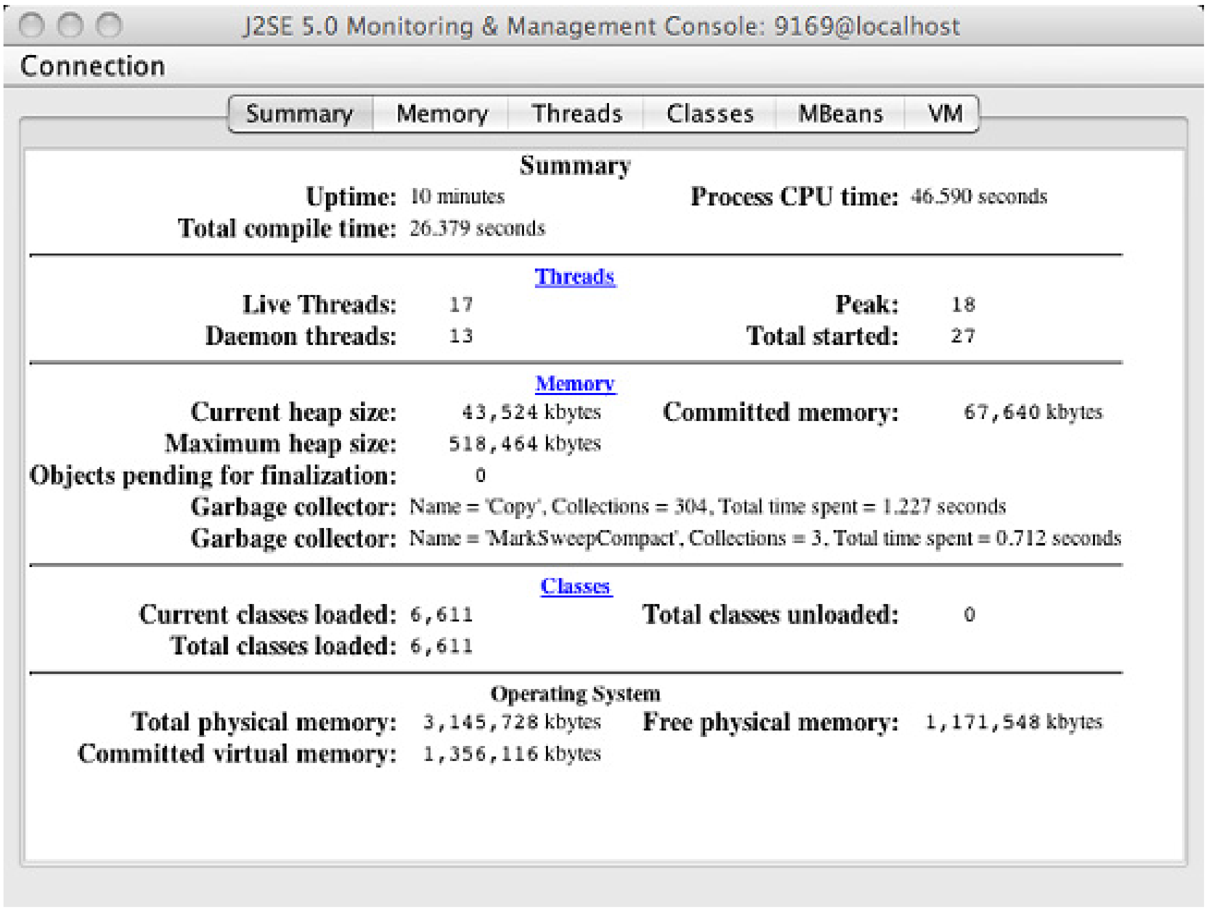


PID	Class and Arguments
9299	org.codehaus.groovy.grails.cli.support.GrailsStarter

Note that for security reasons, local JMX access is available only on Windows systems that use NTFS. If your system uses FAT or FAT32, you might run into problems. Don't worry. In the next section, I'll show you how to set up your JMX agent for remote access. Even though both the agent and the client are technically on the same machine, this will get you past the local security issue.

Once connected, you should see a summary page similar to Figure 2:

Figure 2. The JConsole summary page



```
Summary
                  Uptime: 10 minutes                Process CPU time: 46.590 seconds
      Total compile time: 26.379 seconds

Threads
            Live Threads:   17                                 Peak:   18
         Daemon threads:    13                        Total started:   27

Memory
       Current heap size:   43,524 kbytes    Committed memory:    67,640 kbytes
       Maximum heap size:  518,464 kbytes
Objects pending for finalization:   0
       Garbage collector: Name = 'Copy', Collections = 304, Total time spent = 1.227 seconds
       Garbage collector: Name = 'MarkSweepCompact', Collections = 3, Total time spent = 0.712 seconds

Classes
  Current classes loaded: 6,611        Total classes unloaded:    0
    Total classes loaded: 6,611

Operating System
   Total physical memory:  3,145,728 kbytes   Free physical memory:  1,171,548 kbytes
Committed virtual memory:  1,356,116 kbytes
```

Take a moment to click through the Memory, Threads, Classes, and VM tabs. They give you a real-time view of what is going on inside the JVM. You can see if your server is running low on physical memory, the number of live threads, and even how long the server has been up and running. These tabs are interesting, but your focus will shift in just a moment to the MBeans tab — this is where your classes will appear.

Enabling a remote JMX agent

Don't try this at work

This configuration should never be used in production. I have turned off all authentication and encryption for demonstration purposes. See Resources for detailed instructions on how to secure your JMX Agent for remote management.

To set up your JMX agent to accept remote connections, you need to pass a few more JMX-related flags to the JVM. These additional flags open up a management port and configure security settings (or the lack thereof, in this case).

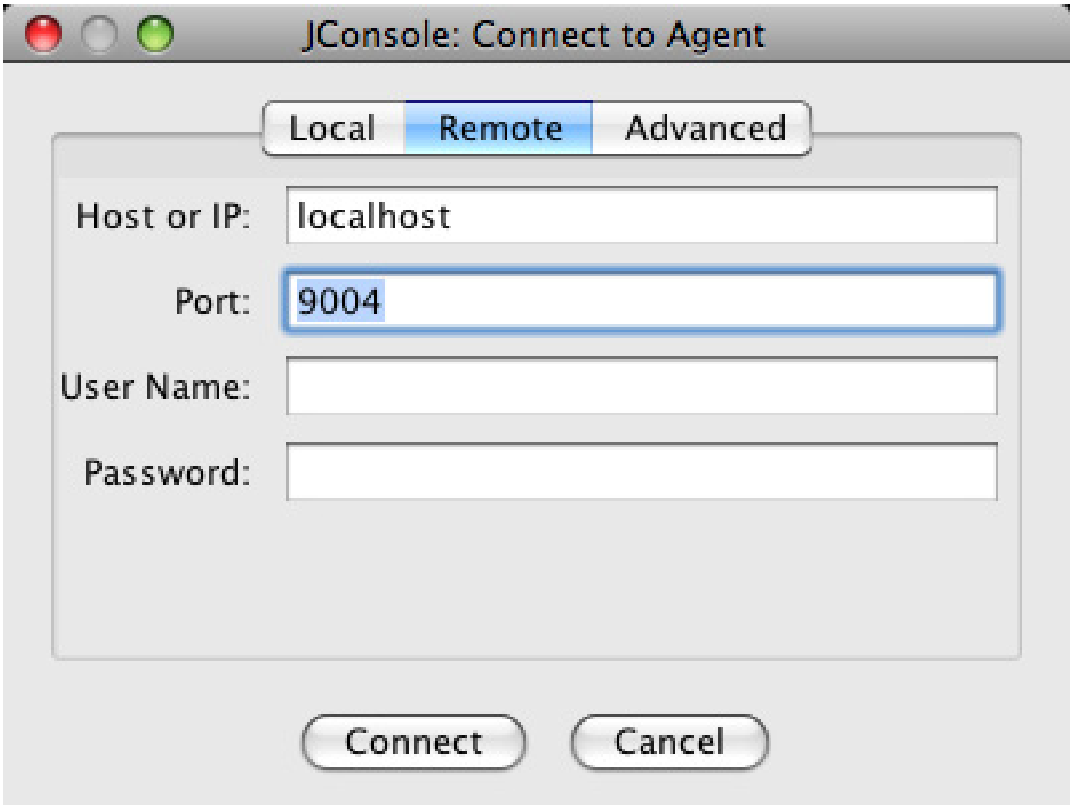
Add three new lines to your Grails startup script, as shown in Listing 4:

Listing 4. Enabling remote JMX monitoring in the Grails startup script

```
export JAVA_OPTS="-Dcom.sun.management.jmxremote"
export JAVA_OPTS=" $JAVA_OPTS -Djava.rmi.server.hostname=localhost"
export JAVA_OPTS=" $JAVA_OPTS -Dcom.sun.management.jmxremote.port=9004"
export JAVA_OPTS=" $JAVA_OPTS -Dcom.sun.management.jmxremote.authenticate=false"
export JAVA_OPTS=" $JAVA_OPTS -Dcom.sun.management.jmxremote.ssl=false"
```

Restart Grails with these new settings. Restart JConsole as well. This time, click on the Remote tab and connect to `localhost` on Port `9004`, as shown in Figure 3:

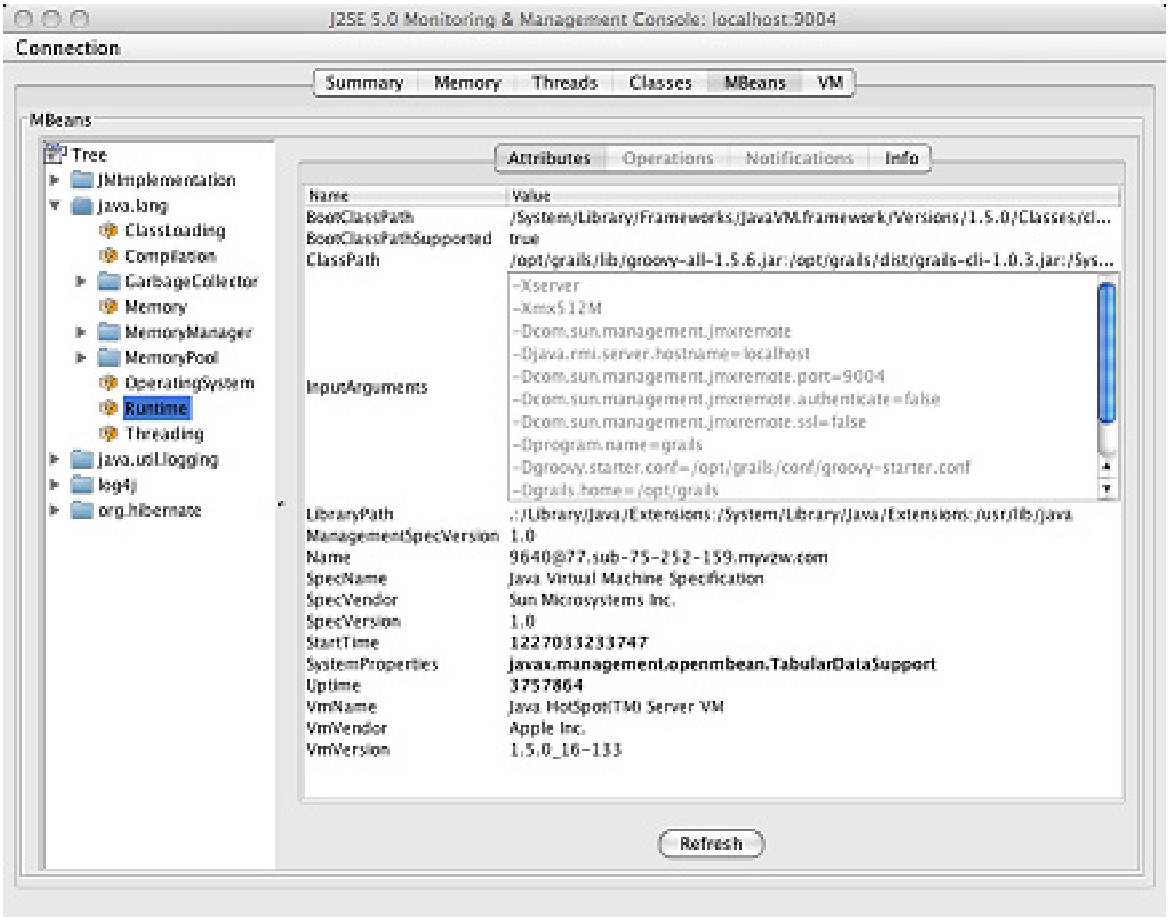
Figure 3. Connecting to a remote JMX agent in JConsole



Host or IP: localhost
Port: 9004
User Name:
Password:

Here's a quick way to verify that you are hitting the remote JVM (even though it is technically running on the same system). Click on the MBeans tab. Expand the java.lang tree on the left. Click on the **Runtime** element. Then, in the Attributes window on the right-hand side of the screen, double-click the **InputArguments**. You should see all of the remote JMX settings shown in Figure 4:

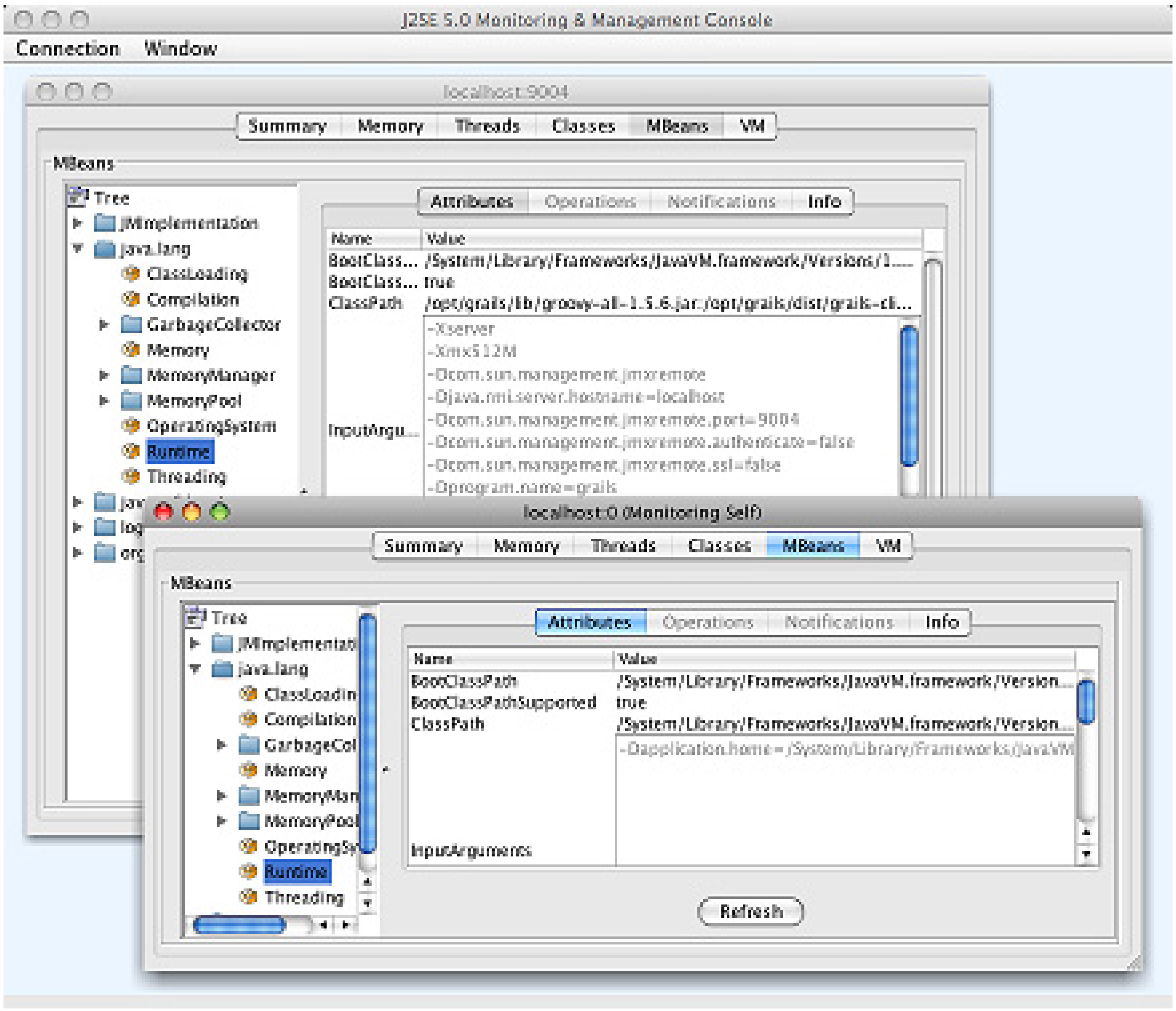
Figure 4. JMX remote agent flags passed to the JVM



Name	Value
BootClassPath	/System/Library/Frameworks/JavaVM.framework/Versions/1.5.0/Classes/cl...
BootClassPathSupported	true
ClassPath	/opt/grails/lib/groovy-all-1.5.6.jar:/opt/grails/dist/grails-cli-1.0.3.jar:/Sys...
InputArguments	-Xserver -Xmx512M -Dcom.sun.management.jmxremote -Djava.rmi.server.hostname=localhost -Dcom.sun.management.jmxremote.port=9004 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false -Dprogram.name=grails -Dgroovy.starter.conf=/opt/grails/conf/groovy-starter.conf -Dgrails.home=/opt/grails
LibraryPath	.:/Library/Java/Extensions:/System/Library/Java/Extensions:/usr/lib/java
ManagementSpecVersion	1.0
Name	9640@77.sub-75-252-159.myvzw.com
SpecName	Java Virtual Machine Specification
SpecVendor	Sun Microsystems Inc.
SpecVersion	1.0
StartTime	1227033233747
SystemProperties	javax.management.openmbean.TabularDataSupport
Uptime	3757864
VmName	Java HotSpot(TM) Server VM
VmVendor	Apple Inc.
VmVersion	1.5.0_16-133

Leave that window open. Open a new connection by clicking on the Connection menu. Click on the Remote tab, and this time accept the defaults (`localhost` on Port `0`). Expand the InputArguments for the Runtime MBean. Notice that the remote JMX flags are not there (as shown in Figure 5):

Figure 5. Monitoring two different JMX agents



Name	Value
BootClass...	/System/Library/Frameworks/JavaVM.framework/Versions/1....
BootClass...	true
ClassPath	/opt/grails/lib/groovy-all-1.5.6.jar:/opt/grails/dist/grails-cli...
InputArgu...	-Xserver -Xmx512M -Dcom.sun.management.jmxremote -Djava.rmi.server.hostname=localhost -Dcom.sun.management.jmxremote.port=9004 -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false -Dprogram.name=grails

Name	Value
BootClassPath	/System/Library/Frameworks/JavaVM.framework/Version...
BootClassPathSupported	true
ClassPath	/System/Library/Frameworks/JavaVM.framework/Version...
InputArguments	-Dapplication.home=/System/Library/Frameworks/JavaVM

If the title bar (Monitoring Self) isn't hint enough, notice that the second JConsole window that you just opened is monitoring the JConsole application itself.

Now that you have JConsole up and monitoring your Grails application, the time has come to do something practical with it like adjusting the logging settings. But before you can do that, you need to understand one last piece of the JMX puzzle: the MBean server.

The MBean server, Grails, and Spring

The Runtime element that you clicked on in JConsole is an MBean. In order for an MBean to be exposed to a JMX client, it must be registered with an MBean server running inside of a JMX agent. Some people use the terms "JMX agent" and "MBean server" interchangeably, but technically the MBean server is one of many components running inside the JMX agent.

To register an MBean programmatically, you call

`MBeanServer.registerMBean()`. However, in Grails this is managed by a configuration file — a Spring configuration file, to be exact.

Spring is at the heart of Grails. It is the dependency-injection framework that controls how all of the classes interact with one another. (For more information on Spring, see Resources.)

From a JMX perspective, you can think, "I am registering this MBean with the MBean server." From the Spring perspective, however, you should think, "I am injecting the MBean into the MBean server." The verbs may be different, but the end result is the same: your MBean becomes visible to the JMX client.

To begin, create a file named resources.xml in grails-app/conf/spring. (You'll see the relationship between resources.groovy and resources.xml later in the article.) Stub out resources.xml as shown in Listing 5:

Listing 5. Setting up the Spring/JMX infrastructure in resources.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

  <bean id="mbeanServer"
        class="org.springframework.jmx.support.MBeanServerFactoryBean">
           <property name="locateExistingServerIfPossible" value="true" />
  </bean>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="server" ref="mbeanServer"/>
    <property name="beans">
      <map>
      </map>
    </property>
  </bean>
</beans>
```

You can reboot Grails now if you'd like to make sure that the basic configuration is correct, but only half of the puzzle is in place: you have an MBean server but no MBeans. The two beans you see here — `mbeanServer` and `exporter` — are the infrastructure you need to register MBeans. The `mbeanServer` bean holds a reference to the existing MBean server. The `mbeanServer` bean gets injected into the `exporter` bean — the class that exposes the list of MBeans to JMX clients like JConsole. The only thing left to do now is register an MBean by adding it to the map of beans inside the `exporter` bean. You'll do that in the next section.

Using log4j with Grails

Open grails-app/conf/Config.groovy to see the log4j settings (as shown in Listing 6):

Listing 6. log4j settings in Config.groovy

```
log4j {
    appender.stdout = "org.apache.log4j.ConsoleAppender"
    appender.'stdout.layout'="org.apache.log4j.PatternLayout"
    appender.'stdout.layout.ConversionPattern'='[%r] %c{2} %m%n'
    // and so on...
}
```

When you start a Grails application, most of the messages that fly by on the command prompt are log4j messages. You have the `org.apache.log4j.ConsoleAppender` to thank for that. (For more on log4j basics, see Resources.)

Registering the log4j MBean

If you want to adjust the logging settings of a Grails application without JMX, you simply edit this file and restart the server. But what if you'd prefer to adjust these settings without rebooting the server or want to adjust them remotely? This sounds like a perfect candidate for JMX. Luckily, log4j ships with an MBean to facilitate these tasks. All you need to do is register the log4j MBean.

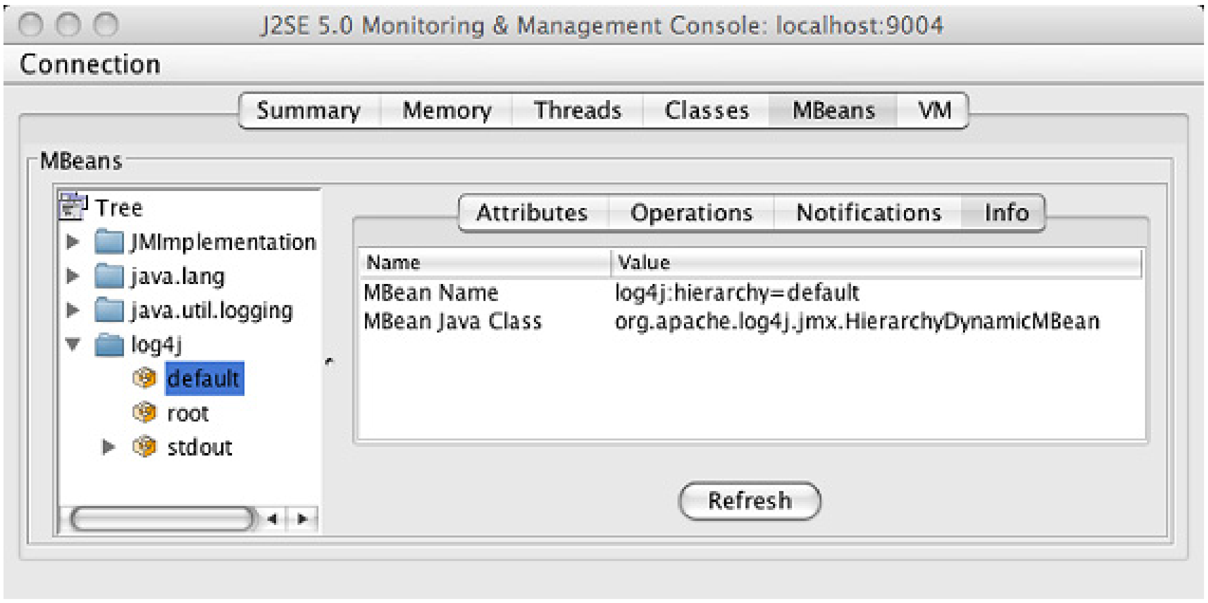
Add the XML for the `entry` (as shown in Listing 7) to resources.xml. This injects the log4j MBean into the MBean server.

Listing 7. Injecting an MBean into an MBean server

```
<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
  <property name="server" ref="mbeanServer"/>
  <property name="beans">
    <map>
      <entry key="log4j:hierarchy=default">
        <bean class="org.apache.log4j.jmx.HierarchyDynamicMBean"/>
      </entry>
    </map>
  </property>
</bean>
```

Reboot Grails, and then restart JConsole. If you connect to `localhost` on Port `9004`, your new log4j MBean should appear on the MBeans tab. Expand the log4j tree element, click on default, and then click on the Info tab. You should recognize the configuration snippets from the entry you just added to resources.xml (see Figure 5):

Figure 6. Viewing the default MBean information



Name	Value
MBean Name	log4j:hierarchy=default
MBean Java Class	org.apache.log4j.jmx.HierarchyDynamicMBean

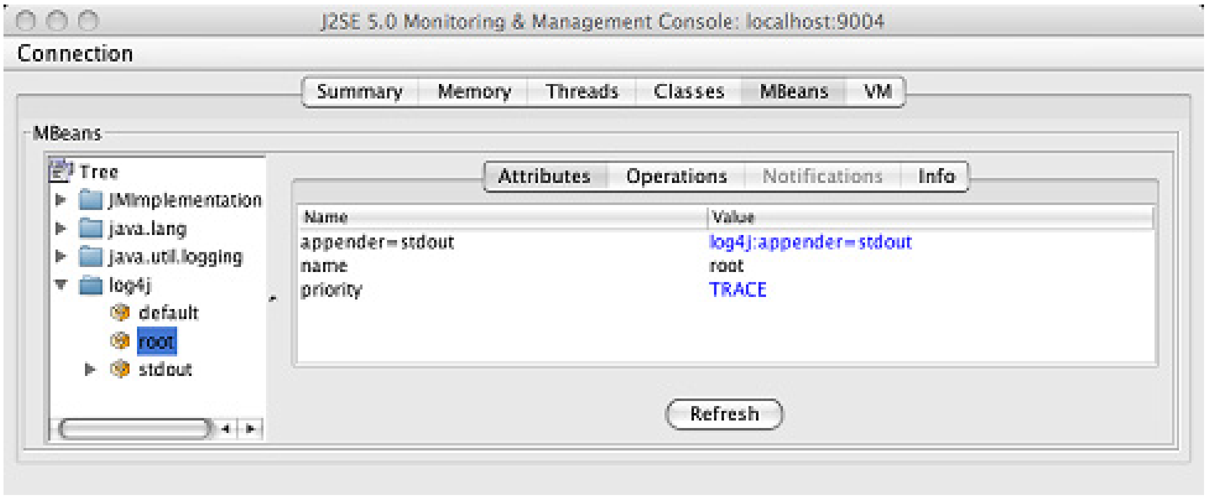
Now that you can see log4j through JMX, the next step is to adjust some of the logging settings.

Changing log4j settings on the fly

Pretend for a moment that your Grails application is acting oddly. You'd like to have a better idea of what is going on under the covers. Looking at grails-app/conf/Config.groovy, you see that the root logger is sending its output to the console, but the filter is set to `error` — `rootLogger="error,stdout"`. You'd like to reset the log level to `trace` to increase the amount of console output.

Take a look at JConsole. Under the log4j folder, you should see the root MBean. You can see that the priority attribute is set to `ERROR`, just as it is in Config.groovy. Double-click on the `ERROR` value and type in `TRACE`, as shown in Figure 6:

Figure 7. Changing the root logger priority from ERROR to TRACE



Name	Value
appender=stdout	log4j:appender=stdout
name	root
priority	TRACE

To verify that your console is more chatty than before, click on the link to the `AirportMappingController` on the home page of your Grails application in a browser. Amidst the avalanche of new output, you should find some details about what Grails actually does to bring up the initial list. See Listing 8 for a sample:

Listing 8. Increased log4j output

```
[11277653] metaclass.RedirectDynamicMethod
  Dynamic method [redirect] looking up URL mapping for
  controller [airportMapping] and action [list] and
  params [["action":"index", "controller":"airportMapping"]]
  with [URL Mappings
------------
org.codehaus.groovy.grails.web.mapping.ResponseCodeUrlMapping@1bab0b
/rest/airport/(*)?
/(*)/(*)?/(*)?
]
[11277653] metaclass.RedirectDynamicMethod Dynamic method
  [redirect] mapped to URL [/trip/airportMapping/list]
[11277653] metaclass.RedirectDynamicMethod Dynamic method
  [redirect] forwarding request to [/trip/airportMapping/list]
[11277653] metaclass.RedirectDynamicMethod Executing redirect
  with response
  [com.opensymphony.module.sitemesh.filter.PageResponseWrapper@19243f]
```

When can you safely ignore a Fatal Error?

If you have been running Grails 1.0.3 for some time, you may have noticed a mysterious error that shows up frequently in the console output — `[Fatal Error] :-1:-1: Premature end of file`. Most folks just ignore it because it doesn't really seem to cause any errors, fatal or otherwise.

If you turn the log level up to `trace`, you can see the details surrounding the supposedly fatal error: `converters.XMLParsingParameterCreationListener Error parsing incoming XML request: Error parsing XML`.

As the more verbose log output explains, Grails is trying to parse every incoming request as if it were XML. Most requests aren't XML, so the request handler duly reports the error but processes the request correctly anyway.

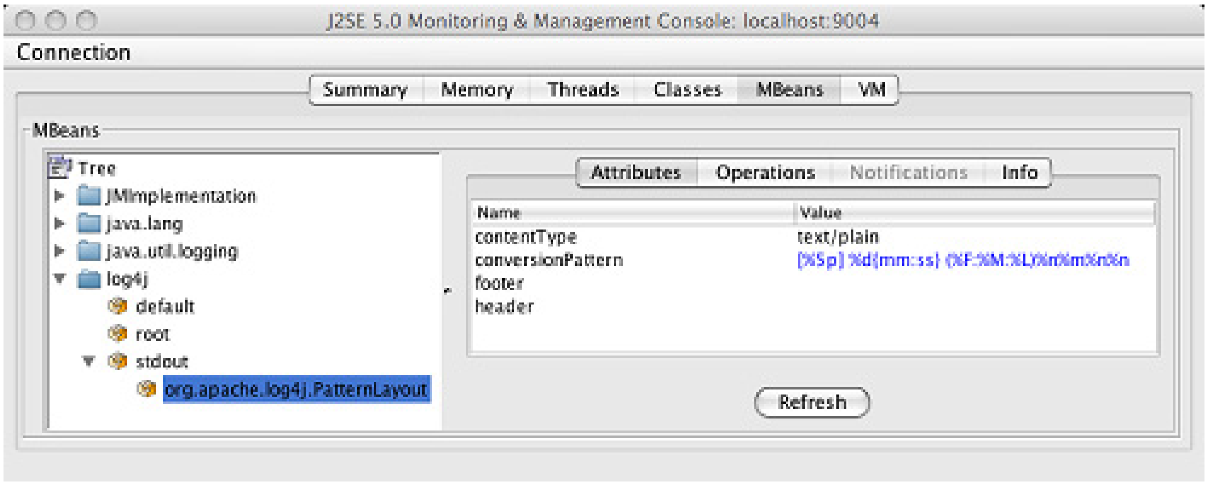
This "little bug that cried wolf" is fixed in version 1.0.4.

Changing the log4j ConversionPattern

Now you'd like to change the pattern used for the output. In Config.groovy, the pattern is set using this line: `appender.'stdout.layout.ConversionPattern'='[%r] %c{2} %m%n'`. Looking at the log4j documentation, you decide to set it to something more descriptive.

Click on the **stdout** MBean in JConsole. Change the `conversionPattern` attribute from its original value to `[%5p] %d{hh:mm:ss} (%F:%M:%L)%n%m%n%n`. After you generate some new log output, I'll describe what this magic incantation does. (See Resources for more information on setting the `conversionPattern`.)

Figure 8. Changing the conversionPattern in PatternLayout



Name	Value
contentType	text/plain
conversionPattern	[%5p] %d{mm:ss} (%F:%M:%L)%n%m%n%n
footer	
header	

Now click on the home link and the `AirportMappingController` link again in your Web browser. The format of the output changes dramatically, as shown in Listing 9:

Listing 9. Console output using the new conversionPattern

```
[DEBUG] 09:04:47 (RedirectDynamicMethod.java:invoke:127)
Dynamic method [redirect] looking up URL mapping for controller
[airportMapping] and action [list] and params
[["action":"index", "controller":"airportMapping"]] with [URL Mappings
------------
org.codehaus.groovy.grails.web.mapping.ResponseCodeUrlMapping@e73cb7
/rest/airport/(*)?
/(*)/(*)?/(*)?
]

[DEBUG] 09:04:47 (RedirectDynamicMethod.java:invoke:144)
Dynamic method [redirect] mapped to URL [/trip/airportMapping/list]

[DEBUG] 09:04:47 (RedirectDynamicMethod.java:redirectResponse:162)
Dynamic method [redirect] forwarding request to [/trip/airportMapping/list]

[DEBUG] 09:04:47 (RedirectDynamicMethod.java:redirectResponse:168)
Executing redirect with response
   [com.opensymphony.module.sitemesh.filter.PageResponseWrapper@47b2e7]
```

Now that you can see the output, here's what is going on. %p writes out the priority level. These messages are clearly `DEBUG` level. `%d{hh:mm:ss}` shows the date stamp in hours:minutes:seconds. `(%F:%M:%L)` puts the filename, method, and line number in parentheses. Finally, `%n%m%n%n` writes a new line, the message, and two more new lines.

No changes that you make to log4j via JMX are permanent. If you reboot Grails, it reverts to the persistent settings in Config.groovy. This means that you can play around with the JMX settings all you like without worrying about messing things up permanently. In the case of `ConversionPatterns`, using JMX is a great way to experiment with the setting until you find the one that you like best. Just don't forget to copy the pattern back into Config.groovy to make the change permanent.

Looking at Hibernate DEBUG output

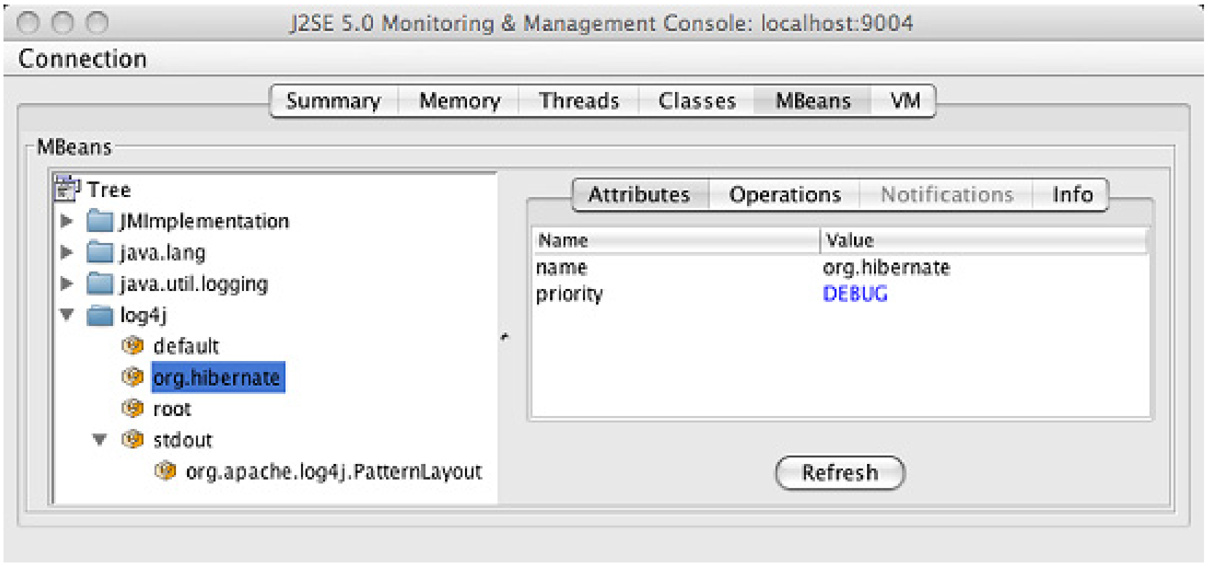
Getting back to the hypothetical story of you debugging a live Grails application, you still haven't found what you are looking for yet. Set the priority attribute of the root MBean back to `ERROR` to cut down the noise level.

Maybe the problem is in Hibernate. Looking back at Config.groovy, you notice that logging output for the `org.hibernate` package is set to `off`. Rather than cranking up the output level for the entire application, perhaps focusing in on a specific package will yield more information.

In JConsole, click on the default MBean. In addition to changing attribute values, you can also call methods on an MBean. Click on the Operations tab. Type `org.hibernate` for the name parameter and click on the **addLoggerMBean** button. You should see a new MBean appear in the tree on the left.

Click on the new **org.hibernate** MBean and change the priority attribute to `DEBUG`, as shown in Figure 8:

Figure 9. Changing the priority on the org.hibernate MBean



Name	Value
name	org.hibernate
priority	DEBUG

Now go back to your Web browser, click on the home link, and click on `AirportMappingController` again. You should see a long series of `DEBUG` log statements, as shown in Listing 10:

Listing 10. Hibernate log4j output

```
[DEBUG] 10:05:52 (AbstractBatcher.java:logOpenPreparedStatement:366)
about to open PreparedStatement (open PreparedStatements: 0, globally: 0)

[DEBUG] 10:05:52 (ConnectionManager.java:openConnection:421)
opening JDBC connection

[DEBUG] 10:05:52 (AbstractBatcher.java:log:401)
select this_.airport_id as airport1_0_0_, this_.locid as locid0_0_,
this_.latitude as latitude0_0_, this_.longitude as longitude0_0_,
this_.airport_name as airport5_0_0_, this_.state as state0_0_
from usgs_airports this_ limit ?

[DEBUG] 10:05:52 (AbstractBatcher.java:logOpenResults:382)
about to open ResultSet (open ResultSets: 0, globally: 0)

[DEBUG] 10:05:52 (Loader.java:getRow:1173)
result row: EntityKey[AirportMapping#1]

[DEBUG] 10:05:52 (Loader.java:getRow:1173)
result row: EntityKey[AirportMapping#2]
```

Take a moment to scroll through the Hibernate `DEBUG` output. You'll get a detailed, step-by-step look at what happens when your data is culled from the database and transformed into an `ArrayList` of beans.

Using the Spring Bean Builder

Now that you know how to configure JMX through resources.xml, it's time to add a new twist. Grails supports Spring configuration through an alternate file: resources.groovy. Rename grails-app/conf/spring/resources.xml to resources.xml.old. Add the code shown in Listing 11 to resources.groovy:

Listing 11. Configuring Spring using Bean Builder

```
import org.springframework.jmx.support.MBeanServerFactoryBean
import org.springframework.jmx.export.MBeanExporter
import org.apache.log4j.jmx.HierarchyDynamicMBean

beans = {
  log4jBean(HierarchyDynamicMBean)

  mbeanServer(MBeanServerFactoryBean) {
    locateExistingServerIfPossible=true
  }

  exporter(MBeanExporter) {
    server = mbeanServer
          beans = ["log4j:hierarchy=default":log4jBean]
  }
}
```

As you can see, the Spring beans are being configured in Groovy code instead of XML. You saw the Groovy `MarkupBuilder` in action in "Grails and legacy databases" as well as "RESTful Grails." This is a slight variation on the theme — a Bean Builder specifically defines beans for Spring configuration.

Reboot Grails and JConsole. Confirm that nothing has changed from the XML configuration.

Using the XML dialect to configure Spring puts the collective wisdom of the Web at your fingertips — you can copy and paste snippets from a wide variety of sources. But using the Bean Builder dialect is more in line with the rest of the configuration in Grails. By this point in your Grails career, you've seen DataSource.groovy, Config.groovy, BootStrap.groovy, and Events.groovy, to name just a few. You are doing your configuration in code, which means that you can do things like conditionally expose an MBean based on the environment you are running in.

For example, Listing 12 shows you how to expose the `log4jBean` in production but hide it in development:

Listing 12. Conditionally exposing JMX beans

```
import org.springframework.jmx.support.MBeanServerFactoryBean
import org.springframework.jmx.export.MBeanExporter
import org.apache.log4j.jmx.HierarchyDynamicMBean
import grails.util.GrailsUtil

beans = {
  log4jBean(HierarchyDynamicMBean)

  mbeanServer(MBeanServerFactoryBean) {
    locateExistingServerIfPossible=true
  }

  switch(GrailsUtil.environment){
    case "development":
    break

    case "production":
      exporter(MBeanExporter) {
        server = mbeanServer
              beans = ["log4j:hierarchy=default":log4jBean]
      }
    break
  }
}
```

Type `grails run-app` and confirm in JConsole that the log4j MBean doesn't show up in development mode. Now type `grails prod run-app` (or `grails war` and deploy the WAR file to the application server of your choice). The MBean should be waiting for you when you relaunch JConsole.

JMX in Groovy

The last thing I'll show you how to do is tweak your JMX MBeans programmatically. As nice as the JConsole GUI is, it's even nicer being able to make changes from a Groovy script.

To begin, create a file named testJmx.groovy. Add the code in Listing 13 to it:

Listing 13. Calling a remote JMX agent in Groovy

```
import javax.management.MBeanServerConnection
import javax.management.remote.JMXConnectorFactory
import javax.management.remote.JMXServiceURL

def agentUrl = "service:jmx:rmi:///jndi/rmi://localhost:9004/jmxrmi"
def connector = JMXConnectorFactory.connect(new JMXServiceURL(agentUrl))
def server = connector.mBeanServerConnection

println "Number of registered MBeans: ${server.mBeanCount}"

println "\nRegistered Domains:"
server.domains.each{println it}

println "\nRegistered MBeans:"
server.queryNames(null, null).each{println it}
```

If Grails is running, you should see the output shown in Listing 14:

Listing 14. Output from the testJmx.groovy script

```
$ groovy testJmx.groovy
Number of registered MBeans: 20

Registered Domains:
java.util.logging
JMImplementation
java.lang
log4j

Registered MBeans:
java.lang:type=MemoryManager,name=CodeCacheManager
java.lang:type=Compilation
java.lang:type=GarbageCollector,name=Copy
java.lang:type=MemoryPool,name=Eden Space
log4j:appender=stdout
java.lang:type=Runtime
log4j:hierarchy=default
log4j:logger=root
log4j:appender=stdout,layout=org.apache.log4j.PatternLayout
java.lang:type=ClassLoading
java.lang:type=MemoryPool,name=Survivor Space
java.lang:type=Threading
java.lang:type=GarbageCollector,name=MarkSweepCompact
java.util.logging:type=Logging
java.lang:type=Memory
java.lang:type=OperatingSystem
java.lang:type=MemoryPool,name=Code Cache
java.lang:type=MemoryPool,name=Tenured Gen
java.lang:type=MemoryPool,name=Perm Gen
JMImplementation:type=MBeanServerDelegate
```

A word of warning

The testJmx.groovy script might throw a `groovy.lang.MissingMethodException` similar to Listing 15:

Listing 15. A possible JMX exception

```
Caught: groovy.lang.MissingMethodException: No signature of method:
 javax.management.remote.rmi.RMIConnector$RemoteMBeanServerConnection.queryNames()
 is applicable for argument types: (java.lang.String, null)
```

If this happens, delete mx4j-3.0.2.jar from $GROOVY_HOME/lib. It is included in the Groovy distribution to bring JMX support to the 1.4 JDK, but it conflicts with later versions of the Java platform.

The interesting part of this script comes from the `javax.management.MBeanServer` returned from the `connector.mBeanServerConnection` call. (Remember that a `getFoo()` method call in Java can be shortened to `foo` in Groovy.) Calling `server.mBeanCount` returns the number of registered MBeans. Calling `server.domains` returns a `String[]` of domain names. Domain names are the first part of the MBean identifier — the comma-delimited list of name/value pairs fully qualifies the name. Calling `server.queryNames(null, null)` returns a `Set` of all of the registered MBeans. (For more information on the methods available on an `MBeanServer` class, see Resources.)

To get a specific MBean, add the code in Listing 16 to the bottom of the script:

Listing 16. Getting an MBean

```
println "\nHere is the Runtime MBean:"
def mbean = new GroovyMBean(server, "java.lang:type=Runtime")
println mbean
```

Once you have a connection to an MBean server and know the name of an MBean, getting a new `GroovyMBean` is a one-line exercise. Listing 17 shows the script output:

Listing 17. GroovyMBean output

```
Here is the Runtime MBean:
MBean Name:
  java.lang:type=Runtime

Attributes:
  (r) javax.management.openmbean.TabularData SystemProperties
  (r) java.lang.String VmVersion
  (r) java.lang.String VmName
  (r) java.lang.String SpecName
  (r) [Ljava.lang.String; InputArguments
  (r) java.lang.String ManagementSpecVersion
  (r) java.lang.String SpecVendor
  (r) long Uptime
  (r) long StartTime
  (r) java.lang.String LibraryPath
  (r) java.lang.String BootClassPath
  (r) java.lang.String VmVendor
  (r) java.lang.String ClassPath
  (r) java.lang.String SpecVersion
  (r) java.lang.String Name
  (r) boolean BootClassPathSupported
```

Do you remember the `InputArguments` from early in the article? They are all of the `-D` parameters passed to the JVM. You used them to confirm that you were, indeed, connected to the remote JMX agent. Add two more lines of code (as shown in Listing 18) to print out the `String[]`:

Listing 18. Getting the InputArguments from the runtime MBean

```
println "\nHere are the InputArguments:"
mbean.InputArguments.each{println it}
```

If you see the output in Listing 19, you know that you have come full circle:

Listing 19. Displaying the InputArguments

```
Here are the InputArguments:
-Xserver
-Xmx512M
-Dcom.sun.management.jmxremote
-Djava.rmi.server.hostname=localhost
-Dcom.sun.management.jmxremote.port=9004
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
-Dprogram.name=grails
-Dgroovy.starter.conf=/opt/grails/conf/groovy-starter.conf
-Dgrails.home=/opt/grails
-Dbase.dir=.
-Dtools.jar=/Library/Java/Home/lib/tools.jar
```

For more information on `GroovyMBeans`, see Resources.

Conclusion

Grails is ready for the enterprise. Common enterprise libraries like JMX, Spring, and log4j are available in Grails because, despite appearances to the contrary, you are still doing traditional Java EE development.

This article brings to a close a year's worth of columns exploring the Trip Planner application. I wanted to keep the domain consistent across the articles so that the focus could remain on core Grails functionality. I'll preserve this spirit in next year's columns, but I also want to expand the horizon to include a variety of Grails applications.

For example, next month I'll introduce a new blogging system. You'll get a quick refresher on how to bootstrap a new Grails application, but it will be anything but a rehash of old material. You'll revisit the RESTful side of Grails, but in the context of setting up a full Atom infrastructure. You'll use JSON and Ajax again but this time to enable calendars and tag-clouds. And after a couple of months, I'll roll out another new idea.

Grails continues to gain mainstream acceptance with each new Web site. The hallmark of a mature Web framework is seeing it used in a variety of ways. Next year's *Mastering Grails* articles will demonstrate the diversity of Web sites possible in Grails. Until then, have fun mastering Grails.

Resources

Learn

- *Mastering Grails*: Read more in this series to gain a further understanding of Grails and all you can do with it.
- Grails: Visit the Grails Web site.
- *Groovy Recipes* (Scott Davis, Pragmatic Programmers, 2007): Learn more about Groovy and Grails in Scott Davis' latest book.
- Grails Framework Reference Documentation: The Grails bible.
- "*Java theory and practice*: Instrumenting applications with JMX" (Brian Goetz, developerWorks, September 2006): Learn more about getting visibility into the JVM and your classes with JMX.
- Monitoring and Management Using JMX: Take a look at Sun's guide to JMX monitoring.
- "Log4j delivers control over logging" (Ceki Gulcu, developerWorks, January 2001): The log4j project's founder describes the log4j API, its unique features, and its design rationale.
- `PatternLayout`: Check out the Javadoc for this class.
- `MBeanServer`: Learn more about the methods available on an `MBeanServer` class.
- Groovy and JMX: Get more information on `GroovyMBeans`.
- Grails Bean Builder: Programmatically build your Spring beans.
- *Practically Groovy*: This developerWorks series is dedicated to exploring the practical uses of Groovy and teaching you when and how to apply them successfully.
- Groovy: Learn more about Groovy at the project Web site.
- AboutGroovy.com: Keep up with the latest Groovy news and article links.
- Technology bookstore: Browse for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Grails: Download the latest Grails release.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Scott Davis
Scott Davis is an internationally recognized author, speaker, and software developer. His books include *Groovy Recipes: Greasing the Wheels of Java*, *GIS for Web Developers: Adding Where to Your Application*, *The Google Maps API*, and *JBoss At Work*.

Trademarks