

# **Instrumented Spring Framework Reference Guide**

**Version 2.5.6.RELEASE**

*Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.*

**Published April 2009**

## Table of Contents

Preface .....	iv
1. Introduction .....	1
1.1. What this guide covers .....	1
1.2. What Instrumented Spring Framework requires to run .....	1
1.3. Intended audience .....	1
1.4. Components of Instrumented Spring Framework .....	1
Spring Framework jars woven with Instrumentation .....	1
Spring Framework Instrumentation .....	1
Management API .....	2
1.5. How Spring Framework Instrumentation works .....	2
Auto-Discovery .....	2
Export to JMX .....	2
Monitoring .....	3
2. Managed Beans .....	4
2.1. Overview of Spring Framework Instrumentation .....	4
Spring Core .....	4
Spring DAO .....	4
Spring ORM .....	4
Spring JEE Integration .....	5
Spring MVC .....	5
2.2. Spring Framework Instrumentation details .....	5
3. Instrumenting Applications .....	12
3.1. Introduction .....	12
3.2. Adding Instrumented Spring Framework jars to applications .....	12
3.3. Configuring applications for auto-discovery .....	13
Configuring web applications for auto-discovery .....	13
Configuring standalone applications for auto-discovery .....	14
Configuring applications for managed resource discovery .....	14
3.4. Configuring applications for monitoring (optional) .....	15
3.5. Configuring applications for remote JMX access .....	16
JMX in SpringSource dm Server .....	16
JMX in SpringSource tc Server .....	16
JMX in WebLogic .....	16
JMX in WebSphere .....	16
JMX in Apache Tomcat .....	17
JMX in JBoss .....	17
JMX in standalone applications .....	17
3.6. Deploying applications .....	18
Verifying auto-discovery through logging .....	18
Verifying auto-discovery through JConsole .....	18

- 4. Managing Custom Components .....20
  - 4.1. Introduction .....20
  - 4.2. Exporting components to JMX .....20
    - Source-level metadata types .....21
    - Example: Exporting a custom component .....22
- 5. Integrating with SpringSource AMS .....24
- 6. Using Instrumented Spring Framework in SpringSource dm Server .....25
  - 6.1. Using Instrumented Spring Framework in dm Server 1.0 .....25
  - 6.2. Using Instrumented Spring Framework in dm Server 2.0 .....25

# Preface

Instrumented Spring Framework is a commercial version of the Spring Framework product, instrumented for management and monitoring. Simply by using the instrumented jars in place of the Spring Framework jars, a number of Spring Beans in your application will be automatically exported to JMX for management and automatically monitored for performance using AOP. This reference guide explains all of the data collected and exposed by Instrumented Spring Framework, how it works, and how to use Instrumented Spring Framework in your application.

# 1. Introduction

## 1.1. What this guide covers

This guide covers all of the data collected and exposed via AOP and JMX by Instrumented Spring Framework. It covers how the instrumentation auto-discovers, exports, and monitors your application components. It covers how to easily export your own components to the same JMX management model from within your application. It covers how the performance data exposed by Instrumented Spring Framework can be easily consumed by SpringSource AMS, providing a robust application and system management and monitoring solution in production. It also covers how to use Instrumented Spring Framework to develop OSGi-based applications for use in SpringSource dm Server.

## 1.2. What Instrumented Spring Framework requires to run

Java 5.0 or higher

The third party jar files included in the Instrumented Spring Framework distribution lib directory

## 1.3. Intended audience

This document is mostly intended for Application Developers who wish to monitor performance and utilization and have runtime control of their applications in all phases of the application lifecycle. The section on [Managed Beans](#) is also intended for Administrators and Operators, as the exposed data can be used by SpringSource AMS or other management systems to provide more granular monitoring of applications in production.

## 1.4. Components of Instrumented Spring Framework

### Spring Framework jars woven with Instrumentation

All of the `org.springframework.<module>.instrumented` jars distributed with this product are simply compile-time woven versions of their counterparts in open source Spring Framework. For example, the `org.springframework.beans.instrumented` jar file is a drop-in replacement for the `org.springframework.beans` jar distributed with open source Spring Framework. The instrumented jars may have compile-time dependencies on the Instrumentation and Management API jars described below.

### Spring Framework Instrumentation

All of the `com.springsource.management.instrumentation.<module>` jars distributed with this product contain the AspectJ aspects used to weave the Spring Framework jars at compile time. Thus, an instrumented Spring Framework jar will likely have a compile-time dependency on 0 to n of these instrumentation jars.

## Management API

All of the `com.springsource.management.agent.<module>` and `com.springsource.management.adapter.<module>` jars distributed with this product contain the API used to auto-discover Spring beans, export them to JMX, and obtain and store monitoring data. Thus, a Spring Framework Instrumentation jar will likely have a compile-time dependency on 0 to n of these management API jars.

# 1.5. How Spring Framework Instrumentation works

## Auto-Discovery

Instrumented Spring Framework automatically discovers applications by advising the refresh method of the `AbstractApplicationContext`. Any bean that is created through an `AbstractApplicationContext` will be discovered as a managed resource if the instrumented components know how to manage and monitor it (see [Managed Beans](#) for a list of recognized components). There are some components created outside of `ApplicationContexts` (such as the `DispatcherServlet`) that will also be auto-discovered. When an `AbstractApplicationContext` is closed, its associated managed resources will be undeployed.

## Export to JMX

Once managed resources are auto-discovered, the Spring Framework Instrumentation automatically creates JMX `ModelMBeans` representing each application resource it discovers, and registers these `ModelMBeans` in an auto-detected `MBeanServer`. These `MBeans` contain pre-determined attributes representing either metrics or properties of the resource being managed and operations that provide runtime control of the resource being managed.

All `MBeans` are registered under the domain "spring.application". They are segmented by application (using an "application" key property) and contain "type" and "name" key properties. The "type" key property value typically refers to the component class (for example, "DispatcherServlet") and the "name" key property value is usually set to the bean name.



## 2. Managed Beans

### 2.1. Overview of Spring Framework Instrumentation

Instrumented Spring Framework automatically exposes a number of metrics, attributes, and runtime operations for many of the Spring components used in your application. The following section highlights just some of the data and control operations available to you out of the box with Instrumented Spring Framework. Please see [Spring Framework Instrumentation details](#) for a complete listing.

#### Spring Core

Instrumented Spring Framework provides visibility into the performance and resource utilization of the Spring container. For example, each `ApplicationContext` exposes execution time metrics for common operations such as `refresh` and `getBean` and provides the names of its bean definitions. Each `BeanFactory` provides the number of prototype and singleton beans created and the average execution time of the bean creations. Instrumented Spring Framework also provides runtime control of the Spring container. For example, `ApplicationContexts` allow runtime invocation of their `refresh` and `close` methods.

Components marked as `@Component` (stereotype indicating a generic Spring-managed component) or `@Service` (stereotype indicating a business service) are automatically exposed to the management model as well. As such, execution time of all the components' public methods will be monitored. This allows for analysis of performance on a per use case basis.

#### Spring DAO

Instrumented Spring Framework provides summary visibility into JDBC operations and transaction management. Spring DAO components expose metrics such as transaction commit and rollback rates, transaction throughput, and JDBC query execution throughput. Additionally, components marked as `@Repository` or `@Transactional` are exposed to the management model as services. As such, execution time of all the components' public methods will be monitored.

#### Spring ORM

The object-relational mapping integration in Instrumented Spring Framework exposes ORM statistics in a manner consistent with the rest of the application model. Currently, Hibernate and JPA are the only ORMs whose metrics are exposed to the Instrumented Spring Framework model. A number of Hibernate `SessionFactory` metrics are available, including query execution count and entity fetch count. JPA metrics are currently only related to transaction management. The `JpaTransactionManager` exposes transaction commit, suspend, resume, and rollback rates.

## Spring JEE Integration

Instrumented Spring Framework provides visibility into Spring JMS, Java Mail, and thread pooling integration components.

Instrumented Spring JMS provides data for both JMS message senders and receivers. The `MessageListenerContainers` track average execution time of message receipt and provide the number of failed message receipts. They expose the number of concurrent and/or scheduled consumer threads and allow for modification of the number of consumers, to help alleviate JMS queue backup at runtime. The `JmsTemplate` provides average time taken to send or receive messages and number of failed message sends or receipts. It allows runtime modification of the receive timeout.

Instrumented Spring `JavaMailSender` provides statistics on emails sent and allows you to configure the connection properties of the underlying mail server at runtime.

Instrumented Spring `ThreadPoolTaskExecutor` exposes thread pool utilization statistics and allows for dynamic resizing of the pool and underlying queue at runtime.

## Spring MVC

Instrumented Spring MVC provides HTTP request statistics (via `DispatcherServlet`), view resolution and view rendering statistics, and Controller performance metrics. Most Spring MVC Controller implementations expose request statistics, and components annotated with `@Controller` will have execution time metrics exposed for each of their public methods. `ViewResolvers` and `ViewRenderers` provide throughput and failure data. `DispatcherServlet` exposes throughput of HTTP requests and also exposes the number of unhandled Exceptions, allowing you to track how often a stack trace is being displayed to your application's end user.

## 2.2. Spring Framework Instrumentation details

Table 2.1. Spring Core

Spring Framework Class	JMX MBean Type Key Property	Metrics	Attributes	Operations
Abstract Application Context	Application Context	GetBean-AverageExecutionTime(ms) GetBean-Executions GetBean-FailedExecutions GetBeanNamesForType-AverageExecutionTime(ms) GetBeanNamesForType-Executions GetBeanNamesForType-FailedExecutions	Active	Close Refresh

Spring Framework Class	JMX MBean Type Key Property	Metrics	Attributes	Operations
		GetBeansOfType-AverageExecutionTime(ms) GetBeansOfType-Executions GetBeansOfType-FailedExecutions Refresh-AverageExecutionTime(ms)		
Configurable Listable Bean Factory	Bean Factory	AverageElapsedTimePerPrototypeBeanCreation(ms) AverageElapsedTimePerSingletonBeanCreation(ms) PrototypeBeansCreated SingletonBeansCreated		
@Component	Component, subtype= Method	AverageElapsedExecutionTime(ms) ExecutionsPerSecond InvocationCount MaximumExecutionTime(ms) MinimumExecutionTime(ms) ThrownExceptionCount		
@Service	Service, subtype= Method	AverageElapsedExecutionTime(ms) ExecutionsPerSecond InvocationCount MaximumExecutionTime(ms) MinimumExecutionTime(ms) ThrownExceptionCount		

Table 2.2. Spring DAO

Spring Framework Class	JMX MBean Type Key Property	Metrics	Attributes	Operations
Jdbc Template	Jdbc Template	AverageExecutionTime(ms) ExecutionsPerSecond FailedExecutions		
Abstract Platform Transaction	Platform Transaction Manager	CommitsPerSecond FailedCommits FailedResumes		

Spring Framework Class	JMX MBean Type Key Property	Metrics	Attributes	Operations
Manager		FailedRollbacks FailedSuspends ResumesPerSecond RollbacksPerSecond SuspendsPerSecond		
Transaction Template	Transaction Template	AverageExecutionTime(ms) ExecutionsPerSecond FailedExecutions		
@Transactional	Transactional, subtype= Method	AverageElapsedExecutionTime(ms) ExecutionsPerSecond InvocationCount MaximumExecutionTime(ms) MinimumExecutionTime(ms) ThrownExceptionCount		
@Repository	Repository, subtype= Method	AverageElapsedExecutionTime(ms) ExecutionsPerSecond InvocationCount MaximumExecutionTime(ms) MinimumExecutionTime(ms) ThrownExceptionCount		

Table 2.3. Spring ORM

Spring Framework Class	JMX MBean Type Key Property	Metrics	Attributes	Operations
org.hibernate.Session Factory	Hibernate Session Factory	EntityInsertCount EntityInsertCount1m QueryExecutionMaxTime(ms) EntityUpdateCount EntityUpdateCount1m CollectionUpdateCount CollectionUpdateCount1m		

Spring Framework Class	JMX MBean Type Key Property	Metrics	Attributes	Operations
		EntityLoadCount EntityLoadCount1m EntityFetchCount EntityFetchCount1m EntityDeleteCount EntityDeleteCount1m CollectionRecreateCount CollectionRecreateCount1m QueryExecutionCount QueryExecutionCount1m FlushCount FlushCount1m CollectionLoadCount CollectionLoadCount1m SuccessfulTransactionCount SuccessfulTransactionCount1m QueryCacheHitCount QueryCacheHitCount1m CollectionRemoveCount CollectionRemoveCount1m ConnectCount ConnectCount1m StartTime SecondLevelCachePutCount SecondLevelCachePutCount1m QueryCachePutCount QueryCachePutCount1m SessionOpenCount SessionOpenCount1m TransactionCount TransactionCount1m CollectionFetchCount CollectionFetchCount1m SessionCloseCount SessionCloseCount1m QueryCacheMissCount QueryCacheMissCount1m SecondLevelCacheMissCount SecondLevelCacheMissCount		

Table 2.4. Spring JEE Integration

Spring Framework Class	JMX MBean Type Key Property	Metrics	Attributes	Operations
Default Message Listener Container	Default Message Listener Container	ActiveConsumers AverageElapsedTimePerMessage(ms) FailedMessages MessagesPerSecond MessagesReceived ScheduledConsumers	Max Concurrent Consumers, Concurrent Consumers, Max Messages Per Task, Idle Task Execution Limit, Running	start, stop, set Concurrent Consumers, set Max Concurrent Consumers, scale Max Concurrent Consumers set Max Messages Per Task, set Idle Task Execution Limit
Server Session Message Listener Container	Server Session Message Listener Container	AverageElapsedTimePerMessage(ms) FailedMessages MessagesPerSecond MessagesReceived	Max Messages Per Task, Running	start, stop, set Max Messages Per Task
Simple Message Listener Container	Simple Message Listener Container	AverageElapsedTimePerMessage(ms) FailedMessages MessagesPerSecond MessagesReceived	Concurrent Consumers, Running	start, stop, set Concurrent Consumers
Jms Template	Jms Template	AverageElapsedTimePerMessageSent(ms) FailedMessageSends MessagesSent MessagesSentPerSecond	Receive Timeout	set Receive Timeout
Java Mail Sender Impl	Java Mail Sender	AverageElapsedTimePerMessage(ms) FailedMessages	Host Port	setHost setPort

Spring Framework Class	JMX MBean Type Key Property	Metrics	Attributes	Operations
		MessagesSent MessagesPerSecond		
Thread Pool Task Executor	Thread Pool Task Executor	ActiveTasks LargestPoolSize PoolSize QueueSize	Core Pool Size, Max Pool Size, Keep Alive Seconds	set Core Pool Size, set Max Pool Size, set Keep Alive Seconds

Table 2.5. Spring MVC

Spring Framework Class	JMX MBean Type Key Property	Metrics	Attributes	Operations
Controller (Interface)	Controller	FailedRequests RequestsHandled RequestsHandledPerSecond		
@Controller	Multi Action Controller, subtype=Method	AverageElapsedExecutionTime(ms) ExecutionsPerSecond InvocationCount MaximumExecutionTime(ms) MinimumExecutionTime(ms) ThrownExceptionCount		
Dispatcher Servlet	Dispatcher Servlet	AverageElapsedTimePerRequest(ms) MaxRequestTime(ms) MinRequestTime(ms) RequestsProcessed RequestsPerSecond UnhandledExceptions		
Abstract View	View	AverageElapsedTimePerViewRender(ms) FailedViewRenders		

---

<b>Spring Framework Class</b>	<b>JMX MBean Type Key Property</b>	<b>Metrics</b>	<b>Attributes</b>	<b>Operations</b>
		ViewsRendered ViewsRenderedPerSecond		
View Resolver	View Resolver	AverageElapsedTimePerView(ms) FailedViewResolutions ViewsResolved ViewsPerSecond		

## 3. Instrumenting Applications

### 3.1. Introduction

This chapter will show you how to setup your application to use Instrumented Spring Framework. This chapter outlines the build and configuration steps necessary to allow Instrumented Spring Framework to automatically export its components to your application's JMX MBeanServer and enable automatic monitoring of those components.

Follow the steps below to begin exposing management data from your standalone or web application:

- Add Instrumented Spring Framework jars to your application
- Configure your application for auto-discovery
- Configure your application for monitoring (optional)
- Configure your application for remote JMX access
- Deploy your application

### 3.2. Adding Instrumented Spring Framework jars to applications

All of the jars that must be added to your application will be in the "dist" or "lib" directories of the unpacked spring-framework-instrumented-management zip file.

As explained in [Components of Instrumented Spring Framework](#), the modules of Instrumented Spring Framework are separated into three categories: Spring Framework jars woven with Instrumentation, Spring Framework Instrumentation, and Management API. The jars you need to use will vary depending on the application you are instrumenting.

To start, replace any Spring Framework jars you are using in your application with the instrumented jars in the dist directory. These jars are named `org.springframework.<module>.instrumented-<version>.jar`.

For each instrumented jar you add to your application, be sure to include the matching `com.springsource.management.instrumentation.springframework.<module name>-<version>.jar`. These instrumentation jars contain the aspects and Java classes used to manage and monitor the components in the "instrumented" jar.

Always include `com.springsource.management.instrumentation.springframework.applicationcontext.jar`

It must be present to enable auto-discovery.

Some instrumentation jars are shared among the instrumented Spring modules. For example, there are subclasses of `AbstractPlatformTransactionManager` in multiple jars. As such, all of these jars may have a dependency on the `com.springsource.management.instrumentation.springframework.transaction.manager.jar`. When in doubt, consult the `Import-Packages` statement of the instrumented jar Manifest for the required dependencies.

Also include the following Management API jars:

- `com.springsource.management.adapter.jmx-<version>.jar`
- `com.springsource.management.agent.bootstrap-<version>.jar`
- `com.springsource.management.agent.config-<version>.jar`
- `com.springsource.management.agent.control-<version>.jar`
- `com.springsource.management.agent.discovery.domain-<version>.jar`
- `com.springsource.management.agent.discovery.resource-<version>.jar`
- `com.springsource.management.agent.inventory-<version>.jar`
- `com.springsource.management.agent.monitoring-<version>.jar`

You also need to include one of the following jars, depending on whether you are instrumenting a standalone application or a web application:

- `com.springsource.management.agent.discovery.application.standalone-<version>.jar` OR
- `com.springsource.management.agent.discovery.application.web-<version>.jar`

We will be providing support for downloading all of the above jars for use with Ant/Ivy and Maven in the near future.

### 3.3. Configuring applications for auto-discovery

Instrumented Spring Framework automatically discovers applications by advising the refresh method of the `AbstractApplicationContext`. Similarly, an application's managed resources are undeployed when all of its `AbstractApplicationContexts` have been closed.

#### Configuring web applications for auto-discovery

If a `WebApplicationContext` is detected, the `ServletContext` will be obtained from it. The discovered application name will then be set to the value of

`ServletContext.getServletContextName()`, if the servlet context name is set through the `display-name` attribute of the `web.xml`. If the context name is not set, the discovered application name will be set to the value of `ServletContext.getRealPath("/")`.

NOTE: WebLogic does not return a value from `ServletContext.getRealPath()`. If you are using WebLogic, please set the `display-name` attribute in your application's `web.xml` file to a logical name for your application. This MUST be done in order for the auto-discovery component to export any managed resources.

Application discovery using `ServletContext.getRealPath()` has been tested on SpringSource tc Server 6.0, SpringSource dm Server 1.0, Apache Tomcat 6.0, WebSphere 6.1, and JBoss 4.2 and should work without issue. If you do not see any managed resources being exported (see [Deploying applications](#) for details on how to troubleshoot auto-discovery), try setting the `display-name` attribute and redeploying the application.

## Configuring standalone applications for auto-discovery

If any other subclass of `AbstractApplicationContext` is detected, the application is assumed to be a standalone application. If you are running a standalone application, you will need to set the System property `"spring.managed.application.name=<application name>"`. If this property is not set, managed resources will not be exported.

## Configuring applications for managed resource discovery

Any bean that is created through an `AbstractApplicationContext` will be discovered as a managed resource if the instrumented components know how to manage and monitor it. When managed resources are exported, they are named using bean names, therefore it is important to make sure that all beans you wish to manage are consistently named. An instance of an anonymous bean could have a different name each time the application is restarted, which will cause inconsistencies in data persisted by external management systems such as SpringSource AMS.

A managed resource representing an `ApplicationContext` or `BeanFactory` is named using the display name of the `ApplicationContext`. This is set consistently by Spring within web applications. However, if you are creating your own `ApplicationContext` (such as a new `ClasspathXmlApplicationContext`), it is recommended that you set `"refresh"` to `false`, then set the display name on the `ApplicationContext` before calling `"refresh"`, as in the example below:

```
ClasspathXmlApplicationContext applicationContext = new ClasspathXmlApplicationContext(new
    String[] { "classpath:com/springsource/management/test-context.xml" }, false);
applicationContext.setDisplayName("My Application's ApplicationContext");
applicationContext.refresh();
```

This will ensure that your `ApplicationContext` and its `BeanFactory` are named the same each time the application is restarted.

### 3.4. Configuring applications for monitoring (optional)

Configure instrumented Spring applications by adding a properties file called "management.config" to a directory in your application's classpath (for example, WEB-INF/classes/management.config in a web application). Properties defined in this file will be read and applied by Instrumented Spring Framework when the instrumented application is deployed.

Table 3.1. Configuration Properties

Property	Description
monitor.proxyTargetClass	This property specifies the Spring AOP proxy behavior used when creating monitoring proxies for user-defined components, such as those annotated with @Service, @Component, etc. Setting the value to true will force the use of CGLIB to proxy target classes for every component to which monitoring proxies are applied. Default value is false, which means that JDK proxies will be created for any component implementing an interface. CGLIB will only be used for components that do not implement interfaces.
jmx.discoverWeblogicServer	Determines if the MBeanServer auto-detector should attempt to detect a WebLogic environment through classpath scanning. If a WebLogic environment is not detected, this component will still attempt local MBean server discovery (if discoverLocalServer is set to true) and/or local MBeanServer creation. Default value is true. Change this value if a classpath scan reveals a WebLogic environment, but you do not wish to retrieve the WebLogic MBeanServer through JNDI lookup.
jmx.discoverWebSphereServer	Determines if the MBeanServer auto-detector should attempt to detect a WebSphere environment through classpath scanning. If a WebSphere environment is not detected, this component will still attempt local MBean server discovery (if discoverLocalServer is set to true) and/or local MBeanServer creation. Default value is true. Change this value if a classpath scan reveals a WebSphere environment, but you do not wish to retrieve the WebSphere MBeanServer through AdminClient lookup.
jmx.discoverLocalServer	Determines if MBeanServerFactory.findMBeanServer() should be used to locate an existing MBeanServer if a WebLogic or WebSphere MBeanServer is not found, or if discovery of WebLogic and WebSphere servers has been disabled. Default value is true.
jmx.agentId	The agent id of the MBeanServer to locate. Default is none. If specified, this will result in an automatic attempt being made to locate the attendant

Property	Description
	MBeanServer, and (importantly) if said MBeanServer cannot be located no attempt will be made to create a new MBeanServer
jmx.defaultDomain	The default domain to be used by the MBeanServer, to be passed to MBeanServerFactory.createMBeanServer() or MBeanServerFactory.findMBeanServer(). Default is none.

## 3.5. Configuring applications for remote JMX access

Instrumented Spring Framework automatically creates JMX ModelMBeans representing each application resource it discovers, and registers these ModelMBeans in an auto-detected MBeanServer. If running your application in SpringSource dm Server, SpringSource tc Server, JBoss, WebLogic or WebSphere, Instrumented Spring Framework should be able to automatically detect the container MBeanServer and export the managed resources, with no additional configuration required.

### JMX in SpringSource dm Server

Remote monitoring via JMX is automatically enabled in SpringSource dm Server versions 1.0.1 or higher. Consult the SpringSource dm Server documentation for additional information.

### JMX in SpringSource tc Server

Remote monitoring via JMX is automatically enabled in SpringSource tc Server. Consult the SpringSource tc Server documentation for additional information.

### JMX in WebLogic

If deploying your application in WebLogic 9.1+, Instrumented Spring Framework will automatically export all managed resources to the WebLogic MBeanServer obtained through a JNDI lookup of "java:comp/env/jmx/runtime". See [Configuration Properties](#) for options related to WebLogic MBeanServer discovery.

### JMX in WebSphere

If deploying your application in WebSphere 5.1+, Instrumented Spring Framework will automatically export all managed resources to the WebSphere MBeanServer obtained through WebSphere's proprietary AdminServiceFactory API. See [Configuration Properties](#) for options related to WebSphere MBeanServer discovery.

## JMX in Apache Tomcat

Tomcat will need to be configured to create an MBeanServer and a JSR-160 connector. See [http://tomcat.apache.org/tomcat-\\${product.version}-doc/monitoring.html](http://tomcat.apache.org/tomcat-${product.version}-doc/monitoring.html) for further information. The quick start method recommended by Tomcat is as follows:

Add the following properties to \$CATALINA\_HOME/bin/setenv.sh or setenv.bat:

```
CATALINA_OPTS="-Dcom.sun.management.jmxremote \  
-Dcom.sun.management.jmxremote.port=6969 \  
-Dcom.sun.management.jmxremote.ssl=false \  
-Dcom.sun.management.jmxremote.authenticate=false"
```

A full list of options for configuring JMX remote through these System properties can be found on [Sun's website](#).

## JMX in JBoss

If deploying your application in JBoss 3.2+, Instrumented Spring Framework will automatically export all managed resources to the JBoss MBeanServer obtained through lookup of the local MBeanServer. The exported MBeans are also visible from the JBoss JMX Console.

## JMX in standalone applications

For standalone applications (or possibly other containers not listed above), Instrumented Spring Framework will simply export managed resources to any local MBeanServer it finds. For remote access to this MBeanServer, a JSR-160 connector should be created. Spring JMX can be used to create a local MBeanServer and JSR-160 connector in your application. Consult the Spring Framework reference manual for further information on Spring JMX.

Additionally, you can use the JMX remote system properties to create these. The following system properties would create a local MBeanServer and remote connector:

```
-Dcom.sun.management.jmxremote  
-Dcom.sun.management.jmxremote.port=6969  
-Dcom.sun.management.jmxremote.ssl=false  
-Dcom.sun.management.jmxremote.authenticate=false"
```

A full list of options for configuring JMX remote through these System properties can be found on [Sun's website](#).

## 3.6. Deploying applications

Now that the configuration is complete, you are ready to deploy your application and let Instrumented Spring Framework begin automatically monitoring your application components.

NOTE for WebSphere users: WebSphere ships an older version of aspectjrt.jar in their lib directory than was used to weave the instrumented Spring Framework jars. As a result, Instrumented Spring Framework will not work correctly unless the application class loader order is changed to load classes from the application's classpath first. The application classloader order can be changed from the WebSphere Administrative Console by navigating to Enterprise Applications->\${application name}->Manage Modules->Class Loader Order

### Verifying auto-discovery through logging

Instrumented Spring Framework uses Commons Logging. All logger names begin with "com.springsource.management". The log file can be used to verify that your application was discovered on deployment. If the logging level is set to "Info", you should see a message in your log when an application is discovered. For example:

```
INFO [com.springsource.management.agent.discovery.application.DefaultApplicationDiscoverer] -  
    Discovered application: Spring PetClinic
```

You can also see each managed resource that is discovered and exported. For example:

```
INFO [com.springsource.management.adapter.jmx.ManagedResourceExporter] - Registered MBean:  
    spring.application:application=Spring PetClinic,type=PlatformTransactionManager,  
    name=transactionManager
```

### Verifying auto-discovery through JConsole

If your container supports it, a tool such as JConsole can be used to verify that Instrumented Spring Framework has automatically discovered and exported your application resources. For example, the following is displayed in JConsole upon deploying the Spring Framework Pet Clinic sample application to Tomcat:



## 4. Managing Custom Components

### 4.1. Introduction

This chapter will show you how to configure any of your custom application components for inclusion in the Instrumented Spring Framework management model. This chapter will demonstrate how to use simple annotations to automatically export your components to JMX as ModelMBeans. These generated ModelMBeans will contain descriptor information that enables automatic discovery by [SpringSource AMS](#). This information can also be used by other JMX clients to more easily visualize or import your application model.

### 4.2. Exporting components to JMX

The recommended mechanism for exporting JMX MBeans is the Spring JMX annotations. Consult the JMX chapter of the Spring Framework reference manual for additional information on Spring JMX. Instrumented Spring Framework provides an additional annotation, `@ManagedMetric`, that should be added to getters to denote that the return value of the getter is a metric. In order to make use of the new `ManagedMetric` annotation, use `com.springsource.management.adapter.jmx.MetadataAMBeanInfoAssembler` when defining your `MBeanExporter`, as demonstrated below.

NOTE: The `ManagedMetric` annotation, the enums used by its fields, and the `MetadataAMBeanInfoAssembler` logic will likely be moved from Instrumented Spring Framework to open source Spring Framework in the future.

```
<beans>
  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="assembler" ref="assembler"/>
    <property name="namingStrategy" ref="namingStrategy"/>
    <property name="autodetect" value="true"/>
  </bean>

  <bean id="jmxAttributeSource"
        class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>

  <!-- will create management interface using annotation metadata -->
  <bean id="assembler"
        class="com.springsource.management.adapter.jmx.MetadataAMBeanInfoAssembler">
    <property name="attributeSource" ref="jmxAttributeSource"/>
  </bean>

  <!-- will pick up the ObjectName from the annotation -->
  <bean id="namingStrategy"
        class="org.springframework.jmx.export.naming.MetadataNamingStrategy">
    <property name="attributeSource" ref="jmxAttributeSource"/>
  </bean>
</beans>
```

## Source-level metadata types

The following source level metadata types are available for use in Spring JMX and Instrumented Spring Framework.

Table 4.1. Source-level metadata types

Purpose	Annotation	Annotation Type
Mark all instances of a Class as JMX managed resources	@ManagedResource	Class
Mark a method as a JMX operation	@ManagedOperation	Method
Mark a getter or setter as one half of a JMX attribute	@ManagedAttribute	Method (only getters and setters)
Mark a getter as a JMX attribute with JMX metricType and units descriptor values	@ManagedMetric	Method (only getters)

The following configuration parameters are available for use on these source-level metadata types. They are translated to ModelMBeanInfo fields by the MetadataMBeanInfoAssembler.

Table 4.2. Configuration Parameters

Parameter	Description	Applies To	ModelMBean Field Translated To
objectName	Used by MetadataNamingStrategy to determine the ObjectName of a managed resource	@ManagedResource	ModelMBean ObjectName
description	Sets the friendly description of the resource, attribute, metric or operation	ManagedResource, ManagedAttribute, ManagedOperation, ManagedMetric	ModelMBeanInfo.getDescription(), ModelMBeanAttributeInfo.getDescription(), ModelMBeanOperationInfo.getDescription()

Parameter	Description	Applies To	ModelMBean Field Translated To
category	Sets the category of the metric	ManagedMetric	ModelMBeanAttributeInfo. getDescriptor(). getField("metricCategory")
displayName	Sets the display name of the metric for possible use by external clients	ManagedMetric	ModelMBeanAttributeInfo. getDescriptor(). getField("displayName")
indicator	Indicates that this metric provides relevant insight into the state of the system and should be considered a possible indicator of a system issue	ManagedMetric	ModelMBeanAttributeInfo. getDescriptor(). getField("indicator")
metricType	A description of how the metric's values change over time	ManagedMetric	ModelMBeanAttributeInfo. getDescriptor(). getField("metricType")
unit	The unit in which an attribute is measured, for example "B" or "ms".	ManagedMetric	ModelMBeanAttributeInfo. getDescriptor(). getField("units")

## Example: Exporting a custom component

The following example uses annotations to automatically export JMX ModelMBeans:

```

package org.springframework.webflow.samples.booking.messaging;

import java.util.ArrayList;
import java.util.List;

import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.ObjectMessage;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.webflow.samples.booking.Booking;
import org.springframework.webflow.samples.booking.mail.BookingMailService;

```

```
@ManagedResource(objectName = "spring.application:application=swf-booking-mvc,  
    type=MessageListener,name=bookingQueueMessageListener")  
public class BookingQueueMessageListener implements MessageListener {  
  
    private List messageQueue = new ArrayList();  
  
    private long messageQueueSize=0;  
  
    private boolean sendConfirmationEmail = false;  
  
    @ManagedMetric(category=MetricCategory.UTILIZATION, displayName="Message Queue Size",  
        indicator=true, description="The size of the Message Queue",  
        metricType = MetricType.COUNTER, unit="none")  
    public long getMessageQueueSize() {  
        return messageQueueSize;  
    }  
  
    @ManagedAttribute(description="Send a confirmation email")  
    public boolean isSendConfirmationEmail() {  
        return sendConfirmationEmail;  
    }  
  
    @ManagedOperation  
    public void resetMessageQueue() {  
        this.messageQueue.clear();  
    }  
  
    @ManagedAttribute(description = "Send a confirmation email")  
    public void setSendConfirmationEmail(boolean sendConfirmationEmail) {  
        this.sendConfirmationEmail = sendConfirmationEmail;  
    }  
}
```

## 5. Integrating with SpringSource AMS

SpringSource Enterprise includes SpringSource Application Management Suite (AMS), a comprehensive enterprise application management solution that is designed to manage and monitor all of your Spring-powered applications, the Spring runtime, and a variety of platforms and application servers.

SpringSource AMS can be used in all stages of product lifecycle, but it is most often used in production by System Administration and Operations teams. By integrating the detailed information provided by Instrumented Spring Framework with high level OS and application server metrics, AMS assists in diagnosing and preventing application performance issues, bridging the gap between your Operations and Development teams. Please see the [AMS Product Page](#) for an overview of AMS features.

Instrumented Spring Framework exports ModelMBeans to JMX with specific descriptor metadata that can be read and interpreted by AMS. This allows AMS to automatically discover, manage, and monitor your instrumented application without requiring additional configuration. This includes any custom components exported to JMX as described in [Managing Custom Components](#). Please consult the [AMS Documentation](#) for details on how SpringSource AMS imports ModelMBean metadata for automatic inclusion in the system inventory model.

## 6. Using Instrumented Spring Framework in SpringSource dm Server

### 6.1. Using Instrumented Spring Framework in dm Server 1.0

SpringSource dm Server Enterprise Edition 1.0 includes Instrumented Spring Framework and an associated Instrumented Spring library in its local repository. This allows you to use Instrumented Spring Framework in your OSGi application exactly as you would Spring Framework. Instrumented Spring Framework exports all packages using the underlying version number of Spring Framework. This allows you to easily switch to using Instrumented Spring Framework without changing any of your application dependencies. Please consult the dm Server reference manuals for additional information on building OSGi applications.

SpringSource dm Server Enterprise Edition publishes bundle and application start/stop events to Instrumented Spring Framework. Thus, your application or bundle MBeans will still be segmented by the proper "application" ObjectName key property, just as they are for standalone and web applications.

### 6.2. Using Instrumented Spring Framework in dm Server 2.0

SpringSource dm Server 2.0 is currently under development. Instrumented Spring Framework cannot yet be used properly in dm Server 2.0 milestone releases. Stay tuned for details on using Instrumented Spring Framework in dm Server 2.0.