JMX Support in Spring Integration

Java Management Extensions are a standard way of exposing runtime information about a running Java application to allow the application's monitoring and runtime management. Attributes exposed via JMX can either be read only, such as the number of messages on a queue or read write such as the maximum number of database connections allowed in a connection pool. In addition to exposing attributes, JMX allows operations to be exposed, this relates directly to a Java method invocation with zero or more parameters. In addition to operations and attributes, JMX has a concept of notifications which are essentially events emitted by a component. These are generally used to notify listeners of some sort of problem rather than a more general form of inter-process or inter-component communication. One of the advantages of using JMX is the tooling support with many monitoring systems supporting JMX allowing operations teams to use the tooling they may already use to monitor hardware to also monitor Java applications that expose information via JMX. Where JMX is not directly supported, it is also possible to map some of the JMX concepts to other monitoring technologies for example JMX notifications can be mapped to Simple Network Monitoring Protocol (SNMP) traps.

JMX is supported in Spring Integration in two ways. First, we'll look at the out-of-the-box support for exposing information about message channels, message sources, and message handlers. We'll then look at Spring Integration's support for adapting JMX concepts to Spring Integration messages and vice versa.

Monitoring channels and endpoints with JMX

The out-of-the-box JMX support means that, by simply adding the configuration below, a wide range of information about the runtime behavior of your application will be made available. The channel's implementation will result in a default set of information for all channels, additional information for channels that are pollable, and yet more information for channels that are both pollable and backed by a queue. The type of monitoring applied is determined by checking which of the interfaces `MessageChannel`, `PollableChannel`, and `QueueChannel` are implemented by beans during the application's initialization phase.

```
<jmx:mbean-export default-domain="quote" />
```

The default message channel implementation exposes the attributes shown in Table 1. Where an attribute relating to rate or ratio over time is exposed, this is calculated using an exponential moving average. This allows for the calculation of rates and ratios, which give a higher weight to more recent data items while at the same time avoiding the need to constantly recalculate or to maintain a long list of data points. In applications requiring very high performance or throughput it is worth considering that there will be some effect from observing this data over time and, while the calculations are optimized, there is some computational overhead and some need for synchronization of threads. Where the return type in Table 1 is shown as `org.springframework.integration.monitor.Statistics`, a call to get the attribute will return an instance of this class containing count, mean, max, min, and the standard deviation of the attribute.

Table 1 Default set of metrics exposed for message channels

Send count	`Int`	Number of messages sent to this channel. Includes unsuccessful message sends that result in errors. Resending the same message a number of times will increment this value repeatedly even where it is not successfully processed.
Send rate	`Statistics`	Provides statistics relating to the rate of messages sent on this channel. Includes a mean number of messages per second using the previously mentioned exponential moving average approach. As for the send count, retrying failed message sends will result in multiple entries for the purposes of this statistic.
Time since last send	`Double`	Time in seconds since last successful or unsuccessful send.
Mean send rate	`Double`	Calculated mean number of messages sent per second.
Send duration	`Statistics`	Statistics related to successful send durations in seconds.
Min send duration	`Double`	Minimum recorded time for a successful send in milliseconds.
Max send duration	`Double`	Maximum recorded time for a successful send in milliseconds.
Mean send duration	`Double`	Mean milliseconds per successful send.
Standard deviation of send duration	`Double`	Standard deviation of measured milliseconds per successful send.
Send error count	`Int`	Number of sends that resulted in an error.
Mean error rate	`double`	Calculated mean per second of sent messages that resulted in an error.
Error rate	`Statistics`	Statistics relating to message sends resulting in an error.
Mean error ratio	`Double`	Mean per second ratio of messages causing an error, where 1 indicates no successful sends and 0 indicates no errors.

The above attributes relate only to sending since the `MessageChannel` interface does not define any methods related to the receiving messages. Channel implementations that cater to asynchronous receive operations where the send does not result in a direct pass through to the receiver will additionally implement the interface. Where this interface is `PollableChannel`, detected additional details related to receive operations will be exposed, as detailed below.

Table 2 Additional metrics exposed for pollable channels

Receive count	`Int`	Calls to receive that returned a non-null result and did not result in an error.
Receive error count	`Int`	Calls to receive that resulted in an error.

Where a channel is backed by a queue, it is often useful to know how many messages are currently queued and what the remaining capacity of the queue is. `QueueChannel` implementing classes additionally expose the metrics shown in Table 3.

Table 3 Additional metrics exposed for channels backed by a queue

Queue size	`Int`	Current number of messages queued and waiting to be received
Queue remaining capacity	`Int`	Message that can be queued before queue is full

Monitoring channels is useful for checking throughput and error rates; however, it is also useful to be able to monitor Spring Integration components such as routers, transformers, and adapters, which act as message sources and handlers.

Components that act as handlers implement the `MessageHandler` interface in some form. It may not be obvious when writing a POJO router that it will effectively implement this interface; however, there will always be either an adapter or a super class that implements this interface somewhere at runtime. Table 4 details the MessageHandler metrics exposed for monitoring purposes.

Table 4 Metrics exposed for message handlers

Handle execution count	`int`	The number of calls that have been made to the `handleMessage(Message<?> message)` method.
Handler error count	`int`	The number of calls to the handle message method that resulted in an error.
Error count	`int`	Message that can be queued before queue is full.
Handle duration	`Statistics`	Statistics relating to the duration in milliseconds of calls to the handle message method.
Mean duration	`double`	Mean duration of handle calls in milliseconds.
Min duration	`double`	Min duration of handle calls in milliseconds.
Max duration	`double`	Max duration of handle calls in milliseconds.
Standard deviation of duration	`double`	Standard deviation for duration of handle calls.
Active count duration	`int`	The number of calls to handle message that are currently in process.

In addition to channel and handlers, the built-in support also exposes message sources. These will typically be pollable sources of messages such as inbound JMS channel adapter where a scheduled task periodically calls receive on the message source with any non null result then being published to a channel. In the case of the message sources, a simple count of messages received in response to calls to receive will be maintained and exposed as a JMX attribute.

In addition to exposing attributes mbean export in Spring Integration also exposes a number of operations. All of the exposed components expose a reset operation that allows all metrics such as counts and rates to be reset to zero. In addition, components that are active in that they have some scheduled periodic behavior such as polling a channel generally implement the Spring `Lifecycle` interface, which defines stop, start, and `isRunning` methods. These methods allow active components to be stopped individually without the need to stop the whole application or application context. For convenience, `MessageHandlers` and `MessageSource` instances that implement the `Lifecycle` interface have these methods exposed as JMX operations.

Integration using JMX adapters

In this section we will look at the support offered by Spring Integration for JMX notifications, operations and attributes in turn. The JMX support offered by Spring Integration covers both inbound and outbound adapters. Outbound adapters are primarily there to make it easy to manage and monitor Spring Integration applications. The inbound adapters allow the use of Spring Integration to carry out the management and monitoring of an application via JMX. It is also common to use Spring Integration both for the core application functionality and the management and monitoring of the core application.

One of the most common monitoring requirements for an application is that it should produce notifications when things go wrong so problems can be addressed early rather than waiting until a small problem that no one has noticed becomes a big show stopper of a problem. The best way to achieve this using JMX is through notifications, which allow an application to send notification of a problem to subscribed listeners. The concept of a notification maps well to a message and hence the support for JMX notifications takes the form of channel adapters, which map between notifications and Spring Integration messages.

The inbound channel adapter is very simple to setup and requires a channel name and a JMX object name. The below configuration assumes that there is a Spring bean named mBeanServer available.

```
<jmx:notification-listening-channel-adapter id="adapter"
                   channel="channel"
                   object-name="example.domain:name=publisher" />
```

Where not all notifications will be of interest to an application, an instance of `javax.management.NotificationFilter` can be provided to the inbound channel adapter. The example below also demonstrates use of an MBean server bean with a nonstandard name.

```
<jmx:notification-listening-channel-adapter id="adapter"
                  channel="channel"
                  mbean-server="someServer"
                  object-name="example.domain:name=somePublisher"
                  notification-filter="notificationFilter" />
```

The outbound equivalent is also very straightforward. The example below shows how to configure a notification-publishing channel adapter that publishes Spring Integration messages as JMX notifications.

```
<context:mbean:export/>

<jmx:notification-publishing-channel-adapter id="adapter"
                              channel="channel"
                              object-name="example.domain:name=publisher" />
```

When it comes to JMX operations, Spring Integration provides support for operation invocations, where no result is expected through a channel adapter implementation, and support for operation invocations, where a response is of interest through a gateway implementation. In both cases, the invocation of the JMX operation is triggered by the receipt of a Spring Integration Message, which is then used to determine the parameters to pass the operation, if any are required.

Both the gateway and the channel implementations use the same strategy for mapping the inbound trigger message to operation invocation parameters. Where the payload of the message is a `Map,` the payload is assumed to be a set of key value pairs. Where a single parameter is expected, the payload itself will be assumed to be the parameter. Where the JMX invocation does not expect a parameter the payload will be ignored in all cases.

Given that the adapter carries out the parameter mapping, all that is required is the JMX object name, the operation name, the request channel name, and the name of the MBean server if it is not the default of mbeanServer.

```
<jmx:operation-invoking-channel-adapter id="adapter" channel="requests"
                object-name="example.domain:name=TestBean"
                operation-name="ping" mbean-server="myMbeanServerRef" />
```

The operation-invoking gateway looks almost exactly the same except it allows for the configuration of a reply channel.

```
<jmx:operation-invoking-outbound-gateway request-channel="requests"
  reply-channel="replyChannel"
  object-name="org...jmx.config:type=TestBean,name=testBeanGateway"
  operation-name="methodWithReturn" mbean-server="myMbeanServerRef" />
```

The final JMX adapter allows for the case where it is desirable to periodically poll an attribute exposed over JMX. The attribute value then becomes the payload of the resulting message.

```
<jmx:attribute-polling-channel-adapter id="adapter"
                channel="channel"
                object-name="example.domain:name=someService"
                attribute-name="requestCount">
     <si:poller max-messages-per-poll="1" fixed-rate="500"/>
 </jmx:attribute-polling-channel-adapter>
```

Summary

We explored the ways you can take advantage of Java Management Extensions (JMX) within your message flows. You learned how to monitor certain attributes of the message channels as well as the message endpoints within the application context.

Last updated: October 19, 2011