**CachingConnectionFactory**

*What every Java engineer should know about microservices:*[*Reactive Microservices Architecture*](https://dzone.com/go?i=153025&u=https%3A%2F%2Finfo.lightbend.com%2FCOLL-20XX-Reactive-Microservices-Architecture-RES-LP.html%3Flst%3DDZ%3Futm_source%3Ddzone%26utm_medium%3Dpartner-resources%26utm_campaign%3DCOLL-20XX-Reactive-Microservices-Architecture%26utm_term%3Dnone%26utm_content%3Dnone)*.  Brought to you in partnership with [Lightbend](https://dzone.com/go?i=153025&u=https%3A%2F%2Finfo.lightbend.com%2FCOLL-20XX-Reactive-Microservices-Architecture-RES-LP.html%3Flst%3DDZ%3Futm_source%3Ddzone%26utm_medium%3Dpartner-resources%26utm_campaign%3DCOLL-20XX-Reactive-Microservices-Architecture%26utm_term%3Dnone%26utm_content%3Dnone" \t "_blank).*

As follow up to previous posts covering [JMS](http://city81.blogspot.co.uk/search/label/JMS), this post will delve into more depth on Spring's CachingConnectionFactory.  
  
Spring provides two implementations of the javax.jms.ConnectionFactory interface, namely, the SingleConnectionFactory and the CachingConnectionFactory. The SingleConnectionFactory returns as you might expect the same single connection upon all calls to the createConnection() method. This is fine for certain scenarios and applications but the CachingConnectionFactory provides a more performant and scalable solution.  
  
By default, a single session is cached so for a multi threaded application you would set the sessionCacheSize to be a more suitable number although this number wouldn't reflect the true number of sessions cached as this figure refers to the size of cache per session acknowledgement type eg AUTO\_ACKNOWLEDGE, CLIENT\_ACKNOWLEDGE, DUPS\_OK\_ACKNOWLEDGE and SESSION\_TRANSACTED.  
  
By default, the CachingConnectionFactory will cache the Message Producers and Message Consumers for every session. As an aside the Message Consumers are cached using keys which include the JMS selector so the more fine grained the message filter the more Message Consumers there would be, and Message Consumers aren't closed until the session is closed and removed from the pool. An alternative is to use a Listener Container for consuming messages.  
  
Also to be noted is that on creating a CachingConnectionFactory instance, the reconnect on exception flag is set to be true. This should mean that the onException method on the default ExceptionListener class gets called which will reset the connections. You can also override the default exception listener with your own implementation.  
  
The below snippet of XML shows a simple configuration of a CachingConnectionFactory:

<bean id="jmsQueueConnectionFactory"

    class="org.springframework.jms.connection.UserCredentialsConnectionFactoryAdapter">

    <property name="targetConnectionFactory" ref="mqConnectionFactory" />

    <property name="username" value="${mq.username}" />

</bean>

<bean id="jmsConnectionFactory"

    class="org.springframework.jms.connection.CachingConnectionFactory">

    <property name="targetConnectionFactory" ref="jmsQueueConnectionFactory" />

    <property name="sessionCacheSize" value="50" />

    <property name="exceptionListener" ref="jmsExceptionListener" />

</bean>

<bean id="jmsExceptionListener"

    class="com.city81.messaging.MessageMQExceptionListener"">

    <property name="cachingConnectionFactory" ref="jmsConnectionFactory" /">

</bean>

JMS TEMPLATE E PERFOMANCES:

<https://singztechmusings.wordpress.com/2011/04/24/problem-with-creating-jms-messageproducer-using-spring-jmstemplate-how-to-solve/>

USING CACHINGCONNECTIONFACTORY WITH SPRING JMSTEMPLATE: WHAT PROBLEM DOES IT SOLVE?

*April 24, 2011 · by*[Singaram Subramanian](https://singztechmusings.wordpress.com/author/singztechmusings/" \o "Posts by Singaram Subramanian)*· in [ActiveMQ](https://singztechmusings.wordpress.com/category/open-source/activemq/),*[*Caching*](https://singztechmusings.wordpress.com/category/design-decisions/caching/) *·*[*1 Comment*](https://singztechmusings.wordpress.com/2011/04/24/problem-with-creating-jms-messageproducer-using-spring-jmstemplate-how-to-solve/#comments)

9 Votes

From Spring framework API docs for **org.springframework.jms.core.JmsTemplate**,

*“…The ConnectionFactory used with this template should return pooled Connections (or a single shared Connection) as well as pooled Sessions and MessageProducers. Otherwise, performance of ad-hoc JMS operations is going to suffer.*

*The simplest option is to use the Spring-provided SingleConnectionFactory as a decorator for your target ConnectionFactory, reusing a single JMS Connection in a thread-safe fashion; this is often good enough for the purpose of sending messages via this template.*

*In a J2EE environment, make sure that the ConnectionFactory is obtained from the application’s environment naming context via JNDI; application servers typically expose pooled, transaction-aware factories there.”*

JmsTemplate code uses several anti-patterns like *creating a new connection, session, producer/consumer just to send/receive a message, then closing them again*. It’s designed for use in EJBs using the EJB containers’ JMS pooling abstraction. So, without pooling, this’s probably one of the *worst ways of working with JMS*.

Now, lets come to the solution for JMS MessageProducer. We’ll see about using MessageListener in place of MessageConsumer in the next post.

If you’ve written code this way for MessageProducer based on JmsTemplate (see below – Problematic code), it can easily be fixed by the mentioned approach using a CachedConnectionFactory:

I’m assuming you’re using ActiveMQ as Message Broker, but this applies for any broker.

**Problematic Code**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | ...  <!-- A connection to ActiveMQ -->  <bean id="amqConnectionFactory"      class="org.apache.activemq.ActiveMQConnectionFactory"      p:brokerURL='<tcp://localhost:61616>" />    <!-- A destination in ActiveMQ -->  <bean id="destination"      class="org.apache.activemq.command.ActiveMQQueue">  <constructor-arg value="FOO.TEST" />  </bean>    <!-- A JmsTemplate instance that uses the uncached connection and destination -->  <bean id="producerTemplate"      class="org.springframework.jms.core.JmsTemplate"      p:connectionFactory-ref="amqConnectionFactory"      p:defaultDestination-ref="destination" />  ... |

**Fix: Using CachingConnectionFactory**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | ...  <!-- A connection to ActiveMQ -->  <bean id="amqConnectionFactory"      class="org.apache.activemq.ActiveMQConnectionFactory"      p:brokerURL='<tcp://localhost:61616>" />    <!-- A cached connection to wrap the ActiveMQ connection -->  <bean id="cachedConnectionFactory"      class="org.springframework.jms.connection.CachingConnectionFactory"      p:targetConnectionFactory-ref="amqConnectionFactory"      p:sessionCacheSize="10" />    <!-- A destination in ActiveMQ -->  <bean id="destination"      class="org.apache.activemq.command.ActiveMQQueue">  <constructor-arg value="FOO.TEST" />  </bean>    <!-- A JmsTemplate instance that uses the cached connection and destination -->  <bean id="producerTemplate"      class="org.springframework.jms.core.JmsTemplate"      p:connectionFactory-ref="cachedConnectionFactory"      p:defaultDestination-ref="destination" />  ... |

From the API docs for **org.springframework.jms.connection.CachingConnectionFactory**,

*This is a SingleConnectionFactory subclass that adds Session caching as well MessageProducer caching. This ConnectionFactory also switches the “reconnectOnException” property to “true” by default, allowing for automatic recovery of the underlying Connection.*

*By default, only one single Session will be cached, with further requested Sessions being created and disposed on demand. Consider raising the “sessionCacheSize” value in case of a high-concurrency environment.*

*This ConnectionFactory requires explicit closing of all Sessions obtained from its shared Connection. This is the usual recommendation for native JMS access code anyway. However, with this ConnectionFactory, its use is mandatory in order to actually allow for Session reuse.*

*Note also that MessageConsumers obtained from a cached Session won’t get closed until the Session will eventually be removed from the pool. This may lead to semantic side effects in some cases. For a durable subscriber, the logical Session.close() call will also close the subscription. Re-registering a durable consumer for the same subscription on the same Session handle is not supported; close and reobtain a cached Session first.*

POOLEDCONNECTIONFACTORY VS CACHINGCONNECTIONFACTORY: WHICH ONE IS A PERFECT MATCH FOR SPRING JMSTEMPLATE?

*June 21, 2011 · by*[Singaram Subramanian](https://singztechmusings.wordpress.com/author/singztechmusings/" \o "Posts by Singaram Subramanian)*· in [ActiveMQ](https://singztechmusings.wordpress.com/category/open-source/activemq/),*[*Spring*](https://singztechmusings.wordpress.com/category/java-frameworks/spring/) *·*[*Leave a comment*](https://singztechmusings.wordpress.com/2011/06/21/pooledconnectionfactory-vs-cachingconnectionfactory-which-one-is-a-perfect-match-for-spring-jmstemplate/#respond)

4 Votes

JmsTemplate, part of Core Spring [JMS](http://en.wikipedia.org/wiki/Java_Message_Service) framework, simplifies the use of JMS since it handles the creation and release of resources when sending or synchronously receiving messages. As discussed in this post – <https://singztechmusings.wordpress.com/2011/04/24/problem-with-creating-jms-messageproducer-using-spring-jmstemplate-how-to-solve/> – we need to have pooling in place to make it efficient.

We’ve got two JMS provider choices: [ActiveMQ](http://activemq.apache.org/" \o "Apache ActiveMQ)‘s PooledConnectionFactory or Spring’s own CachingConnectionFactory

They’re meant for the same purpose – to pool [Connection](http://en.wikipedia.org/wiki/Database_connection), Session and MessageProducer instances. But we need to consider a thing or two before deciding to use one of these:

1. If you have clustered ActiveMQs, and use [failover](http://en.wikipedia.org/wiki/Failover) transport (Eg. failover:(tcp://firstbox:61616,tcp://secondbox:61616)?timeout=30000), it has been reported that CachingConnectionFactory is not a right choice.

The problem I’m having is that if one box goes down, we should start sending messages on the other, but it seems to still be using the old connection (every send times out). If I restart the program, it’ll connect again and everything works.  
Source: <http://stackoverflow.com/questions/5916638/autoreconnect-problem-with-activemq-and-cachingconnectionfactory>

The problem is that cached connections to the failed ActiveMQ was still in use and that created the problem for the user. Now, the choice for this scenario is PooledConnectionFactory.

2. If you’re using ActiveMQ today, and chances are that you may switch to some other broker ([JBoss](http://www.jboss.com/products/platforms/application/" \o "JBoss application server) MQ, [WebSphere MQ](http://en.wikipedia.org/wiki/IBM_WebSphere_MQ" \o "IBM WebSphere MQ)) in future, do not use PooledConnectionFactory, as it tightly couples your code to ActiveMQ.

# ConnectionFactories and Caching with Spring and ActiveMQ

<https://codedependents.com/2010/07/14/connectionfactories-and-caching-with-spring-and-activemq/>

In my previous [two](https://codedependents.com/2009/10/16/efficient-lightweight-jms-with-spring-and-activemq/) [posts](https://codedependents.com/2010/03/04/synchronous-request-response-with-activemq-and-spring/) on ActiveMQ and Spring I’ve shown how to implement both asynchronous and synchronous messaging using MessageListenerContainers and JmsTemplates. Since then I have continued to tweak and improve my use of Spring’s JMS support and have uncovered a few more details about the ConnectionFactories supplied by both ActiveMQ and Spring and how they play together nicely and not so nicely. Some of this info is new while some was covered in my second post and lead to an edit in my initial post but I will reiterate that information here to keep it all in one place.

## [PooledConnectionFactory](http://activemq.apache.org/maven/activemq-pool/apidocs/org/apache/activemq/pool/PooledConnectionFactory.html)

This bad boy is the ActiveMQ provide connection pooler. It pools connections, sessions and producers and does a fine job of it. There is nothing particularly wrong with this connection factory though it does strike me as odd that it caches connections since, according to the JMS spec, connections support concurrent use.

## [SingleConnectionFactory](http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/jms/connection/SingleConnectionFactory.html)

It might seem odd to have a connection factory that ensures only one connection but in reality this is quite useful. When using multiple JmsTemplates and MessageListenerContainers in Spring they will end up each creating their own connection since the default ConnectionFactory will create a new connection for each createConnection() call. Initially I thought this would be fine but I ran into some very weird issues where producers and consumers couldn’t see each other. I’m not sure if this had something to do with using embedded brokers and the VM Transport but it wasn’t good. Using a SingleConnectionFactory as the initial decorator around the default ConnectionFactory solved these issues.

## [CachingConnectionFactory](http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/jms/connection/CachingConnectionFactory.html)

This is the real heavy lifter. It will cache sessions and can optionally cache consumers and producers. As any post on JmsTemplate will tell you, you MUST use caching for any sort of reasonable performance if you are not running in a JCA container. As such, this is the factory to use when using a JmsTemplate. A few points should be made clear though. First, the default is for only one session to be cached. The javadoc claims this is sufficient for low concurrency but if you are doing something more high end you probably want to increase the SessionCacheSize. Additionally, there is some weirdness around cached consumers. They don’t get closed until they are removed from the pool and they are cached using a key that contains the destination and the selector. This lead to essentially a memory leak when trying to use cached consumers for request response semantics using JMSCorrelationIds. In general caching consumers is hard which is why MessageListenerContainers should be used instead.

## How’s it all work

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <amq:connectionFactory id="connectionFactory" brokerURL="${jms.url}"/>  <bean id="singleConnectionFactory" p:targetConnectionFactory-ref="connectionFactory" p:reconnectOnException="true"/>  <bean id="cachingConnectionFactory" p:targetConnectionFactory-ref="singleConnectionFactory" p:sessionCacheSize="100"/>  <bean id="jmsTemplate" p:connectionFactory-ref="cachingConnectionFactory"/>  <jms:listener-container connection-factory="singleConnectionFactory">  <jms:listener id="QueueHandler" destination="Queue" ref="queueHandler"/>  </jms:listener-container> |

First we start with a basic ActiveMQ ConnectionFactory. From there we wrap it in a SingleConnectionFactory and then wrap that in a CachingConnectionFactory. I’ve increased the SessionCacheSize on the CachingConnectionFactory to 100, this might be overkill but my application is anything but memory hungry so I figured I could err on the side of too many sessions. Additionally, and this is important, ReconnectOnException is set to true on the SingleConnectionFactory. The CachingConnectionFactory requires this and will complain loudly but in such as way as to make it unclear what is going on if this is not set. From there we use the CachingConnectionFactory to create a JmsTemplate and we use the SingleConnectionFactory to create a MessageListenerContainer since it takes care of any caching of sessions and consumers that it needs.

# Synchronous Request Response with ActiveMQ and Spring

# https://codedependents.com/2010/03/04/synchronous-request-response-with-activemq-and-spring/

A few months ago I did a deep dive into [Efficient Lightweight JMS with Spring and ActiveMQ](https://codedependents.com/2009/10/16/efficient-lightweight-jms-with-spring-and-activemq/)where I focused on the details for asynchronous sending and receiving of messages. In an ideal world all messaging would be asynchronous. If you need a response then you should set up an asynchronous listener and either have enough state stored in the service or in the message that you can continue processing once the response has been received. However, when ideals lead to complexity, we have to make the decision of how much complexity can we tolerate for the performance we need. Sometimes it just makes more sense to use a synchronous request/response.

## Request Response with JMS

ActiveMQ documentation actually has a pretty good overview of [how request-response semantics work in JMS](http://activemq.apache.org/how-should-i-implement-request-response-with-jms.html).

*You might think at first that to implement request-response type operations in JMS that you should create a new consumer with a selector per request; or maybe create a new temporary queue per request.*

*Creating temporary destinations, consumers, producers and connections are all synchronous request-response operations with the broker and so should be avoided for processing each request as it results in lots of chat with the JMS broker.*

*The best way to implement request-response over JMS is to create a temporary queue and consumer per client on startup, set JMSReplyTo property on each message to the temporary queue and then use a correlationID on each message to correlate request messages to response messages. This avoids the overhead of creating and closing a consumer for each request (which is expensive). It also means you can share the same producer & consumer across many threads if you want (or pool them maybe).*

This is a pretty good start but it requires some tweaking to work best in Spring. It also should be noted that [Lingo](http://lingo.codehaus.org/home) and [Camel](http://camel.apache.org/) are also suggested as options when using ActiveMQ. In my [previous post](http://activemq.apache.org/how-should-i-implement-request-response-with-jms.html) I addressed why I don’t use either of these options. In short Camel is more power than is needed for basic messaging and Lingo is built on [Jencks](http://jencks.codehaus.org/Home), neither of which have been updated in years.

## Request Response in Spring

The first thing to notice is that its infeasible to create a consumer and temporary queue per client in Spring since pooling resources is required overcome the [JmsTemplate gotchas](http://activemq.apache.org/jmstemplate-gotchas.html). To get around this, I suggest using predefined request and response queues, removing the overhead of creating a temporary queue for each request/response. To allow for multiple consumers and producers on the same queue the JMSCorrelationId is used to correlated the request with its response message.

At this point I implemented the following naive solution:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | @Component  public class Requestor {        private static final class CorrelationIdPostProcessor implements MessagePostProcessor {            private final String correlationId;            public CorrelationIdPostProcessor( final String correlationId ) {              this.correlationId = correlationId;          }            @Override          public Message postProcessMessage( final Message msg ) throws JMSException {              msg.setJMSCorrelationID( correlationId );              return msg;          }      }        private final JmsTemplate jmsTemplate;        @Autowired      public RequestGateway( JmsTemplate jmsTemplate ) {          this.jmsTemplate = jmsTemplate;      }        public String request( final String request, String queue ) throws IOException {          final String correlationId = UUID.randomUUID().toString();          jmsTemplate.convertAndSend( queue+".request", request, new CorrelationIdPostProcessor( correlationId ) );          return (String) jmsTemplate.receiveSelectedAndConvert( queue+".response", "JMSCorrelationID='" + correlationId + "'" );      }  } |

This worked for a while until the system started occasionally timing out when making a request against a particularly fast responding service. After some debugging it became apparent that the service was responding so quickly that the receive() call was not fully initialized, causing it to miss the message. Once it finished initializing, it would wait until the timeout and fail. Unfortunately, there is very little in the way of documentation for this and [the best suggestion](http://forum.springsource.org/showpost.php?p=141588&postcount=7) I could find still seemed to leave open the possibility for the race condition by creating the consumer after sending the message. Luckily, according to the [JMS spec](http://technology-related.com/j2ee/1.4/docs/tutorial/doc/JMS4.html#wp79145), a consumer becomes active as soon as it is created and, assuming the connection has been started, it will start consuming messages. This allows for the reordering of the method calls leading to the slightly more verbose but also more correct solution. (**NOTE**: Thanks to Aaron Korver for pointing out that ProducerConsumer needs to implement SessionCallback and that true needs to be passed to the JmsTemplate.execute() for the connection to be started.)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59 | @Component  public class Requestor {        private static final class ProducerConsumer implements SessionCallback<Message> {            private static final int TIMEOUT = 5000;            private final String msg;            private final DestinationResolver destinationResolver;            private final String queue;            public ProducerConsumer( final String msg, String queue, final DestinationResolver destinationResolver ) {              this.msg = msg;              this.queue = queue;              this.destinationResolver = destinationResolver;          }            public Message doInJms( final Session session ) throws JMSException {              MessageConsumer consumer = null;              MessageProducer producer = null;              try {                  final String correlationId = UUID.randomUUID().toString();                  final Destination requestQueue =                          destinationResolver.resolveDestinationName( session, queue+".request", false );                  final Destination replyQueue =                          destinationResolver.resolveDestinationName( session, queue+".response", false );                  // Create the consumer first!                  consumer = session.createConsumer( replyQueue, "JMSCorrelationID = '" + correlationId + "'" );                  final TextMessage textMessage = session.createTextMessage( msg );                  textMessage.setJMSCorrelationID( correlationId );                  textMessage.setJMSReplyTo( replyQueue );                  // Send the request second!                  producer = session.createProducer( requestQueue );                  producer.send( requestQueue, textMessage );                  // Block on receiving the response with a timeout                  return consumer.receive( TIMEOUT );              }              finally {                  // Don't forget to close your resources                  JmsUtils.closeMessageConsumer( consumer );                  JmsUtils.closeMessageProducer( producer );              }          }      }        private final JmsTemplate jmsTemplate;        @Autowired      public Requestor( final JmsTemplate jmsTemplate ) {          this.jmsTemplate = jmsTemplate;       }        public String request( final String request, String queue ) {          // Must pass true as the second param to start the connection          return (String) jmsTemplate.execute( new ProducerConsumer( msg, queue, jmsTemplate.getDestinationResolver() ), true );      }  } |

## About Pooling

Once the request/response logic was correct it was time to load test. Almost instantly, memory usage exploded and the garbage collector started thrashing. Inspecting ActiveMQ with the [Web Console](http://activemq.apache.org/web-console.html) showed that MessageConsumers were hanging around even though they were being explicitly closed using Spring’s own JmsUtils. Turns out, the [CachingConnectionFactory](http://static.springsource.org/spring/docs/2.5.x/api/org/springframework/jms/connection/CachingConnectionFactory.html)‘s JavaDoc held the key to what was going on: “Note also that MessageConsumers obtained from a cached Session won’t get closed until the Session will eventually be removed from the pool.” However, if the MessageConsumers could be reused this wouldn’t be an issue. Unfortunately, CachingConnectionFactory caches MessageConsumers based on a hash key which contains the selector among other values. Obviously each request/response call, with its necessarily unique selector, was creating a new consumer that could never be reused. Luckily ActiveMQ provides a [PooledConnectionFactory](http://activemq.apache.org/maven/activemq-core/apidocs/org/apache/activemq/pool/PooledConnectionFactory.html) which does not cache MessageConsumers and switching to it fixed the problem instantly. However, this means that each request/response requires a new MessageConsumer to be created. This is adds overhead but its the price that must be payed to do synchronous request/response.

## SimpleMessageListenerContainer

Message listener container that uses the plain JMS client API's MessageConsumer.setMessageListener() method to create concurrent MessageConsumers for the specified listeners.

This is the simplest form of a message listener container. It creates a fixed number of JMS Sessions to invoke the listener, not allowing for dynamic adaptation to runtime demands. Its main advantage is its low level of complexity and the minimum requirements on the JMS provider: Not even the ServerSessionPool facility is required.

See the [AbstractMessageListenerContainer](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/listener/AbstractMessageListenerContainer.html" \o "class in org.springframework.jms.listener) javadoc for details on acknowledge modes and transaction options. Note that this container exposes standard JMS behavior for the default "AUTO\_ACKNOWLEDGE" mode: that is, automatic message acknowledgment after listener execution, with no redelivery in case of a user exception thrown but potential redelivery in case of the JVM dying during listener execution.

For a different style of MessageListener handling, through looped MessageConsumer.receive() calls that also allow for transactional reception of messages (registering them with XA transactions), see [DefaultMessageListenerContainer](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/listener/DefaultMessageListenerContainer.html" \o "class in org.springframework.jms.listener).

## DefaultMessageListenerContainer

Message listener container variant that uses plain JMS client APIs, specifically a loop of MessageConsumer.receive() calls that also allow for transactional reception of messages (registering them with XA transactions). Designed to work in a native JMS environment as well as in a Java EE environment, with only minimal differences in configuration.

This is a simple but nevertheless powerful form of message listener container. On startup, it obtains a fixed number of JMS Sessions to invoke the listener, and optionally allows for dynamic adaptation at runtime (up to a maximum number). Like [SimpleMessageListenerContainer](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/listener/SimpleMessageListenerContainer.html" \o "class in org.springframework.jms.listener), its main advantage is its low level of runtime complexity, in particular the minimal requirements on the JMS provider: not even the JMS ServerSessionPool facility is required. Beyond that, it is fully self-recovering in case the broker is temporarily unavailable, and allows for stops/restarts as well as runtime changes to its configuration.

Actual MessageListener execution happens in asynchronous work units which are created through Spring's [TaskExecutor](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/core/task/TaskExecutor.html" \o "interface in org.springframework.core.task) abstraction. By default, the specified number of invoker tasks will be created on startup, according to the ["concurrentConsumers"](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/listener/DefaultMessageListenerContainer.html#setConcurrentConsumers-int-) setting. Specify an alternative TaskExecutor to integrate with an existing thread pool facility (such as a Java EE server's), for example using a [CommonJ WorkManager](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/scheduling/commonj/WorkManagerTaskExecutor.html" \o "class in org.springframework.scheduling.commonj). With a native JMS setup, each of those listener threads is going to use a cached JMS Session and MessageConsumer (only refreshed in case of failure), using the JMS provider's resources as efficiently as possible.

Message reception and listener execution can automatically be wrapped in transactions by passing a Spring [PlatformTransactionManager](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/transaction/PlatformTransactionManager.html" \o "interface in org.springframework.transaction) into the ["transactionManager"](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/listener/AbstractPollingMessageListenerContainer.html#setTransactionManager-org.springframework.transaction.PlatformTransactionManager-) property. This will usually be a [JtaTransactionManager](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/transaction/jta/JtaTransactionManager.html" \o "class in org.springframework.transaction.jta) in a Java EE environment, in combination with a JTA-aware JMS ConnectionFactory obtained from JNDI (check your Java EE server's documentation). Note that this listener container will automatically reobtain all JMS handles for each transaction in case an external transaction manager is specified, for compatibility with all Java EE servers (in particular JBoss). This non-caching behavior can be overridden through the ["cacheLevel"](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/listener/DefaultMessageListenerContainer.html#setCacheLevel-int-) / ["cacheLevelName"](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/listener/DefaultMessageListenerContainer.html#setCacheLevelName-java.lang.String-) property, enforcing caching of the Connection (or also Session and MessageConsumer) even if an external transaction manager is involved.

Dynamic scaling of the number of concurrent invokers can be activated by specifying a ["maxConcurrentConsumers"](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/listener/DefaultMessageListenerContainer.html#setMaxConcurrentConsumers-int-) value that is higher than the ["concurrentConsumers"](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/listener/DefaultMessageListenerContainer.html#setConcurrentConsumers-int-) value. Since the latter's default is 1, you can also simply specify a "maxConcurrentConsumers" of e.g. 5, which will lead to dynamic scaling up to 5 concurrent consumers in case of increasing message load, as well as dynamic shrinking back to the standard number of consumers once the load decreases. Consider adapting the ["idleTaskExecutionLimit"](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/listener/DefaultMessageListenerContainer.html#setIdleTaskExecutionLimit-int-) setting to control the lifespan of each new task, to avoid frequent scaling up and down, in particular if the ConnectionFactory does not pool JMS Sessions and/or the TaskExecutor does not pool threads (check your configuration!). Note that dynamic scaling only really makes sense for a queue in the first place; for a topic, you will typically stick with the default number of 1 consumer, otherwise you'd receive the same message multiple times on the same node.

**Note: Don't use Spring's [CachingConnectionFactory](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/connection/CachingConnectionFactory.html" \o "class in org.springframework.jms.connection) in combination with dynamic scaling.** Ideally, don't use it with a message listener container at all, since it is generally preferable to let the listener container itself handle appropriate caching within its lifecycle. Also, stopping and restarting a listener container will only work with an independent, locally cached Connection - not with an externally cached one.

**It is strongly recommended to either set**[**"sessionTransacted"**](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/listener/AbstractPollingMessageListenerContainer.html#setSessionTransacted-boolean-)**to "true" or specify an external**[**"transactionManager"**](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/listener/AbstractPollingMessageListenerContainer.html#setTransactionManager-org.springframework.transaction.PlatformTransactionManager-)**.** See the [AbstractMessageListenerContainer](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/listener/AbstractMessageListenerContainer.html" \o "class in org.springframework.jms.listener) javadoc for details on acknowledge modes and native transaction options, as well as the [AbstractPollingMessageListenerContainer](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/listener/AbstractPollingMessageListenerContainer.html" \o "class in org.springframework.jms.listener) javadoc for details on configuring an external transaction manager. Note that for the default "AUTO\_ACKNOWLEDGE" mode, this container applies automatic message acknowledgment before listener execution, with no redelivery in case of an exception.

**SingleConnectionFactory**

extends [Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html?is-external=true)

implements [ConnectionFactory](https://docs.oracle.com/javaee/7/api/javax/jms/ConnectionFactory.html?is-external=true), [QueueConnectionFactory](https://docs.oracle.com/javaee/7/api/javax/jms/QueueConnectionFactory.html?is-external=true), [TopicConnectionFactory](https://docs.oracle.com/javaee/7/api/javax/jms/TopicConnectionFactory.html?is-external=true), [ExceptionListener](https://docs.oracle.com/javaee/7/api/javax/jms/ExceptionListener.html?is-external=true), [InitializingBean](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/beans/factory/InitializingBean.html), [DisposableBean](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/beans/factory/DisposableBean.html)

A JMS ConnectionFactory adapter that returns the same Connection from all [createConnection()](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/connection/SingleConnectionFactory.html" \l "createConnection--) calls, and ignores calls to [Connection.close()](https://docs.oracle.com/javaee/7/api/javax/jms/Connection.html?is-external=true" \l "close--" \o "class or interface in javax.jms). According to the JMS Connection model, this is perfectly thread-safe (in contrast to e.g. JDBC). The shared Connection can be automatically recovered in case of an Exception.

You can either pass in a specific JMS Connection directly or let this factory lazily create a Connection via a given target ConnectionFactory. This factory generally works with JMS 1.1 as well as the JMS 1.0.2 API.

Note that when using the JMS 1.0.2 API, this ConnectionFactory will switch into queue/topic mode according to the JMS API methods used at runtime: createQueueConnection and createTopicConnection will lead to queue/topic mode, respectively; generic createConnection calls will lead to a JMS 1.1 connection which is able to serve both modes.

Useful for testing and standalone environments in order to keep using the same Connection for multiple [JmsTemplate](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/core/JmsTemplate.html" \o "class in org.springframework.jms.core) calls, without having a pooling ConnectionFactory underneath. This may span any number of transactions, even concurrently executing transactions.

Note that Spring's message listener containers support the use of a shared Connection within each listener container instance. Using SingleConnectionFactory in combination only really makes sense for sharing a single JMS Connection *across multiple listener containers*.

**CachingConnectionFactory**

extends [SingleConnectionFactory](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/connection/SingleConnectionFactory.html)

[SingleConnectionFactory](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/connection/SingleConnectionFactory.html) subclass that adds [Session](https://docs.oracle.com/javaee/7/api/javax/jms/Session.html?is-external=true) caching as well [MessageProducer](https://docs.oracle.com/javaee/7/api/javax/jms/MessageProducer.html?is-external=true" \o "class or interface in javax.jms) caching. This ConnectionFactory also switches the ["reconnectOnException" property](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/connection/SingleConnectionFactory.html#setReconnectOnException-boolean-) to "true" by default, allowing for automatic recovery of the underlying Connection.

By default, only one single Session will be cached, with further requested Sessions being created and disposed on demand. Consider raising the ["sessionCacheSize" value](http://docs.spring.io/spring-framework/docs/4.3.8.RELEASE/javadoc-api/org/springframework/jms/connection/CachingConnectionFactory.html#setSessionCacheSize-int-) in case of a high-concurrency environment.

When using the JMS 1.0.2 API, this ConnectionFactory will switch into queue/topic mode according to the JMS API methods used at runtime: createQueueConnection and createTopicConnection will lead to queue/topic mode, respectively; generic createConnection calls will lead to a JMS 1.1 connection which is able to serve both modes.

**NOTE: This ConnectionFactory requires explicit closing of all Sessions obtained from its shared Connection.** This is the usual recommendation for native JMS access code anyway. However, with this ConnectionFactory, its use is mandatory in order to actually allow for Session reuse.

Note also that MessageConsumers obtained from a cached Session won't get closed until the Session will eventually be removed from the pool. This may lead to semantic side effects in some cases. For a durable subscriber, the logical Session.close() call will also close the subscription. Re-registering a durable consumer for the same subscription on the same Session handle is not supported; close and reobtain a cached Session first.

**DESTINATION RESOLVERS:**

<http://koenserneels.blogspot.com.ee/2013/04/jms-and-spring-small-things-sometimes.html>

### JMS and Spring: Small Things Sometimes Matter

[JmsTemplate](http://static.springsource.org/spring/docs/current/javadoc-api/org/springframework/jms/core/JmsTemplate.html) and [DefaultMessageListenerContainer](http://static.springsource.org/spring/docs/current/javadoc-api/org/springframework/jms/listener/DefaultMessageListenerContainer.html" \t "_blank) are Spring helpers for accessing JMS compatible MOM. Their main goal is to form a layer above the JMS API and deal with infrastructure such as transaction management/message acknowledgement and hiding some of the repetitive and clumsy parts of the JMS API (hang in there: [JMS 2.0](http://java.net/projects/jms-spec/pages/Home)is on its way!). To use either one of these helpers you have to supply them with (at least) a [JMS ConnectionFactory](http://docs.oracle.com/javaee/6/api/javax/jms/ConnectionFactory.html) and a valid [JMS Destination](http://docs.oracle.com/javaee/6/api/javax/jms/Destination.html).

When running your app on an application server, the ConnectionFactory will most likely be defined using the JEE architecture. This boils down adding the ConnectionFactory and its configuration parameters allowing them to be published in the directory service under a given alias (eg. jms/myConnectionFactory). Within your app you might for example use the "jndi-lookup" out of the JEE namespace or JndiTemplate/JndiObjectFactoryBean beans if more configuration is required for looking up the ConnectionFactory and pass it along to your JmsTemplate and/or DefaultMessageListenerContainer.

The latter, JMS destination, identifies a JMS Queue or Topic for which you want to produce messages to or consume mesages from. However, both JmsTemplate as DefaultMessageListenerContainer have two different properties for injecting the destination. There is a method taking the destination as [String](http://static.springsource.org/spring/docs/current/javadoc-api/org/springframework/jms/listener/AbstractMessageListenerContainer.html#setDestinationName%28java.lang.String%29) and one taking it as a [JMS Destination](http://static.springsource.org/spring/docs/current/javadoc-api/org/springframework/jms/listener/AbstractMessageListenerContainer.html#setDestination%28javax.jms.Destination%29) type. This functionality is nothing invented by Spring, the [JMS specification](http://www.oracle.com/technetwork/java/docs-136352.html)mentions both approaches:

|  |  |
| --- | --- |
| 1  2  3  4 | 4.4.4 Creating Destination Objects  Most clients will use Destinations that are JMS administered objects that they have looked up via JNDI. This is the most portable approach.  Some specialized clients may need to create Destinations by dynamically manufacturing one using a provider-specific destination name.  Sessions provide a JMS provider-specific method for doing this. |

If you pass along a destination as String then the helpers will hide the extra steps required to map them to a valid JMS Destination. In the end a [createConsumer on a JMS Session](http://docs.oracle.com/javaee/6/api/javax/jms/Session.html" \l "createConsumer%28javax.jms.Destination%29" \t "_blank) expects you to pass along a Destination object to indicate where to consume messages from before returning a [MessageConsumer](http://docs.oracle.com/javaee/6/api/javax/jms/MessageConsumer.html" \t "_blank).

When destinations are configured as String, the Destination is looked up by Spring using the JMS API itself. By default JmsTemplate and DefaultMessageListenerContainer have a reference to a [DestinationResolver](http://static.springsource.org/spring/docs/current/javadoc-api/org/springframework/jms/support/destination/DestinationResolver.html" \t "_blank) which is [DynamicDestinationResolver](http://static.springsource.org/spring/docs/current/javadoc-api/org/springframework/jms/support/destination/DynamicDestinationResolver.html" \t "_blank) by default (more on that later). The code below is an extract from DynamicDestinationResolver, the highlighted lines indicate the usage of the JMS API to transform the String to a Destination (in this example a Queue):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | protected Queue resolveQueue(Session session, String queueName) throws JMSException {   if (session instanceof QueueSession) {    // Cast to QueueSession: will work on both JMS 1.1 and 1.0.2    return ((QueueSession) session).createQueue(queueName);   }   else {    // Fall back to generic JMS Session: will only work on JMS 1.1    return session.createQueue(queueName);   }  } |

The other way mentioned by the spec (JNDI approach) is to configure Destinations as administrable objects on your application server. This follows the principle as with the ConnectionFactory; the destination is published in the applications servers directory and can be looked up by its JNDI name (eg. jms/myQueue). Again you can lookup the JMS Destination in your app and pass it along to JmsTemplate and/or DefaultMessageListenerContainer making use of the property taking the JMS Destination as parameter.

Now, why do we have those two options?

I always assumed that it was a matter of choice between convenience (dynamic approach) and environment transparancy/configurability (JNDI approach). For example: in some situations the name of the physical destination might be different depending on the environment where your application runs. If you configure your physical destination names inside your application you obviously loose this benefit as they cannot be altered without rebuilding your application. If you configured them as administered object on the other hand, it is merely a simple change in the application server configuration to alter the physical destination name.

Remember; having physical Destinations names configurable can make sense. Besides the Destination type, applications dealing with messaging are agnostic to its details. A messaging destination has no functional contract and none of its properties (physical destination, persistence, and so forth) are of importance for the code your write. The actual contract is inside the messages itself (the headers and the body). A database table on the other is an example of something that does expose a contract by itself and is tightly coupled with your code. In most cases renaming a database table does impact your code, hence making something like this configurable has normally no added value compared to a messaging Destination.

Recently I discovered that my understanding of this is not the entire truth. The specification (from "4.4.4 Creating Destination Objects" as pasted some paragraphs above) already gives a hint: "Most clients will use Destinations that are JMS administered objects that they have looked up via JNDI. This is the most portable approach." Basically this tells us that the other approach (the dynamic approach where we work with a destination as String) is "the least portable" way. This was never really clear to me as each provider is required to implement both methods, however "portable" has to be looked at in a broader context.

When configuring Destinations as String, Spring will by default transform them to JMS Desintations whenever it creates a new JMS Session. When using the DefaultMessageListenerContainer for consuming messages each message you process occurs in a transaction and by default the JMS session and consumer are not pooled, hence they are re-created for each receive operation. This results in transforming the String to a JMS Destination each time the container checks for new messages and/or receives a new message. The "non portable" aspect comes into play as it also means that the details and costs of this transformation depend entirely on your MOM's driver/implementation. In our case we experienced this with Oracle AQ as MOM provider. Each time a destination transformation happens the driver executes a specific query:

|  |  |
| --- | --- |
| 1  2  3 | select   /\*+ FIRST\_ROWS \*/  t1.owner, t1.name, t1.queue\_table, t1.queue\_type, t1.max\_retries, t1.retry\_delay, t1.retention, t1.user\_comment, t2. type , t2.object\_type, t2.secure  from  all\_queues t1, all\_queue\_tables t2  where  t1.owner=:1 and  t1.name=:2 and  t2.owner=:3 and  t1.queue\_table=t2.queue\_table |

Forum entry [can be found here](https://forums.oracle.com/forums/thread.jspa?threadID=1097902&tstart=4).

Although this query was improved in the latest drivers (as mentioned by the bug report), it was still causing significant overhead on the database. The two options to solve this:

* Do what the specification advices you to do: configure destinations as resources on the application server. The application server will hand out the same instance each time, so they are already cached there. Even though you will receive the same instance for every lookup, when using JndiTemplate (or JndiDestinationResolver, see below) it will also be chached application side, so even the lookup itself will only happen once.
* Enable session/consumer caching on the DefaultMessageListenerContainer. When the caching is set to consumer, it indirectly also re-use the Destination as the consumer holds a reference to the Destination. This pooling is Spring added functionality and the [JavaDoc](http://static.springsource.org/spring/docs/current/javadoc-api/org/springframework/jms/listener/DefaultMessageListenerContainer.html" \l "setCacheLevel%28int%29" \t "_blank) says it safe when using resource local transaction and it "should" be safe when using XA transaction (except running on JBoss 4).

The first is probably the best. However in our case all destinations are already defined inside the application (and there are plenty of them) and there is no need for having them configurable. Refactoring these merely for this technical reason is going to generate a lot of overhead with no other advantages. The second solution is the least preferred one as this would imply extra testing and investigation to make sure nothing breaks. Also, this seems to be doing more then needed, as there is no indication in our case that creating a Session or Consumer has measurable impact on performance. According to the JMS specification:

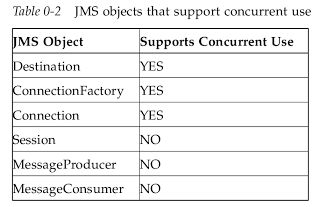
|  |  |
| --- | --- |
| 1  2  3  4 | 4.4 Session  A JMS Session is a single-threaded context\* for producing and consuming  messages. Although it may allocate provider resources outside the Java virtual  machine, it is considered a lightweight JMS object. |

Btw; this is also valid for MessageConsumers/Producers. Both of them are bound to a session, so if a Session is lightweight to open then these objects will be as well.

There is however a third solution; a custom DestinationResolver. The DestinationResolver is the abstraction that takes care of going from a String to a Destination. The default ([DynamicDestinationResolver](http://static.springsource.org/spring/docs/current/javadoc-api/org/springframework/jms/support/destination/DynamicDestinationResolver.html" \t "_blank)) uses the createConsumer(javax.jms.Destination) on the JMS Session to transform, but it does however not cache the resulting Destination. However, if your Destinations are configured on the application server as resources, you can (besides using Spring's JNDI support and injection the Destination directly) also use [JndiDestinationResolver](http://static.springsource.org/spring/docs/current/javadoc-api/org/springframework/jms/support/destination/JndiDestinationResolver.html" \t "_blank). This resolver will treat the supplied String as a JNDI location (instead of physical destination name) and perform a lookup for you. By default it will cache the resulting Destination, avoiding any subsequent JNDI lookups. Now, one can also configure JndiDestinationResolver as a caching decorator for the DynamicDestinationResolver. If you set fallback to true, it will first try to use the String as a location to lookup from JNDI, if that fails it will pass our String along to DynamicDestinationResolver using the JMS API to transform our String to a Destination. The resulting Destination is in both cases cached and thus a next request for the same Destination will be served from the cache. With this resolver there is a solution out of the box without having to write any code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <bean id="cachingDestinationResolver" class="org.springframework.jms.support.destination.JndiDestinationResolver">   <property name="cache" value="true"/>   <property name="fallbackToDynamicDestination" value="true"/>  </bean>    <bean id="infra.abstractMessageListenerContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer" abstract="true">   <property name="destinationResolver" ref="cachingDestinationResolver"/>   ...  </bean> |

The JndiDestinationResolver is thread safe by internally using a ConcurrentHasmap to store the bindings. A JMS Destination is on itself thread safe according to the JMS 1.1 specification (2.8 Multithreading) and can safely be cached:

[](http://4.bp.blogspot.com/-YRrWHJ6iqcM/UVltDJHv16I/AAAAAAAAAVA/YATJMVF2AuA/s1600/jmsObjects.png)

This is again a nice example on how simple things can sometimes have an important impact. This time the solution was straightforward thanks to Spring. It would however been a better idea to make the caching behaviour the default as this would decouple it from any provider specific quirks in looking up the destination. The reason this isn't the default is probably because the DefaultMessageListenerContainer supports [changing the destination on the fly](http://static.springsource.org/spring/docs/current/javadoc-api/org/springframework/jms/listener/AbstractMessageListenerContainer.html#setDestinationName%28java.lang.String%29) (using JMX for example):

|  |  |
| --- | --- |
| 1 | Note: The destination may be replaced at runtime, with the listener container picking up the new destination immediately (works e.g. with DefaultMessageListenerContainer, as long as the cache level is less than CACHE\_CONSUMER). However, this is considered advanced usage; use it with care! |

Posted by Koen Serneels at [1:22 PM](http://koenserneels.blogspot.com.ee/2013/04/jms-and-spring-small-things-sometimes.html)

[Email This](https://www.blogger.com/share-post.g?blogID=3406746278542965235&postID=528221975559689360&target=email)[BlogThis!](https://www.blogger.com/share-post.g?blogID=3406746278542965235&postID=528221975559689360&target=blog)[Share to Twitter](https://www.blogger.com/share-post.g?blogID=3406746278542965235&postID=528221975559689360&target=twitter)[Share to Facebook](https://www.blogger.com/share-post.g?blogID=3406746278542965235&postID=528221975559689360&target=facebook)[Share to Pinterest](https://www.blogger.com/share-post.g?blogID=3406746278542965235&postID=528221975559689360&target=pinterest)

Labels: [JMS](http://koenserneels.blogspot.com.ee/search/label/JMS), [Spring](http://koenserneels.blogspot.com.ee/search/label/Spring)

#### No comments:

# Tuning JMS Message Consumption In Spring

### Posted by: [Bruce Snyder](https://nofluffjuststuff.com/conference/speaker/bruce_snyder) on August 15, 2011

[](http://2.bp.blogspot.com/_pSHP7VYSIjE/S3CoIdTOo7I/AAAAAAAAAcE/1wooXq_IyOs/s1600-h/spring-logo1.jpg)  
  
In a previous blog post titled [Using Spring to Receive JMS Messages](http://bsnyderblog.blogspot.com/2010/02/using-spring-to-receive-jms-messages.html), I introduced the use of the Spring default message listener container for asynchronous consumption of JMS messages. One very common discovery that folks make when first using JMS is that producers can send messages much faster than consumers can receive and process them. When using JMS queues, I always recommend the use of more consumers than you have producers. (When using JMS topics, you should only use a single consumer to guard against receiving the same message multiple times.) This is a normal situation with message-oriented middleware (MOM) and it is easy to handle if you are using the Spring message listener container.   
  
The Spring [DefaultMessageListenerContainer](http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/jms/listener/DefaultMessageListenerContainer.html) (DMLC) is a highly flexible container for consuming JMS messages that can handle many different use cases via the numerous properties that it provides. For the situation mentioned above, the DMLC offers the ability to dynamically scale the number of consumers. That is, as the number of messages available for consumption increases, the DMLC can automatically increase and decrease the number of consumers. To configure the DMLC to automatically scale the number message consumers, the [concurrentConsumers](http://static.springsource.org/spring/docs/2.5.x/api/org/springframework/jms/listener/DefaultMessageListenerContainer.html" \l "setConcurrentConsumers(int)) property and the [maxConcurrentConsumers](https://nofluffjuststuff.com/blog/bruce_snyder/2011/08/tuning_jms_message_consumption_in_spring) property are used. Below is an example JMS namespace style of XML configuration that employs these properties: 

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:jms="http://www.springframework.org/schema/jms"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/jms http://www.springframework.org/schema/jms/spring-jms-3.0.xsd">

<!-- A JMS connection factory for ActiveMQ -->

<bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory"

p:brokerURL="tcp://foo.example.com:61616" />

<!-- A POJO that implements the JMS message listener -->

<bean id="simpleMessageListener" class="com.mycompany.SimpleMessageListener">

<!-- A JMS namespace aware Spring configuration for the message listener container -->

<jms:listener-container

container-type="default"

connection-factory="connectionFactory"

acknowledge="auto"

concurrency="10-50">

<jms:listener destination="TEST.FOO" ref="simpleMessageListener" method="onMessage" />

</jms:listener-container>

</beans>

Notice the concurrency="10-50" property above. This is a simplified configuration for setting the concurrentConsumers=10 and the maxConcurrentConsumers=50 properties of the DMLC. This tells the DMLC to always start up a minimum of 10 consumers. When a new message has been received, if the maxConcurrentConsumers has not been reached and the value of the idleConsumerLimit property has not been reached, then a new consumer is created to process the message. This behavior from the DMLC continues up to the limit set by the maxConcurrentConsumers property. When no messages are being received and the consumers become idle, the number of consumers is automatically decreased.   
  
(NOTE: The idleConsumerLimit property is used to specify the the maximum number of consumers that are allowed to be idle at a given time. The use of this property was recently clarified a bit in the Spring 3.x trunk. Increasing this limit causes invokers to be created more aggressively. This can be useful to ramp up the number of consumers faster.)   
  
It is important to be aware of a couple of things related to this dynamic scaling: 

1. You should not increase the number of concurrent consumers for a JMS topic. This leads to concurrent consumption of the same message, which is hardly ever desirable.
2. The concurrentConsumers property and the maxConcurrentConsumers property can be modified at runtime, e.g., via JMX

The dynamic scaling can be tuned even further through the use of the [idleTaskExecutionLimit](http://static.springsource.org/spring/docs/2.5.x/api/org/springframework/jms/listener/DefaultMessageListenerContainer.html" \l "setIdleTaskExecutionLimit(int))property. The use of this property is best explained by a portion of the Javadoc: 

Within each task execution, a number of message reception attempts (according to the "maxMessagesPerTask" setting) will each wait for an incoming message (according to the "receiveTimeout" setting). If all of those receive attempts in a given task return without a message, the task is considered idle with respect to received messages. Such a task may still be rescheduled; however, once it reached the specified "idleTaskExecutionLimit", it will shut down (in case of dynamic scaling).   
  
Raise this limit if you encounter too frequent scaling up and down. With this limit being higher, an idle consumer will be kept around longer, avoiding the restart of a consumer once a new load of messages comes in. Alternatively, specify a higher "maxMessagesPerTask" and/or "receiveTimeout" value, which will also lead to idle consumers being kept around for a longer time (while also increasing the average execution time of each scheduled task).

Note the recommendations if you experience dynamic scaling taking place too often. To deal with this situation, you should experiment with increases to one or more of the following properties: 

* [idleTaskExecutionLimit](http://static.springsource.org/spring/docs/2.5.x/api/org/springframework/jms/listener/DefaultMessageListenerContainer.html#setIdleTaskExecutionLimit(int)) - The limit for the number of allowed idle executions of a receive task. The default is 1 causing idle resources to be closed early once a task does not receive a message.
* [maxMessagesPerTask](http://static.springsource.org/spring/docs/2.5.x/api/org/springframework/jms/listener/DefaultMessageListenerContainer.html#setMaxMessagesPerTask(int)) - The maximum number of messages to process in a single task. This determines how long a task lives before being reaped. The default is unlimited (-1) so you may not need to change this property.
* [receiveTimeout](http://static.springsource.org/spring/docs/2.5.x/api/org/springframework/jms/listener/AbstractPollingMessageListenerContainer.html#setReceiveTimeout(long)) - The timeout to be used for JMS receive operations. The default is 1000 ms.
* As I noted above, in the Spring 3.x trunk, the **idleConsumerLimit** property was clarified a bit recently and exposed as a writable property. This is yet another property for tuning for situations where you need to ramp up the number of concurrent consumers faster.

One important thing to note about using these various properties for tuning. These are not usable in the JMS namespace style of XML configuration. To use these properties, you must use either a pure Spring XML configuration or straight Java. Below is an example of how to use the receiveTimeout property and the idleTaskExecutionLimit property: 

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- A JMS connection factory for ActiveMQ -->

<bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory"

p:brokerURL="tcp://foo.example.com:61616" />

<!-- The JMS destination -->

<bean id="destination" class="org.apache.activemq.command.ActiveMQQueue"

physicalName="TEST.FOO" />

<!-- A POJO that implements the JMS message listener -->

<bean id="simpleMessageListener" class="com.mycompany.SimpleMessageListener">

<!-- A pure Spring configuration for the message listener container -->

<bean id="msgListenerContainer"

class="org.springframework.jms.listener.DefaultMessageListenerContainer"

p:connectionFactory-ref="connectionFactory"

p:destination-ref="destination"

p:messageListener-ref="simpleMessageListener"

p:concurrentConsumers="10"

p:maxConcurrentConsumers="50"

p:receiveTimeout="5000"

p:idleTaskExecutionLimit="10"

p:idleConsumerLimit="5" />

</beans>

In the example configuration above, the receiveTimeout property is set to five seconds to tell the DMLC's receive operation to poll for message for five seconds instead of the default one second. Also, the idleTaskExecutionLimit property is set to 10 to allow tasks to execute 10 times instead of the default value of 1. Lastly, the idleConsumerLimit property specifies the limit on the number of idle consumers. This property can be used to more aggressively ramp up the number of concurrent consumers.   
  
In addition to tuning these various properties for dynamic consumer scaling, it is also important to understand that the DMLC can also provide various levels of caching for JMS resources (i.e., JMS connections, sessions and consumers). By default, the DMLC will cache all JMS resources unless an external transaction manager is configured (because some containers require fresh JMS resources for external transactions). When an external transaction manager is configured, none of the JMS resources are cached by defualt. The level of caching can be configured using the [cacheLevel](http://static.springsource.org/spring/docs/2.5.x/api/org/springframework/jms/listener/DefaultMessageListenerContainer.html" \l "setCacheLevel(int)) property. This property allows for a tiered caching from connection, to session, to consumer. This allows caching of: 

* The connection
* The connection and the session
* The connection, the session and the consumer

Below is an example configuration that uses the cacheLevel property to specify consumer level caching: 

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:jms="http://www.springframework.org/schema/jms"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/jms http://www.springframework.org/schema/jms/spring-jms-3.0.xsd">

<!-- A JMS connection factory for ActiveMQ -->

<bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory"

p:brokerURL="tcp://foo.example.com:61616" />

<!-- A POJO that implements the JMS message listener -->

<bean id="simpleMessageListener" class="com.mycompany.SimpleMessageListener">

<!-- A JMS namespace aware Spring configuration for the message listener container -->

<jms:listener-container

container-type="default"

connection-factory="connectionFactory"

acknowledge="auto"

concurrency="10-50"

cache="consumer">

<jms:listener destination="TEST.FOO" ref="simpleMessageListener" method="onMessage" />

</jms:listener-container>

</beans>

By caching at the consumer level, this means that the connection, the session and the consumer is cached. Notice that the cacheLevel property can be used with the Spring JMS namespace style of XML configuration.   
  
The session is cached based on ack-mode and the consumer is cached based on the session, the selector and the destination. It's necessary to know this info to better understand where/when the JMS resources can and cannot be reused. For example, caching consumers that use different selectors and consume from different destinations is only going to be relevant if you partition these items appropriately for reuse. That is, you may need to use a separate connection and listener-container configuration if the cache keys are different and if you want to cache sessions or consumers for reuse.   
  
The overall point of the caching is that it can help to reduce the potential recurring thrash involved in creation and destruction of JMS resources. Reducing the thrash by using caching and employing the appropriate partitioning of these resources so as to allow for reuse can definitely improve the overall performance of the application.   
  
Hopefully this post helps you understand how to tune JMS message consumption in Spring. As you employ Spring JMS to your applications and experiment further and further, you will discover how much the DMLC is actually doing for you and how many more features it has beyond what you can easily build yourself.

[inShare](javascript:void(0);)

[Add To DZone](javascript:location.href='http://www.dzone.com/links/add.html?description='+encodeURIComponent(document.getSelection())+'&url='+encodeURIComponent(location.href)+'&title='+encodeURIComponent(document.title))

Bruce Snyder's complete blog can be found at:[*http://bruceblog.org/*](http://bruceblog.org/)