**Gestione ConnectionFactory:**

A ConnectionFactory is one of the JMS administered objects which are preconfigured by an administrator. A client with the help of the configuration will make the connection with a JMS provider.

Spring provides 2 types of ConnectionFactory:

SingleConnectionFactory – is an implementation of ConnectionFactory interface, that will return the same connection on all createConnection() calls and ignore calls to close()

CachingConnectionFactory – extends the functionality of the SingleConnectionFactory and adds enhances it with a caching of Sessions, MessageProducers, and MessageConsumers.

**Destination Management**

As discussed above, along with the ConnectionFactory , destinations are also JMS administered objects and can be stored and retrieved from a JNDI.

Spring provides generic resolvers like DynamicDestinationResolver and specific resolvers such as JndiDestinationResolver.

The JmsTemplate will delegate the resolution of the destination name to one of the implementations basing on our selection.

It will also provide a property called defaultDestination – which will be used with send and receive operations that do not refer to a specific destination.

**Message Conversion**

Spring JMS would be incomplete without the support of Message Converters.

The default conversion strategy used by *JmsTemplate* for both *ConvertAndSend()* and *ReceiveAndConvert()* operations is the *SimpleMessageConverter* class.

*The SimpleMessageConverter* is able to handle  *TextMessages*, *BytesMessages*, *MapMessages*, and *ObjectMessages*. This class implements the *MessageConverter*interface.

Apart from *SimpleMessageConverter*, Spring JMS provides some other *MessageConverter* classes out of the box like *MappingJackson2MessageConverter*, *MarshallingMessageConverter*, *MessagingMessageConverter*.

Moreover, we can create custom message conversion functionality simply by implementing the *MessageConverter* interface’s *toMessage()* and *FromMessage()* methods.

Let us see a sample code snippet on implementing a custom *MessageConverter*,

|  |  |
| --- | --- |
|  | public class SampleMessageConverter implements MessageConverter {      public Object fromMessage(Message message)        throws  JMSException, MessageConversionException {          //...      }        public Message toMessage(Object object, Session session)        throws  JMSException, MessageConversionException {          //...      }  } |

**Configuration with Annotations**

*@JmsListener* is the only annotation required to convert a method of a normal bean into a JMS listener endpoint. Spring JMS provides many more annotations to ease the JMS implementation. We can see some of the sample classes annotated classes below,

|  |  |
| --- | --- |
|  | @JmsListener(destination = "myDestination")  public void SampleJmsListenerMethod(Message<Order> order) { ... } |

In order to add multiple listeners to a single method we just need to add multiple *@JmsListener* annotation.

*@EnableJms* is the annotation added to one of our configuration classes to support the above discussed *@JmsListener* annotated methods.

|  |
| --- |
| @Configuration  @EnableJms  public class AppConfig {        @Bean      public DefaultJmsListenerContainerFactory jmsListenerContainerFactory() {          DefaultJmsListenerContainerFactory factory            = new DefaultJmsListenerContainerFactory();          factory.setConnectionFactory(connectionFactory());          return factory;      }  } |

**Spring Message Driven Pojo:**

1. implementare Listener javax.jms.MessageListener per gestire le operazioni da eseguire alla ricezione di un messaggio
2. Configurare tramite Spring un **messageListenerContainer** e passargli la classe precedente:

<bean id="jmsContainer" class="org.springframework.jms.listener.SimpleMessageListenerContainer">

<property name="connectionFactory" ref="connectionFactory"/>

<property name="destination" ref="destination"/>

**<property name="messageListener" ref="messageListener" />**

</bean>

Quali vantaggi?

Non e’ necessario scrivere tutto il codice boilerplate di costruzione del consumer, collegamento del consumer al listener ed avvio.

Tale lavoro viene eseguito dalla classe di Spring **SimpleMessageListenerContainer**.

(che non supporta però transazioni).

Transazioni  
  
Se e’ necessario gestire la ricezione all’interno di una transazione allora è necessario utilizzare **DefaultMessageListenerContainer**.

Le operazioni per porre in ascolto il consumer saranno automatizzate dalla classe di supporto di Spring (configurabili eventualmente tramite Ioc) per cui all’acquisizione del contesto viene direttamente avviato l’ascolto sulla Destination fornita.

**The Spring DMLC provides many features including:**

* Various levels of caching of the JMS resources (connections and sessions) and JMS consumers for increased performance
* The ability to dynamically grow and shrink the number of consumers to concurrently process messages based on load (see setConcurrentConsumers and setMaxConcurrentConsumers) for additional performance
* Automatically re-establishes connections if the message broker becomes unavailable
* Asynchronous execution of a message listener using the Spring TaskExecutor
* Support for local JMS transactions as well as an external transaction manager around message reception and listener execution
* Support for various message acknowledgement modes, each providing different semantics

**SessionAwareMessageListener:**

Aggiunge al metodo onMessage la ricezione della Session che ha inviato il messaggio che si intende gestire.

package org.springframework.jms.listener;

public interface SessionAwareMessageListener {

void onMessage(Message message, Session session) **throws JMSException**;

}

Si collega come la precedente ad un MessageListenerContainer.

Quando puo’ essere utile:

if you want your MDPs to be able to respond to any received messages (using the Session supplied in theonMessage(Message, Session) method).

**Spring offre le sue implementazioni di entrambe**

All of the message listener container implementations that ship wth Spring have support for MDPs that implement either the MessageListener or SessionAwareMessageListener interface.

**Usare implementazioni di questa classe lega l’applicazione a Spring**

Classes that implement the SessionAwareMessageListener come with the caveat that they are then tied to Spring through the interface. The choice of whether or not to use it is left entirely up to you as an application developer or architect.

**JmsTemplate api:**Astrazione che implementa il codice boilerplate per api jms.

Producer api:

Overload di send() e sendAndConvert()

Consumer api

Overload di receive() e receiveAndConvert()

Lower level api:

.execute (StrategyInterfaceImpl)

**Attenzione!!:**

The thing to remember is JmsTemplate is designed for use in EJBs using the EJB containers JMS pooling abstraction. So every method will typically create a connection, session, producer or consumer, do something, then close them all down again. The idea being that this will use the J2EE containers pooling mechanism to pool the JMS resources under the covers.

Design JmsTemplate presume l esecuzione in un ambiente JEE dove la creazione di Connection e di Session e’ delegate al container JCA.

Per questo motivo in ambienti JSE e’ necessario implementare una soluzione per la cache di Connection e delle Session (pool).

Tale soluzione in Spring viene implementata con la CachingConnectionFactory.

(Vedi dettaglio JmsTemplate)

**Jms Template without caching:**

1) Primary, the overhead of Spring JMS is the use of JmsTemplate to send messages wihtout a caching mechanism underneath. Essentially, JmsTemplate will do the following for each message you send:

* Create Connection
* Create Session
* Create Producer
* Create Message
* Send Message
* Close Session
* Close connection

This of could be compared to manually written code where you reuse things:

* Create Connection
* Create Session
* Create Producer
* Create Message
* Send Message
* Create Message
* Send Message
* Create Message
* Send Message
* Close Session
* Close connection

Since the creation of connections, sessions and producers needs communication between your client and the JMS provider and, of course, resource allocation, it will create pretty large overhead for lots of small messages.

You can easily come around this by caching JMS resources. For instance use the spring [CachingConnectionFactory](http://static.springsource.org/spring/docs/2.5.x/api/org/springframework/jms/connection/CachingConnectionFactory.html) or ActiveMQs [PooledConnectionFactory](http://activemq.apache.org/maven/5.5.0/activemq-pool/apidocs/org/apache/activemq/pool/PooledConnectionFactory.html) (if you are using ActiveMQ, which you tagged this question with).

If you are running inside a full JavaEE container, pooling/caching is often built in and implicit when you retrieve your JNDI connection factory.