Using the JMSConfigFeature

Standard JMS transport configuration in CXF is done by defining a JMSConduit or JMSDestination. There is however an easier configuration option more conformant to Spring dependency injection. Additionally the new configuration offers many more options. For example it is not necessary anymore to use JNDI to resolve the connection factory. Instead it can be defined in the Spring configuration.

The following example configs use the p-namespace from spring 2.5 but the old spring bean style is also possible.

Inside a features element the JMSConfigFeature can be defined.

```
<jaxws:client id="CustomerService"
        xmlns:customer="http://customerservice.example.com/"
        serviceName="customer:CustomerServiceService"
        endpointName="customer:CustomerServiceEndpoint" address="jms://"
        serviceClass="com.example.customerservice.CustomerService">
        <jaxws:features>
                <bean xmlns="http://www.springframework.org/schema/beans"
                        class="org.apache.cxf.transport.jms.JMSConfigFeature"
                        p:jmsConfig-ref="jmsConfig"/>
        </jaxws:features>
</jaxws:client>
```

In the above example it references a bean "jmsConfig" where the whole configuration for the JMS transport can be done.

A jaxws Endpoint can be defined in the same way:

```
<jaxws:endpoint
        xmlns:customer="http://customerservice.example.com/"
        id="CustomerService"
        address="jms://"
        serviceName="customer:CustomerServiceService"
        endpointName="customer:CustomerServiceEndpoint"
        implementor="com.example.customerservice.impl.CustomerServiceImpl">
        <jaxws:features>
                <bean class="org.apache.cxf.transport.jms.JMSConfigFeature"
                        p:jmsConfig-ref="jmsConfig" />
        </jaxws:features>
</jaxws:endpoint>
```

The JMSConfiguration bean needs at least a reference to a connection factory and a target destination.

```
<bean id="jmsConfig" class="org.apache.cxf.transport.jms.JMSConfiguration"
        p:connectionFactory-ref="jmsConnectionFactory"
        p:targetDestination="test.cxf.jmstransport.queue"
/>
```

If your ConnectionFactory does not cache connections you should wrap it in a spring SingleConnectionFactory. This is necessary because the JMS Transport creates a new connection for each message and the SingleConnectionFactory is needed to cache this connection.

```
<bean id="jmsConnectionFactory" class="org.springframework.jms.connection.SingleConnectionFactory">
        <property name="targetConnectionFactory">
                <bean class="org.apache.activemq.ActiveMQConnectionFactory">
                        <property name="brokerURL" value="tcp://localhost:61616" />
                </bean>
        </property>
</bean>
```

Using JMSConfiguration from Java

To do this from Java, you need to initialize a JMSConfiguration object, then store a reference to it in a JMSConfigFeature, and then add that to the features in the server factory. The code that follows is fragmentary. Note that you can't use query parameters in the endpoint URI that you set in the server factory, all the configuration has to be in the JMSConfiguration object.

```
public static JMSConfiguration newJMSConfiguration(String taskId, String jmsBrokerUrl) {
        String destinationUri = "jms:queue:" + taskId;
        JMSConfiguration conf = new JMSConfiguration();
        conf.setRequestURI(destinationUri);
        JNDIConfiguration jndiConfig = new JNDIConfiguration();
        JndiTemplate jt = new JndiTemplate();
        Properties env = new Properties();
        env.put(Context.PROVIDER_URL, jmsBrokerUrl);
        env.put(Context.INITIAL_CONTEXT_FACTORY, "org.apache.activemq.jndi.ActiveMQInitialContextFactory");
        jt.setEnvironment(env);
        jndiConfig.setJndiConnectionFactoryName("ConnectionFactory");
        jndiConfig.setEnvironment(env);
        conf.setJndiTemplate(jt);
        conf.setTargetDestination("com.basistech.jug." + taskId);
        conf.setJndiConfig(jndiConfig);
        conf.setTimeToLive(0);
        return conf;
}
{
        JMSConfigFeature jmsConfigFeature = new JMSConfigFeature();
        JMSConfiguration jmsConfig = JmsUtils.newJMSConfiguration(taskId, jmsBrokerUrl);
        jmsConfig.setConcurrentConsumers(maxServiceThreads);
        jmsConfig.setMaxConcurrentConsumers(maxServiceThreads);
        jmsConfigFeature.setJmsConfig(jmsConfig);
        svrFactory.getFeatures().add(jmsConfigFeature);
        svrFactory.getFeatures().add(jmsConfigFeature);

        server = svrFactory.create();
}
```

JMSConfiguration options

Name	Description
connectionFactory	Mandatory field. Reference to a bean that defines a jms ConnectionFactory. Remember to wrap the connectionFactory like described above when not using a pooling ConnectionFactory
wrapInSingleConnectionFactory	This option was removed since CXF 3.0.0. Will wrap the connectionFactory with a Spring SingleConnectionFactory, which can improve the performance of the jms transport. Default is true.
reconnectOnException	(**deprecated**) If wrapping the connectionFactory with a Spring SingleConnectionFactory and reconnectOnException is true, will create a new connection if there is an exception thrown, otherwise will not try to reconnect if the there is an exception thrown. Default is false. From CXF 3.0.0, CXF always reconnect on exceptions
targetDestination	JNDI name or provider specific name of a destination. Example for ActiveMQ: test.cxf.jmstransport.queue
destinationResolver	Reference to a Spring DestinationResolver. This allows to define how destination names are resolved to jms Destinations. By default a DynamicDestinationResolver is used. It resolves destinations using the jms providers features. If you reference a JndiDestinationResolver you can resolve the destination names using JNDI.
transactionManager	Reference to a spring transaction manager. This allows to take part in JTA Transactions with your webservice. You can also register a spring JMS Transaction Manager to have local transactions.
taskExecutor	This option was removed since CXF 3.0.0. Reference to a spring TaskExecutor. This is used in listeners to decide how to handle incoming messages. Default is a spring SimpleAsyncTaskExecutor.
useJms11	This option was removed since CXF 3.0.0. true means JMS 1.1 features are used false means only JMS 1.0.2 features are used. Default is false.
messageIdEnabled	Default is true. This option was removed since CXF 3.0.0.
messageTimestampEnabled	Default is true. This option was removed since CXF 3.0.0.

cacheLevel	This option was removed since CXF 3.0.0. Specify the level of caching that the JMS listener container is allowed to apply. Please check out the java doc of the org.springframework.jms.listenerDefaultMessageListenerContainer for more information. Default is -1.
pubSubNoLocal	If true, do not receive your own messages when using topics. Default is false.
receiveTimeout	How many milliseconds to wait for response messages. 0 (default) means wait indefinitely. since CXF 3.0, the default value is changed to 60000 (60 seconds)
explicitQosEnabled	If true, means that QoS parameters are set for each message. Default is false.
deliveryMode	NON_PERSISTENT = 1 messages will be kept only in memory PERSISTENT = 2 (default) messages will be persisted to disk
priority	Priority for the messages. Default is 4. See your JMS provider doc for details
timeToLive	After this time the message will be discarded by the jms provider (default 0).
sessionTransacted	If true, means JMS transactions are used. (Default is false). In 2.7.x you will also need to register a JMS Transaction Manager with JMSConfiguration in order for transactions to be enabled.
concurrentConsumers	Number of concurrent consumers for listener (default 1).
maxConcurrentConsumers	This option was removed since CXF 3.0.0. Maximum number of concurrent consumers for listener (default 1).
maxConcurrentTasks	This option was removed since CXF 3.0.0. (**deprecated**) Maximum number of threads that handle the received requests (Default 10).
messageSelector	jms selector to filter incoming messages (allows to share a queue)
subscriptionDurable	Default false.
durableSubscriptionName	
messageType	text (default) binary byte
pubSubDomain	false (default) means use queues true means use topics
jmsProviderTibcoEms	true means that the jms provider is Tibco EMS. Default is false. Currently this activates that the principal in the SecurityContext is populated from the header JMS_TIBCO_SENDER. (available from cxf version 2.2.6)
maxSuspendedContinuations	**Since CXF 3.0.0**, The max suspended continuations that the JMS destination could have, if the current suspended continuations number exceeds the max value, the JMSListenerContainer will be stopped. The default value is -1, which means disable this feature.
reconnectPercentOfMax	**Since CXF 3.0.0,** If the JMSListenerContainer is stopped due to the current suspended continuation exceeds the max value, the JMSListenerContainer will be restarted when the current suspended continuation below the value of (maxSuspendedContinuations*reconnectPercentOfMax/100). The default value is 70.
createSecurityContext	**(Since 2.7.14, 3.0.3)** true (default) means create user security context for incoming messages.
propogateExceptions	**(Since 2.7.15) 2.7.x only** true (default) means that any exceptions occurring while processing the incoming message will be propagated. This setting is only relevant when a transaction manager and sessionTransacted are set.