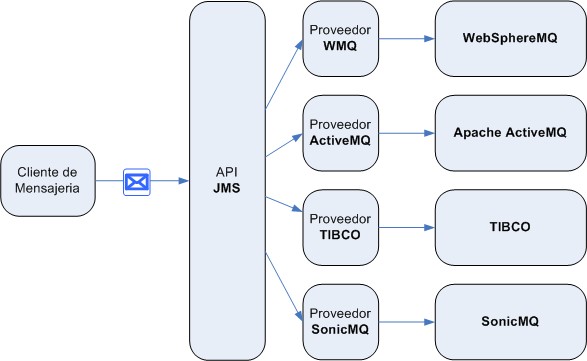
**INVESTIGACIÓN SERVIDOR DE COLAS**

**JMS**

**Java Message Service**

JMS se separa de las APIs propietarias de cada proveedor para ofrece un API estándar (mediante un conjunto de interfaces) para la mensajería empresarial, de modo que mediante Java podamos enviar y recibir mensajes sin atarnos a ningún proveedor. JMS además minimiza el conocimiento de mensajería empresarial que debe tener un programador Java para desarrollar complejas aplicaciones de mensajería, mientras mantiene la portabilidad entre las diferentes implementaciones de proveedores JMS.

Cuidado, JMS no es un MOM. Se trata de una especificación que abstrae la interacción entre los clientes de mensajería y los MOMs del mismo modo que JDBC abstrae la comunicación con las BBDD relacionales. El siguiente gráfico muestra como JMS ofrece un API que utilizan los clientes de mensajería para interactuar con MOMs específicos via proveedores JMS que manejan la interacción con el MOM específico. De este modo, JMS reduce la barrera para la creación de aplicaciones de mensajería, facilitando la portabilidad a otros proveedores JMS.



JMS permite que la comunicación entre componentes sea **débilmente acoplada**, **asíncrona** (el proveedor JMS entrega los mensajes al destino conforme llegan, y el cliente no tiene que solicitar los mensajes para recibirlos) y **fiable** (JMS asegura que cada mensaje se entrega una y solo una vez, y mediante inferiores niveles de fiabilidad permite la pérdida o el duplicado de mensajes en aquellas aplicaciones que requieran menos control).

Originalmente creada por *Sun* conjunto a un grupo de compañías de la industria de la mensajería empresarial, la primera versión de la especificación JMS data de 1998. La última *release* fue en 2002 con mejoras necesarias y desde entonces se trata de una tecnología estable y madura. La *release* JMS 1.1 unificó los dos conjuntos de APIs para trabajar con los dos dominios de mensajería, de modo que ahora sólo necesitamos una API para trabajar con ambos dominios.

**JMS y JavaEE**

A partir de la versión 1.3 de JavaEE, el API JMS forma parte de la especificación *enterprise*, y los desarrolladores la pueden utilizar dentro de componentes JavaEE.

El API JMS dentro de la plataforma JavaEE aporta:

* Aplicaciones cliente, componentes EJB y componentes web que pueden enviar o recibir de forma asíncrona mensajes JMS. Los clientes pueden recibir incluso los mensajes también de forma asíncrona.
* Beans de mensajes (*Message-driven Beans* - MDBs) que son un tipo de EJB que puede consumir mensajes de forma asíncrona. Un proveedor JMS (típicamente el servidor de aplicaciones) puede implementar de forma opcional el procesamiento concurrente de mensajes con MDBs.
* El envío y recepción de mensajes puede participar en transacciones distribuidas lo cual permite que las operaciones JMS y el acceso a BD ocurra dentro de una misma transacción.

Todo esto provoca que JMS mejore JavaEE simplificando el desarrollo de aplicaciones empresariales, mediante interacciones asíncronas, de confianza y con bajo acoplamiento entre los componentes JavaEE y los sistemas legacy que soportan la mensajería. A su vez, JavaEE mejora JMS mediante el soporte de transacciones distribuidas, y el procesamiento de mensajes de manera concurrente.

Arquitectura de Conectores JavaEE

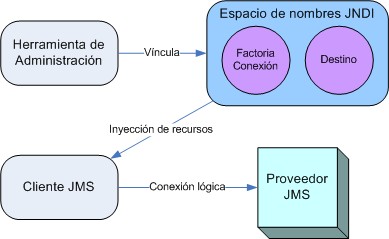
El servidor de aplicaciones integra a diferentes proveedores JMS mediante la arquitectura de conectores. De este modo, accedemos a un proveedor a través de un adaptador de recursos. Esto permite que los propietarios de MOMs creen proveedores JMS que puedan enchufarse en múltiples servidores de aplicaciones, y permite a un servidor de aplicaciones disponer de múltiples proveedores JMS.

**Arquitectura JMS**

Para estandarizar el API, JMS define de un modo formal muchos conceptos y elementos del mundo de la mensajería:

* *Cliente JMS*: Una aplicación 100% Java que envía y recibe mensajes. Cualquier componente JavaEE puede actuar como un cliente JMS.
  + *Clientes No-JMS*: una aplicación escrita en un lenguaje que no es Java que envía y recibe mensajes.
  + *Productor JMS*: una aplicación cliente que crea y envía mensajes JMS.
  + *Consumidor JMS*: una aplicación cliente que recibe y procesa mensajes JMS.
* *Proveedor JMS*: implementación de los interfaces JMS el cual está idealmente escrito 100% en Java. El proveedor debe ofrecer prestaciones tanto de administración como de control de los recursos JMS. Toda implementación de la plataforma Java incluye un proveedor JMS.
* *Mensaje JMS*: elemento principal de JMS; objeto (cabecera + propiedades + cuerpo) que contiene la información y que es enviado y recibido por clientes JMS.
* *Dominio JMS*: Los dos estilos de mensajería: PTP y Pub/Sub.
* *Objetos Administrados*: objetos JMS preconfigurados que contienen datos de configuración específicos del proveedor, los cuales utilizarán los clientes. Los clientes acceden a estos objetos mediante JNDI.
  + *Factoría de Conexión*: los clientes utilizan una factoría para crear conexiones al proveedor JMS.
  + *Destino*: objeto (cola/tópico) al cual se direccionan y envían los mensajes, y desde donde se reciben los mensajes.

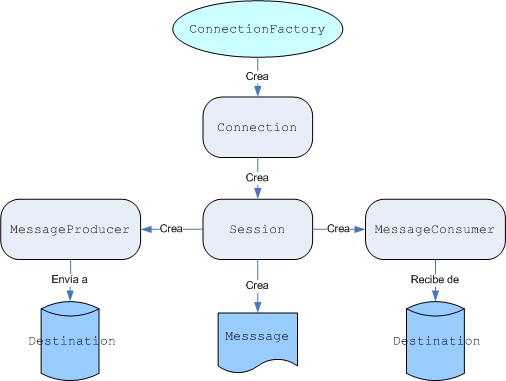
Estos elementos interaccionan del siguiente modo:



Las herramientas de administración permiten vincular destinos y factorías de conexión a través de un espacio de nombres JNDI. Entonces un cliente JMS puede consultar los objetos administrados en dicho espacio vía inyección de recursos y establecer conexiones lógicas con ellos a través del proveedor JMS.

**El Modelo de Programación JMS**

El esquema de trabajo con las **interfaces** JMS queda claramente definido en el siguiente gráfico:



A continuación veremos en detalle cada uno de estos elementos con trozos de código que muestran como instanciarlos.

**Objetos Administrados**

Es más cómodo que los dos extremos de las aplicación JMS, la factoría de conexiones y los destinos, sean mantenidos mediante administración que de forma programativa. Esto se debe a que la tecnología que hay bajo estos objetos va a ser diferente dependiendo del proveedor JMS, y por tanto, su administración varía de un proveedor a otro.

Los clientes JMS acceden a estos objetos vía interfaces que son portables, de modo que un cliente pueda cambiar de implementación JMS sin necesidad de ninguna modificación. La administración de estos objetos (en nuestro caso a través de la consola de administración de *Glassfish*) se realiza dentro de un espacio de nombre JNDI, y los clientes acceden a él mediante la inyección de recursos vía anotaciones.

**Factorías de Conexión**

La factoría de conexión es el objeto que utiliza el cliente para crear una conexión con el proveedor, encapsulando un conjunto de parámetros de configuración de la conexión que han sido previamente definidos por el administrador del servidor de mensajes. Cada factoría de conexión es una instancia de ConnectionFactory, ya sea QueueConnectionFactory o TopicConnectionFactory.

Al inicio de un cliente JMS, normalmente se inyecta un recurso de factoría de conexión en un objeto ConnectionFactory. Por ejemplo, el siguiente fragmento de código muestra como se inyecta el recurso cuyo nombre JNDI es jms/ConnectionFactory y se asigna a un objeto ConnectionFactory:

@Resource(mappedName="jms/ConnectionFactory")

private static ConnectionFactory connectionFactory;

**Destinos**

Un destino (javax.jms.Destination) es el objeto que utiliza el cliente para especificar el destino de los mensajes que produce y el origen de los mensajes que consume. En PTP los destinos son las colas (javax.jms.Queue), mientras que en Pub/Sub son los tópicos (javax.jms.Topic). Una aplicación JMS puede utilizar múltiples colas o tópicos (o ambos).

Para crear un destino mediante el servidor de aplicaciones, hay que crear un recurso JMS que especifique un nombre JNDI para el destino. Dentro de la implementación del servidor de aplicaciones, cada destino referencia a un destino físico.

Del mismo modo que con la factoría de conexiones, los destinos también se inyectan, pero en este caso, son específicos a un dominio u otro. Si quisiéramos crear una aplicación que con el mismo código fuente trabajase tanto con tópicos como con colas deberíamos asignar el destino a un objeto Destination.

El siguiente fragmento específica dos recursos, una cola y un tópico. Los nombres de los recursos se mapean con destinos creados via JNDI.

@Resource(mappedName="jms/Queue")

private static Queue queue;

@Resource(mappedName="jms/Topic")

private static Topic topic;

Mezclando Interfaces

El uso de interfaces comunes permite mezclar factorías de conexiones y destinos. Por poder, se puede crear una QueueConnectionFactory y utilizarla con un Topic, y viceversa.  
**El comportamiento de la aplicación dependerá del tipo de destino**, no del tipo de factoría de conexión.

**Conexiones**

Una conexión encapsula una conexión virtual con el proveedor JMS, y puede representar un socket TCP/IP entre el cliente y un demonio del proveedor. Al crear una conexión, se crean objetos, tanto en la parte del cliente como en la del servidor, que gestionan el trasiego de mensajes entre el cliente y el sistema de mensajes. Mediante una conexión crearemos una o más sesiones en las que se producen y se consumen mensajes. Las conexiones implementan el interfaz javax.jms.Connection. A partir de una ConnectionFactory, podemos crear una conexión del siguiente modo:

Connection connection = connectionFactory.createConnection();

Al finalizar la aplicación, tenemos que cerrar toda conexión. Es muy importante cerrar las conexiones porque sino podemos provocar la sobrecarga del proveedor JMS. Al cerrar una conexión también cerramos sus sesiones y sus productores y consumidores de mensajes.

connection.close();

Antes de que nuestras aplicaciones puedan consumir mensajes, debemos llamar al método start de la conexión. Si queremos parar el envío de mensajes de forma temporal sin cerrar la conexión, podemos utilizar el método stop.

**Sesiones**

Una sesión es un contexto monohilo para producir y consumir mensajes. Mediante las sesiones crearemos:

* Productores de mensajes.
* Consumidores de mensajes.
* Mensajes.
* Navegadores de colas (*Queue Browser*).
* Colas y tópicos temporales.

Existen dos tipos de sesiones: las transaccionales y las no-transaccionales. Las transaccionales se caracterizan porque todos los mensajes enviados y recibidos se tratan como una unidad atómica que está sujeta al protocolo commit/rollback (confirmar o deshacer). En estas sesiones no es necesario realizar acuses de recibo o *acknowledgements*. En las no-transaccionales hay que seleccionar un tipo de acuse de recibo. En este caso, el tipo Session.AUTO\_ACKNOWLEDGE indica que la sesión acusa el recibo de un mensaje una vez que la aplicación receptora lo ha procesado.

Las sesiones implementan el interfaz javax.jms.Session. Tras crear una conexión, la utilizaremos para crear una sesión:

Session session = connection.createSession(false, Session.AUTO\_ACKNOWLEDGE);

El primer parámetro indica que la sesión no es transaccional, y el segundo que la sesión confirmará la recepción exitosa de los mensajes de forma automática mediante un acuse de recibo.

Para crear una sesión transaccional, utilizaremos el siguiente fragmento:

Session session = connection.createSession(true, 0);

En este caso, hemos creado una sesión transaccional pero que no especifica la confirmación de los mensajes.

**Productores de Mensajes**

Un productor de mensajes es un objeto creado por una sesión y que se utiliza para enviar mensajes a un destino. Implementa el interfaz javax.jms.MessageProducer. A partir de la sesión y un destino, podemos crear diferentes tipos de productores:

MessageProducer producer = session.createProducer(dest);

MessageProducer producer = session.createProducer(queue);

MessageProducer producer = session.createProducer(topic);

Una vez creado un productor y el mensaje, para enviar mensajes utilizaremos el método send:

producer.send(message);

Se puede crear un productor sin identificar mediante un null como parámetro en el método createProducer. Mediante este tipo de productores, el destino no se especifica hasta que se envía un mensaje, especificándolo como primer parámetro.

MessageProducer anonProd = session.createProducer(null);

anonProd.send(dest, message);

**Consumidores de Mensajes**

Un consumidor de mensajes es un objeto creado por una sesión y que se utiliza para recibir mensajes enviados desde un destino. Implementa el interfaz javax.jms.MessageConsumer. A partir de la sesión y un destino, podemos crear diferentes tipos de productores:

MessageConsumer consumer = session.createConsumer(dest);

MessageConsumer consumer = session.createConsumer(queue);

MessageConsumer consumer = session.createConsumer(topic);

Un consumidor de mensajes permite a un cliente JMS registrar su interés en un destino con un proveedor JMS. El proveedor gestiona la entrega de mensajes desde un destino a los consumidores registrados en dicho destino.

Tras crear un consumidor, éste queda activo y lo podemos utilizar para recibir mensajes. Para desactivar al consumidor, utilizaremos el método close. La entrega de mensajes no comienza hasta que no se inicia la conexión creada mediante el método start.

Recuerda

Recuerda siempre llamar al método start, es uno de los errores más comunes dentro de la programación JMS

Para consumir un mensaje de forma síncrona utilizaremos el método receive. Esta operación se puede realizar en cualquier momento siempre y cuando previamente hayamos iniciado la conexión (mediante el método start):

connection.start();

Message m = consumer.receive();

connection.start();

Message m = consumer.receive(1000); // timeout tras un segundo

Para consumir un mensaje de forma asíncrona necesitamos un *listener* de mensajes.

**Listener de Mensajes**

Un listener de mensajes es un objeto que actúa como un manejador de eventos asíncronos para mensajes. Este objeto implementa el interfaz javax.jms.MessageListener, el cual únicamente contiene el método onMessage. En este método definiremos las acciones a realizar con el mensaje recibido.

Para registrar el listener utilizaremos el método setMessageListener del interfaz MessageConsumer. Por ejemplo, si tenemos una clase Listener que implementa el interfaz MessageListener, podemos registrar el listener del siguiente modo:

Listener myListener = new Listener();

consumer.setMessageListener(myListener);

Tras registrar el listener, podemos llamar al método start de la conexión para empezar la entrega de mensajes. Si lo hacemos antes, perderemos mensajes. Cuando comienza el envío de los mensajes, cada vez que se recibe un mensaje, el proveedor JMS llama al método onMessage del listener de mensajes. El método onMessage recibe como parámetro un objeto de tipo Message, con los datos recibidos. Nuestro método onMessage debería capturar todas las excepciones. No debe lanzar excepciones *checked*, y relanzar excepciones *unchecked* (RuntimeException) se considera un error de programación.

Un listener de mensajes no es específico para un tipo de destino en particular. El mismo listener puede obtener mensajes tanto de una cola como de un tópico, dependiendo del tipo de destino para el cual se creó el consumidor de mensajes. Sin embargo, un listener normalmente espera un tipo y formato específico de mensaje.

La sesión utilizada para crear el consumidor de mensajes serializa la ejecución de todos los listener de mensajes registrados con la sesión. En un instante cualquiera, uno y solo uno de los listener de mensajes de la sesión está en ejecución.

**Mensajes**

Los mensajes también se crean a partir de objetos de sesión. Por ejemplo, para crear un mensaje de tipo texto:

TextMessage message = session.createTextMessage();

Los mensajes encapsulan información a intercambiar entre aplicaciones. Un mensaje contiene tres componentes: los campos de la *cabecera*, las *propiedades* específicas de la aplicación y el *cuerpo* del mensaje.

Los partes de cada mensaje así como los diferentes tipos de mensajes los estudiaremos en la siguiente sesión.

**Una Aplicación JMS**

Una aplicación JMS será tan simple o compleja como sean sus requisitos de negocio. Igual que con JDBC, es común aislar el código JMS mediante componentes o en su propia capa.

Los pasos que seguirá todo componente JMS serán:

1. Adquirir una factoría de conexión.
2. Crear una conexión mediante la factoría de conexión.
3. Comenzar la conexión.
4. Crear una sesión a partir de la conexión.
5. Adquirir un destino.
6. Dependiendo de si enviamos o recibimos
   * Crear un productor.
     1. Crear un productor.
     2. Crear un mensaje y adjuntarlo a su destino.
   * Crear un consumidor.
     1. Crear un consumidor.
     2. Opcionalmente registrar un listener de mensajes
7. Enviar/Recibir el/los mensaje/s
8. Cerrar los objetos (consumidor, productor, sesión, conexión)