Servicios de Mensajes con JMS

Índice

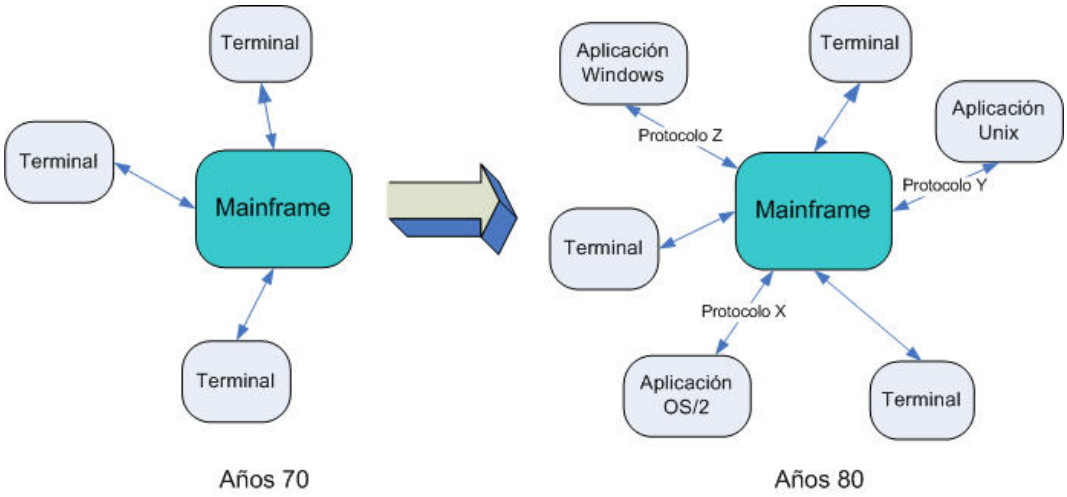
1. Introducción a JMS (Java Message Service)

Tarde o temprano llega el momento en el que todo desarrollador tiene la necesidad de intercomunicar aplicaciones. Existen múltiples soluciones para este tipo de problemas, pero dependiendo de las restricciones y requerimientos, el hecho de decidir como resolver esta situación puede ser otro problema en sí. Los requerimientos de negocio normalmente restringen el número de elementos que tienen un impacto directo como puede ser el rendimiento, escalabilidad, confiabilidad, etc...

1.1. Un Poco de Historia

A partir de los años 60, las grandes empresas invirtieron grandes cantidad de dinero en *mainframes* para las aplicaciones críticas, tales como procesamiento de datos, procesos financieros, análisis estadísticos, etc... La arquitectura *mainframe* ofrece muchos beneficios, como pueden ser la alta disponibilidad, redundancia, gran confiabilidad y escalabilidad, etc... Aunque estos sistemas eran extremadamente potentes (y caros), el acceso a estos sistemas se restringe a unas pocas opciones de entrada. Además, la interconectividad entre estos sistemas todavía no existe, lo que imposibilidad el procesamiento paralelo.

En los 70, los usuarios empezaron a acceder a los mainframes a través de terminales los cuales expandieron el uso de estos sistemas permitiendo el acceso concurrente de miles de usuarios. Durante esta época, se inventaron las redes de ordenadores, y se hizo posible la conectividad entre mainframes. En los 80, además de los terminales gráficos disponibles, los PCs llegaron al mercado y la emulación de terminales se convirtió en algo común. La interconectividad se convirtió en un aspecto muy importante ya que las aplicaciones desarrolladas para ejecutarse sobre PCs necesitaban acceder a los mainframes. Esta diversidad de plataformas y protocolos introdujo nuevos problemas que había que resolver.



Evolución de la arquitectura Mainframe

La conexión de un sistema fuente con un destino no era nada sencillo ya que cada hardware, protocolo y formato de datos requería un tipo diferente de adaptador. Conforme creció esta lista de adaptadores, también lo hizo las diferentes versiones de estos, lo que provoco que se convirtiera en difícil de mantener, hasta que el mantenimiento de los adaptadores llevaba más tiempo que el de los propios sistemas. Esto dio pie a la mensajería empresarial.

El objetivo de la **mensajería empresarial** era transferir información entre sistemas heterogéneos mediante el envío de mensajes de un sistema a otro. Ha habido diversas tecnologías con diferentes formas de mensajería a lo largo de los años, incluyendo:

- Soluciones para llamadas a procedimientos remotos que hacen funciones de middleware mediante una cola de mensajes, tales como *COM* y *CORBA*
- Soluciones para la notificación de eventos, comunicación entre procesos y colas de mensajes los cuales se incluyen en los sistemas operativos, como buffers FIFO, colas de mensajes, tubos (*pipes*), señales, sockets, ...
- Soluciones para una categoría de middleware que ofrece un mecanismo de mensajería fiable y asíncrono tales como *WebShpereMQ*, *SonicMQ*, *TIBCO*, *Apache ActiveMQ*, etc...

Así pues, existen muchos productos que ofrecen un mecanismo de mensajería, pero la que nos interesa a nosotros es la última categoría. La necesidad es la madre de la invención, y por eso apareció el middleware de mensajería. Se necesitaba un software para comunicar y transferir datos entre diferentes aplicaciones, sistemas operativos, protocolos e incluso diferentes lenguajes de programación. Además, el enrutamiento y la transformación de mensajes emergían como una parte importante de estas soluciones. Esto es lo que hoy día se conoce como **MOM** (*Message-Oriente Middleware*)

1.1.1. Mensajería

La mensajería es un método de comunicación entre componentes software o aplicaciones. Un sistema de mensajes es una facilidad *peer-to-peer*: un cliente de mensajería puede enviar mensajes a, o recibir mensajes de, otro cliente. Basta con que ambos se conecten a agentes de mensajería (*MOMs*) que proporcionen facilidades de creación, envío, recepción y lectura de mensajes. Los mensajes permiten una comunicación distribuída **débilmente acoplada**: un componente envía un mensaje a un *destino* y el receptor lo recoge del mismo.

Sin embargo, ni el emisor ni el receptor tienen que estar disponibles al mismo tiempo para comunicarse. De hecho, el emisor no tiene porqué saber nada del receptor, y viceversa. Ambos sólo deben saber el formato del mensaje y cual es el destino del mensaje. Esto hace que la mensajería difiera de tecnologías fuertemente acopladas, tales como RMI, que requieren que la aplicación cliente conozca los métodos de la aplicación remota.

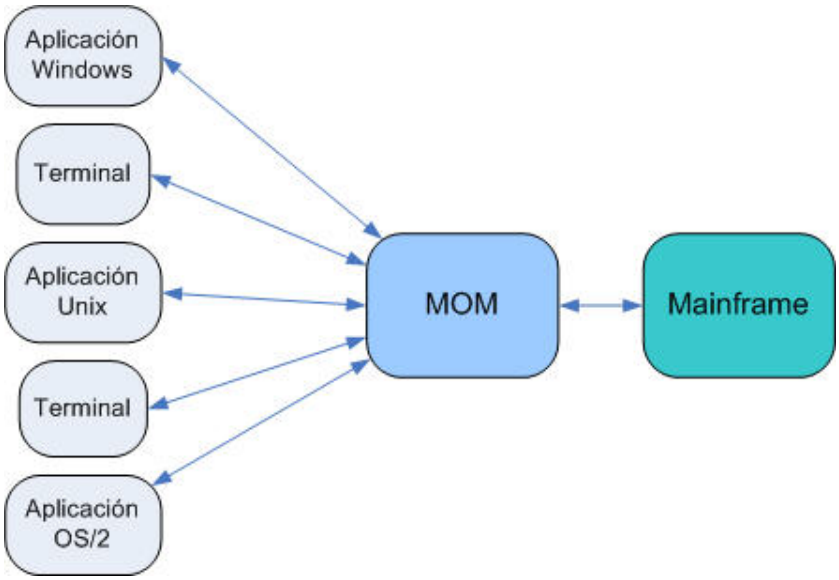
Una aplicación utilizará la mensajería cuando queramos que:

- ciertos componentes no dependan del interfaz de otros componentes, de modo que éstos puedas sustituirse fácilmente
- la aplicación funcione independientemente de si todos los componentes se están ejecutando de forma simultánea
- el modelo de negocio de la aplicación permita a un componente enviar información a otro componente y que continúe su procesamiento sin esperar a una respuesta inmediata.

1.1.2. MOM

El **M**idleware **O**rientado a **M**ensajes (MOM) es un categoría de software para la intercomunicación de sistemas que ofrece una manera segura, escalable, confiable y con bajo acoplamiento. Los MOMs permiten la comunicación entre aplicaciones mediante un conjunto de APIs ofrecidas por cada proveedor y lenguaje, así pues, tendremos un API propietaria y diferente por cada MOM existente.

La idea principal de un MOM es que actúa como un mediador entre los emisores y los receptores de mensajes. Esta mediación ofrece un nuevo nivel de desacoplamiento en la mensajería empresarial. Así pues, un MOM se utiliza para mediar en la conectividad y la mensajería, no solo entre las aplicaciones y el mainframe, sino de una aplicación a otra.



Evolución MOM

A un nivel más alto, los mensajes son unidades de información de negocio que se envían de una aplicación a otra a través de un MOM. Estos mensajes se envían y reciben por aquellos clientes que se conectan o subscriben a los mensajes. Este mecanismo es el que permite el acoplamiento débil entre emisores y receptores, ya que no se requiere que ambos estén conectados simultáneamente al MOM para enviar y/o recibir los mensajes. Esto es la **mensajería asíncrona**.

Los MOMs añadieron muchas características a la mensajería empresarial que previamente no eran posibles cuando los sistemas estaban fuertemente acoplados, tales como la persistencia de los mensajes, enrutamientos complejos de mensajes, transformación de los mensajes, etc... La *persistencia de mensajes* ayuda a mitigar las conexiones lentas o poco fiables realizadas por lo emisores y receptores o en una situación donde el fallo de un receptor no afecta al estado del emisor. El *enrutamiento complejo* de mensajes genera una cantidad de posibilidades que incluyen la entrega de un único mensaje a muchos receptores, enrutamiento de mensajes basados en propiedades del contenido del mensaje, etc... La *transformación de mensajes* permite la comunicación entre dos aplicaciones que no trabajan con el mismo formato de mensajes.

Además, la mayoría de MOMs existentes en el mercado ofrecen soporte para diversos protocolos de conectividad, como pueden ser HTTP/S, SSL, TCP/IP, UDP, etc... Incluso algunos proveedores ofrecen soporte para múltiples lenguajes de programación, facilitando del uso de MOMs en una amplia variedad de entornos. Este gran abanico de protocolos, lenguajes y APIs provoca la aparición de JMS para estandarizar la mensajería dentro del mundo Java.

1.2. Dominios de Mensajería

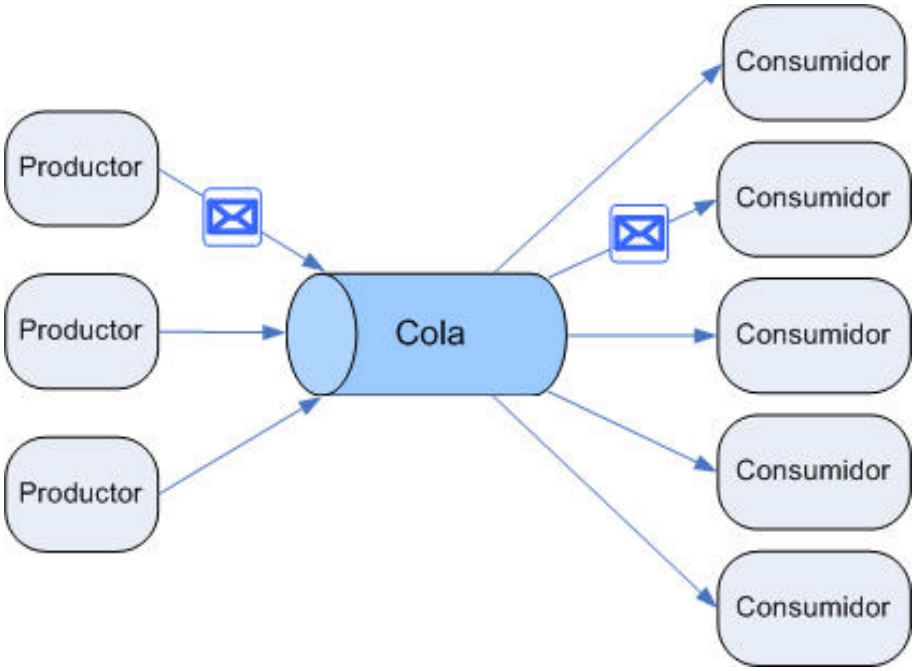
Existen dos modelos/dominios de mensajería:

- Punto a Punto (**PTP**), en la que un mensaje se consume por un único consumidor.
- Publicación/Subscripción (**Pub/Sub**), en la que un mensaje se consume por muchos consumidores.

Destacar que no todos los proveedores implementan ambos. Posteriormente veremos como JMS ofrece diferentes APIs para estos dos modelos.

1.2.1. Punto a Punto

Como hemos mencionado, bajo el modelo PTP, **un mensaje se consume por un único consumidor** (1:1), pero pueden haber varios emisores. El destino del mensaje es un **cola** definida y con un nombre (de manera opuesta a un tópico bajo el modelo Pub/Sub). Dicho de otra manera, se trata de un modelo FIFO, en el cual el mensaje encolado será el primero en salir de la cola (suponiendo que tienen el mismo nivel de prioridad).



Modelo Punto a Punto

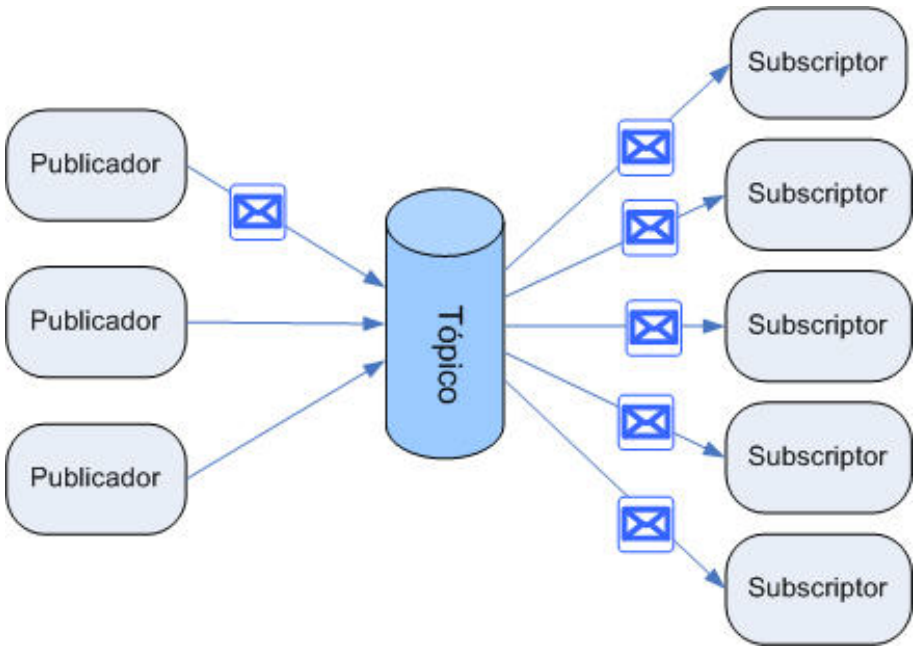
Así pues, bajo el modelo punto a punto, el emisor envía un mensaje a una cola definida (con nombre) con un nivel de prioridad, y el receptor extrae el mensaje de la cola. Al extraer el mensaje, el receptor envía un acuse de recibo a la cola para confirmar su correcta recepción (ACK).

Una aplicación PTP se construye bajo el concepto de colas de mensajes, productores y consumidores. A los emisores se les conoce como **productores**, y a los receptores como **consumidores**. Cada mensaje se envía a una cola específica, y los consumidores extraen

los mensajes de la(s) cola(s) definidas. Estas colas retienen todos los mensajes enviados hasta que son consumidos o hasta que expiren.

1.2.2. Publicación/Subscripción

En este modelo, **un mensaje puede consumirse por múltiples consumidores** (1:N). El destino de un mensaje se conoce como **tópico**. Un tópico no funciona como un pila, ya que los mensajes en los tópicos no se encolan. De hecho, un nuevo mensaje en el tópico sobrescribirá cualquier mensaje existente. Así pues, bajo este modelo de difusión, los productores/emisores/publicadores publican el mensaje en un tópico, y los consumidores se subscriben al tópico.



Modelo Publicación/Subscripcuión

En este modelo, los publicadores (emisores) y los subscriptores (receptores) normalmente son anónimos y pueden, de forma dinámica, publicar o subscribirse a la jerarquía de contenidos. El sistema de mensajería se encarga de distribuir los mensajes que llegan al tópico de los múltiples publicadores a sus respectivos subscriptores, mediante un mecanismo push, de modo que los mensajes se envían automáticamente a los subscriptores.

La mensaria Pub/Sub tiene las siguientes características:

- Cada mensaje puede tener múltiples consumidores
- Existe un dependencia temporal entre los publicadores y los subscritores perecederos (*non-durable*) ya que un cliente que se subscribe a un tópico puede consumir los mensajes publicados después de la subscripción, y el subscriptor debe continuar conectado para consumir los posteriores mensajes.

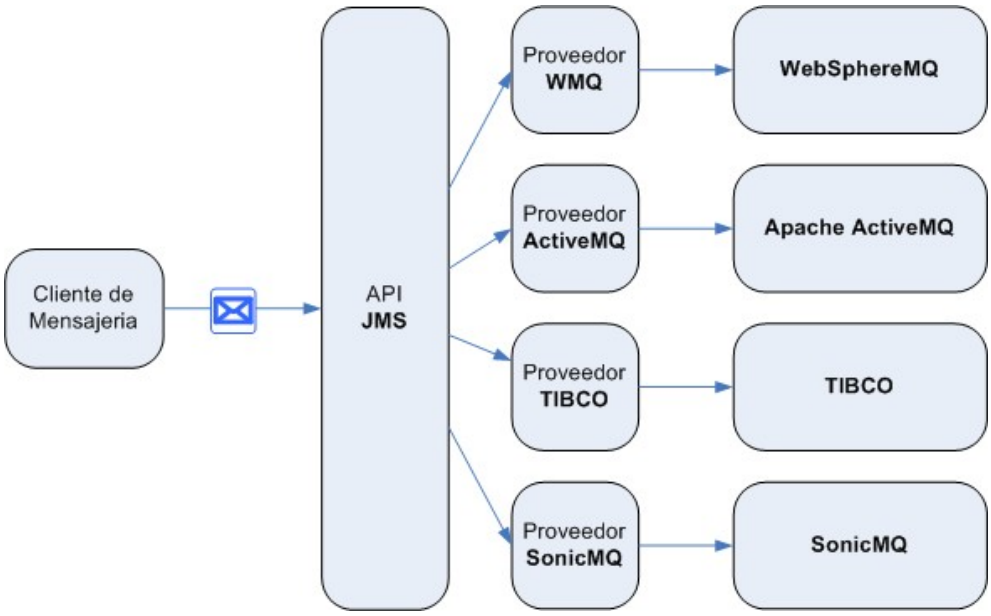
El API JMS disminuye esta dependencia temporal permitiendo a los clientes crear subscripciones duraderas (*durable*). Las subscripciones duraderas permiten recibir mensajes que fueron enviados cuando los subscriptores no estaban conectados. De este modo, las subscripciones duraderas ofrecen flexibilidad y fiabilidad a las colas pero aun así permiten a los clientes enviar mensajes a múltiples recipientes.

Al poder haber múltiples publicadores y múltiples consumidores en el mismo tópico, el modelo Pub/Sub es especialmente útil en situaciones donde un grupo de aplicaciones quiere notificar a otras de un evento particular. Por ejemplo, una aplicación de CRM, al crear un cliente, puede necesitar comunicar a otras aplicaciones la creación de este cliente.

1.3. 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 de un cliente a multiples proveedores

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.

1.3.1. 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

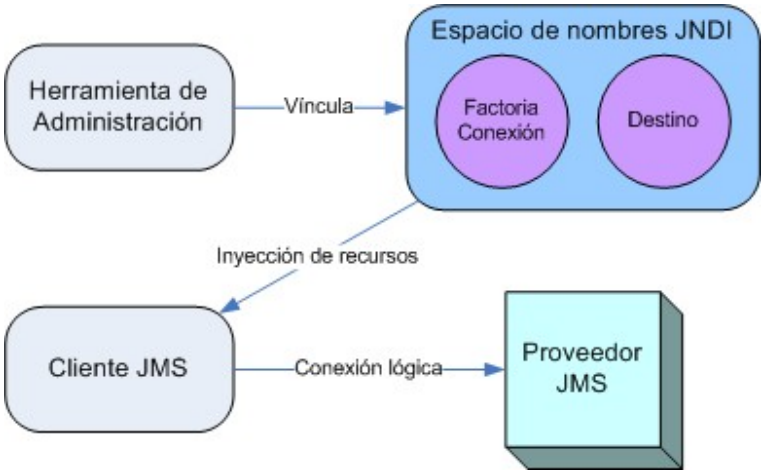
El servidor de aplicaciones integra a diferentes proveedores JMS mediante la arquitectura de conectores. De este modo, accedemos a un proveedor a través de un adaptador de recursos. Esto permite que los propietarios de MOMs creen proveedores JMS que puedan enchufarse en múltiples servidores de aplicaciones,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

1.3.2. 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:

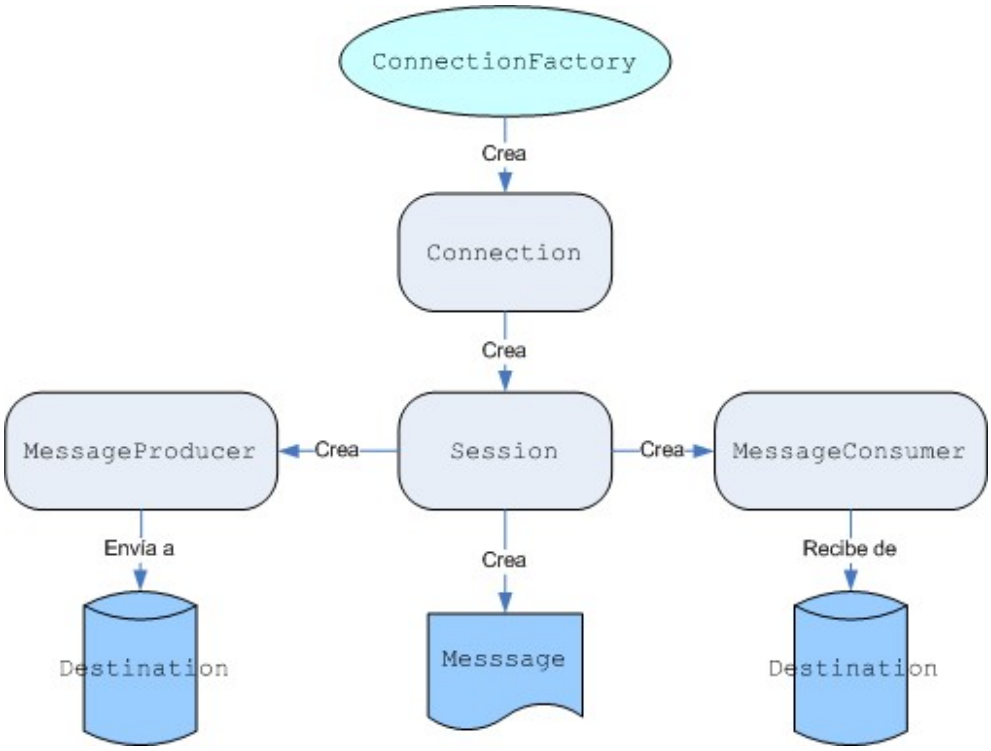


Arquitectura JMS

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.

1.3.3. El Modelo de Programación JMS

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



Modelo Programación JMS

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

1.3.3.1. 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.

1.3.3.2. 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`.

1.3.3.3. 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.

1.3.3.4. 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);
```

1.3.3.5. 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.

1.3.3.6. 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.

1.3.4. Modelos Específicos

Hasta ahora nos hemos centrado en el API JMS, el cual es común para ambos dominios de mensajería. A continuación veremos cada uno de ellos en detalle.

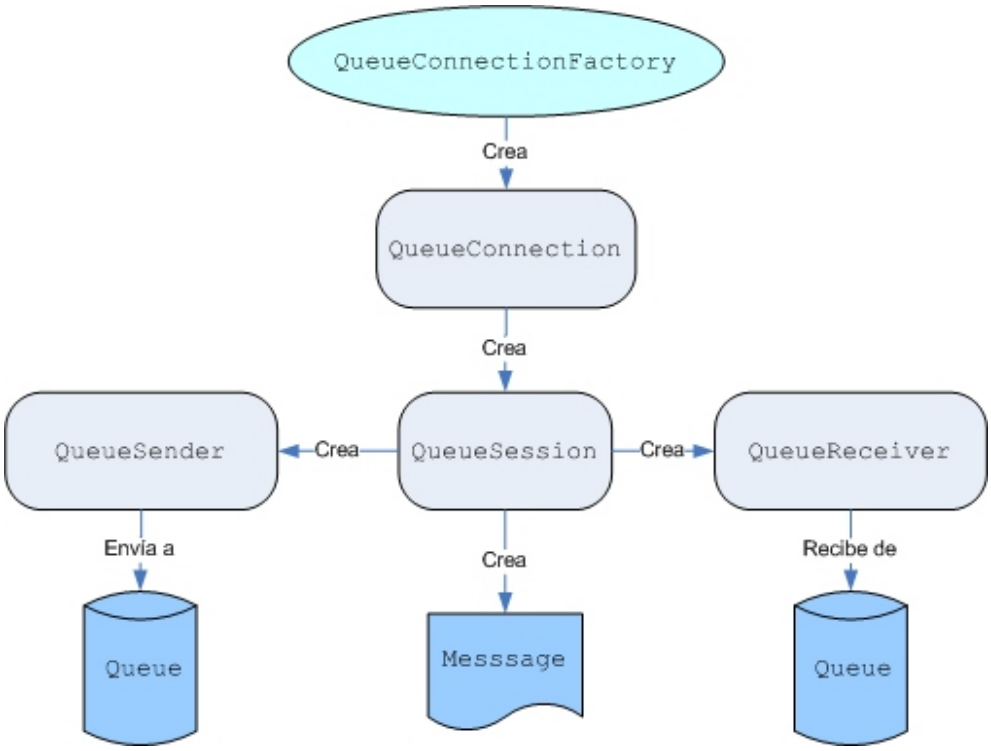
1.3.4.1. API Punto a Punto

El API punto a punto se refiere de manera específica a los interfaces basados en el uso de colas, de modo que los interfaces de este API son:

- `QueueConnectionFactory`
- `Queue`
- `QueueConnection`
- `QueueSession`
- `QueueSender`
- `QueueReceiver`

Igual que en modelo general JMS, obtendremos tanto los objetos `QueueConnectionFactory` como `Queue` del proveedor JMS via JNDI (mediante la inyección de código vía anotaciones). Como puede observarse, la mayoria de los interfaces añaden el sufijo `Queue` al nombre del interfaz JMS. Las excepciones son el interfaz `Destination`, que se llama `Queue`, y los interfaces `MessageProducer` y `MessageConsumer` que pasan a ser `QueueSender` y `QueueReceiver`, respectivamente.

A continuación podemos ver el mismo gráfico de antes pero ahora respecto al modelo punto a punto.



Modelo Programación Punto a Punto

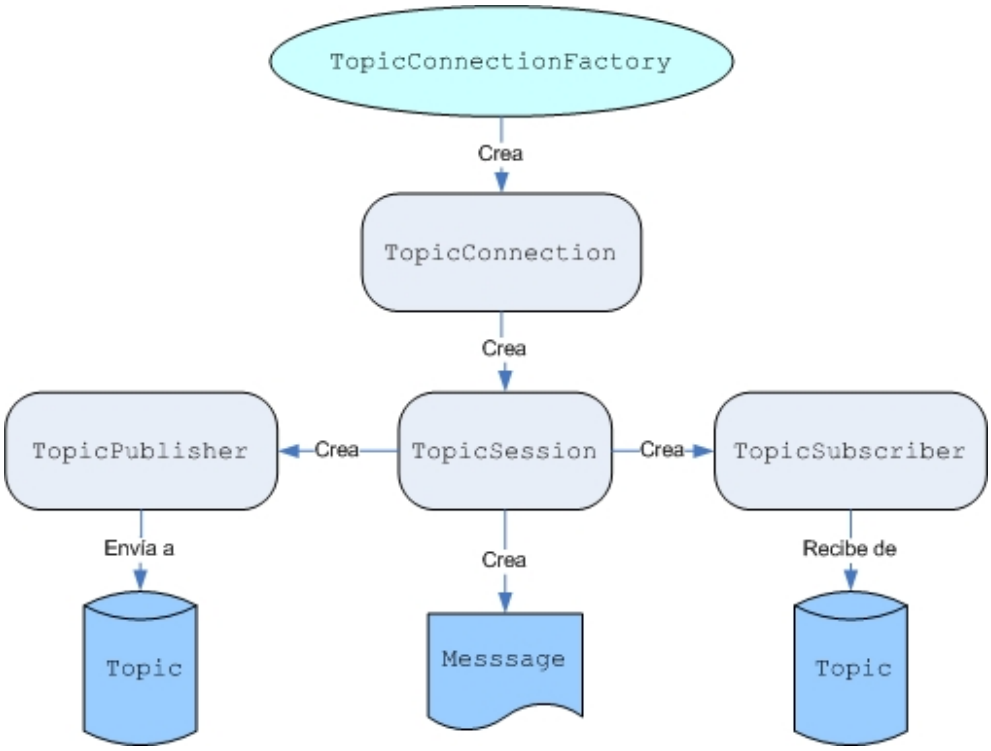
Las aplicaciones que usan un modelo de mensajería punto a punto normalmente utilizan este API específico en vez del API general.

1.3.4.2. API Publicación/Subscripción

Cada elemento específico para las colas, tiene su correspondencia con el uso de tópicos. Así pues, tendremos los siguientes interfaces:

- `TopicConnectionFactory`
- `Topic`
- `TopicConnection`
- `TopicSession`
- `TopicPublisher`
- `TopicSubscriber`

A continuación podemos ver el mismo gráfico de antes pero ahora respecto al modelo *publish-subscribe*.

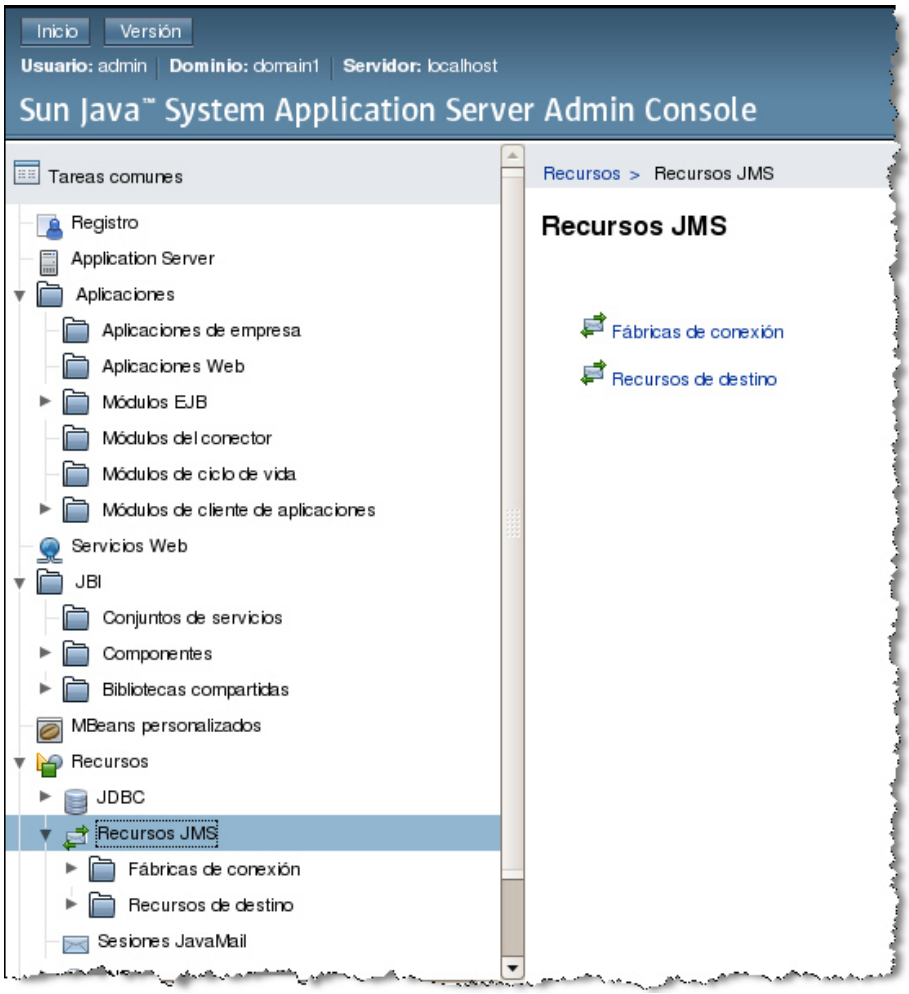


Modelo Programación Publicación-Subscripción

1.4. Recursos JMS en Glassfish

Para una comunicación necesitamos dos tipos de objetos administrados: una factoría de conexiones y una cola (o un tópico). Estos objetos administrados están, como su nombre indica, gestionados por Glassfish. En Glassfish, como en otros servidores de aplicaciones, el administrador puede añadir o actualizar estos recursos. Para ver como se hace entremos en la consola de administración.

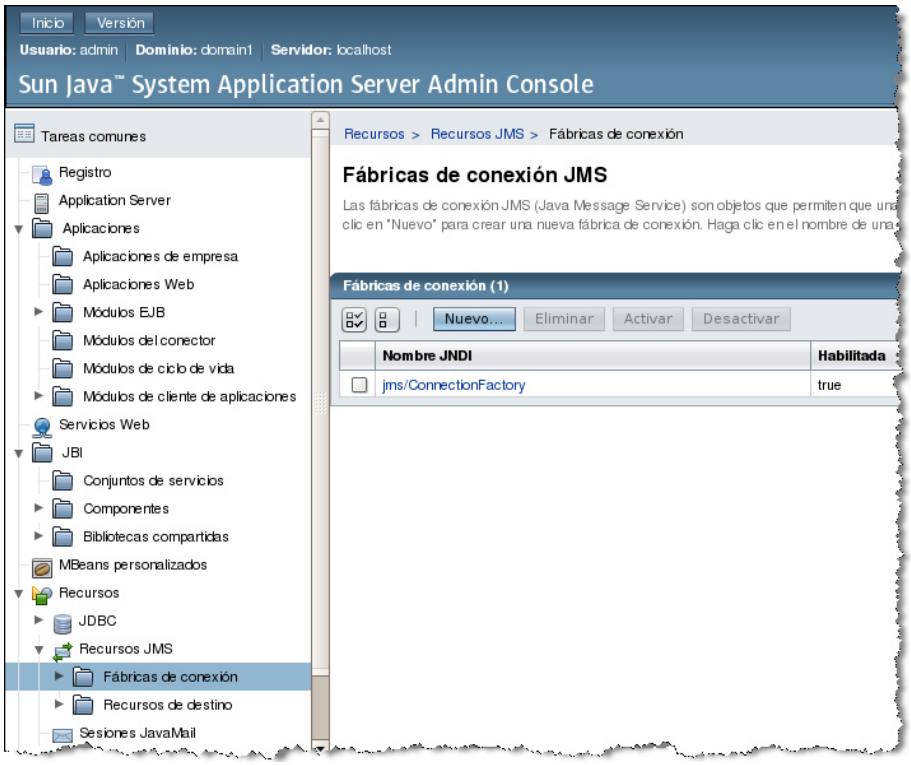
Una vez estemos en la consola, consultaremos el panel izquierdo y seleccionaremos **Recursos** y a su vez **Recursos JMS**:



Acceso a los recursos JMS en Glassfish

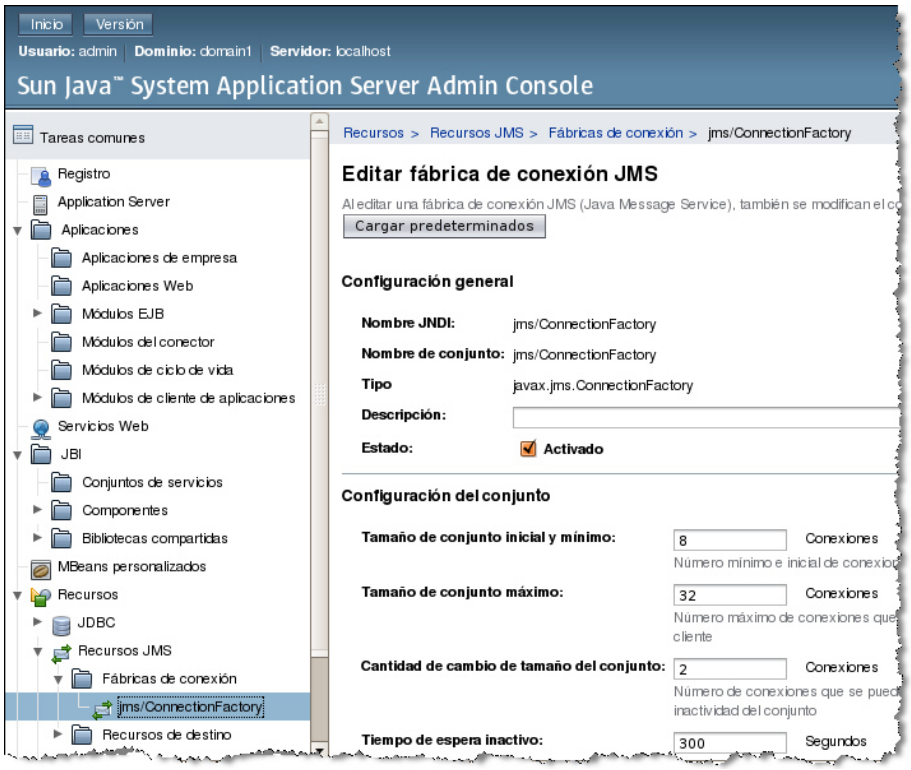
1.4.1. Factorías de Conexión

Entonces, si pinchamos en el panel de la derecha, tenemos acceso a **Fábricas de conexión** y **Recursos de destino**. Al pinchar en la fábricas de conexión podemos crear una nueva o bien editar las que estén creadas.



Acceso a las fábricas de conexión

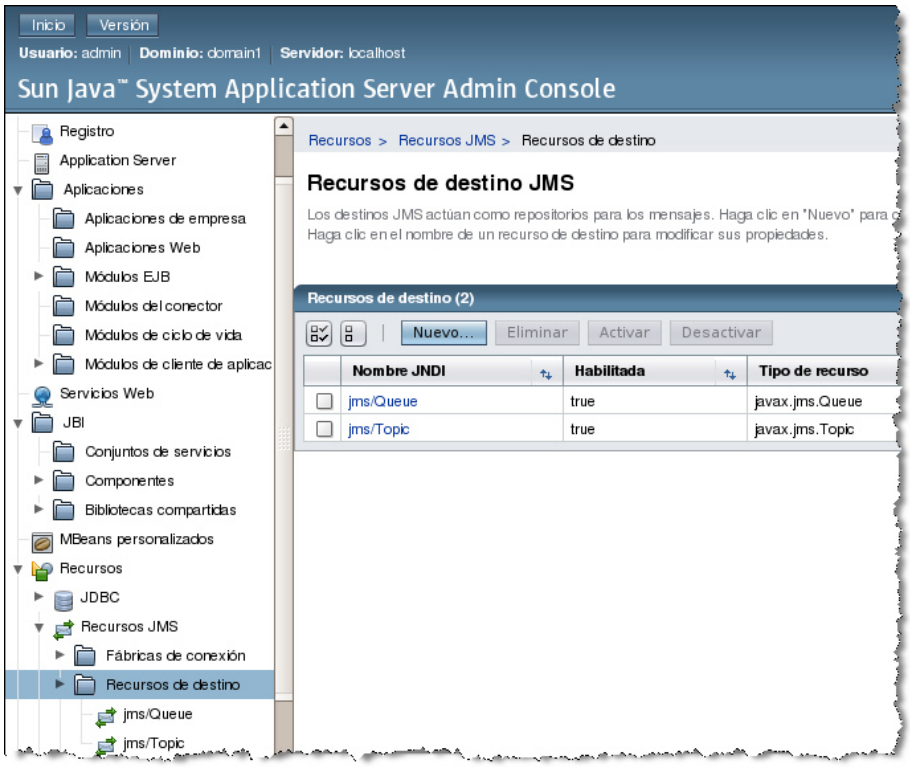
En el ejemplo anterior, hay creada una de ellas cuyo nombre JNDI es **jms/ConnectionFactory** y aparece como habilitada. Si queremos ver sus parámetros hacemos click y vemos lo siguiente:



Algunos parámetros de las fábricas de conexión

1.4.2. Destinos

De la misma forma, si consultamos **Recursos de destino** podemos ver los nombres JNDI de las colas y tópicos que están habilitados o, en su defecto, crearlos.



Recursos de destino

Desde el panel de destinos se pueden activar, desactivar, crear, borrar.

Pero en cualquier caso, ya sea a través de *Ant* o del acceso a la consola, la factoría y los destinos a usar deberán estar creados antes de que el cliente se ejecute.

1.5. 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)

A continuación, veremos un ejemplo por cada tipo de dominio de JMS. Ambos ejemplos se basan

1.5.1. Ejemplo de PTP

A continuación vamos a ver un ejemplo de dos clientes PTP, uno que produce mensajes a una cola, y otro que consume dichos mensajes de la misma cola.

Lo primero que tenemos que hacer es crear un proyecto de cliente empresarial *(`File -> New Project -> Java EE -> Enterprise Application Client`)*.

1.5.1.1. Productor

```
package org.especialistajee.jms;

// imports

public class Productor {

    @Resource(mappedName = "jms/ConnectionFactory")
    private static ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/Queue")
    private static Queue queue;

    public void enviaMensajeCola(String mundo) throws JMSException
{

        Connection connection = null;
        Session session = null;

        MessageProducer producer = null;
        Message message = null;
        boolean esTransaccional = false;

        try {
            connection = connectionFactory.createConnection();
            // Recordar llamar a start() para permitir el envio de
mensajes
            connection.start();
            // Creamos una sesion sin transaccionalidad y con envio
de acuse automatico
            session = connection.createSession(esTransaccional,
Session.AUTO_ACKNOWLEDGE);
            // Creamos el productor a partir de una cola
            producer = session.createProducer(queue);
            // Creamos un mensaje sencillo de texto
            message = session.createTextMessage(mundo);
```

```
            // Mediante le productor, enviamos el mensaje
            producer.send(message);

            System.out.println("Enviado mensaje [" + mundo + "]");
        } finally {
            // Cerramos los recursos
            producer.close();
            session.close();
            connection.close();
        }
    }

    public static void main(String[] args) throws Exception {
        Productor p = new Productor();
        p.enviaMensajeCola("Hola Mundo");
        p.enviaMensajeCola("Adios Mundo");
    }
}
```

Este ejemplo demuestra los pasos necesarios para crear un producto JMS y enviar un mensaje a un destino. Destacar que este cliente no se preocupa de que haya un consumidor JMS al otro lado esperando un mensaje. La mediación de los mensajes entre los productores y los consumidores es tarea del MOM, y de ahí una de las grandes virtudes de las aplicaciones JMS.

Una vez que el mensaje se ha enviado al destino, un consumidor recibirá el mensaje.

1.5.1.2. Consumidor

El consumidor lo hemos separado en dos clases distintas para diferenciar el tratamiento síncrono del asícrono.

Consumidor Síncrono

```
package org.especialistajee.jms;

// imports

public class ConsumidorSincrono {

    @Resource(mappedName = "jms/ConnectionFactory")
    private static ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/Queue")
    private static Queue queue;

    public void recibeMensajeSincronoCola() throws JMSException {

        Connection connection = null;
        Session session = null;

        MessageConsumer consumer = null;
        TextMessage message = null;
        boolean esTransaccional = false;

        try {
```

```
            connection = connectionFactory.createConnection();
            // Recordar llamar a start() para permitir la recepción
de mensajes
            connection.start();
            // Creamos una sesion sin transaccionalidad y con envio
de acuse automatico
            session = connection.createSession(esTransaccional,
Session.AUTO_ACKNOWLEDGE);
            // Creamos el consumidor a partir de una cola
            consumer = session.createConsumer(queue);
            // Recibimos un mensaje de texto
            message = (TextMessage) consumer.receive();

            // Sacamos el mensaje por consola
            System.out.println("Recibido sincrono [" +
message.getText() + "]");
            System.out.println("Fin sincrono");
        } finally {
            // Cerramos los recursos
            consumer.close();
            session.close();
            connection.close();
        }
    }

    public static void main(String[] args) throws Exception {
        ConsumidorSincrono p = new ConsumidorSincrono();
        p.recibeMensajeSincronoCola();
    }
}
```

En el método de tratamiento síncrono, el método `receive` bloquea la ejecución hasta que no recibe el mensaje.

Una llamada como ésta, sin argumentos, o con un argumento `0`, bloquea al receptor indefinidamente hasta que se produce un mensaje o se cierra la aplicación. Si queremos esperar solo por un tiempo limitado podemos pasarle un argumento `long` que indica un *timeout*. Si no queremos esperar nada, llamaremos al método `receiveNoWait()` (sin argumentos) que recibe el siguiente mensaje si hay alguno disponible, y devuelve `null` en otro caso.

Consumidor Asíncrono

```
package org.especialistajee.jms;

// imports

public class ConsumidorAsincrono {

    @Resource(mappedName = "jms/ConnectionFactory")
    private static ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/Queue")
    private static Queue queue;

    public void recibeMensajeAsincronoCola() throws JMSException {
```

```
        Connection connection = null;
        Session session = null;

        MessageConsumer consumer = null;
        TextoListener listener = null;
        boolean esTransaccional = false;

        try {
            connection = connectionFactory.createConnection();
            // Creamos una sesion sin transaccionalidad y con envio
de acuse automatico
            session = connection.createSession(esTransaccional,
Session.AUTO_ACKNOWLEDGE);
            // Creamos el consumidor a partir de una cola
            consumer = session.createConsumer(queue);
            // Creamos el listener, y lo vinculamos al consumidor
-> asincrono
            listener = new TextoListener();
            consumer.setMessageListener(listener);
            // Llamamos a start() para empezar a consumir
            connection.start();

            // Sacamos el mensaje por consola
            System.out.println("Fin asincrono");
        } finally {
            // Cerramos los recursos
            consumer.close();
            session.close();
            connection.close();
        }
    }

    public static void main(String[] args) throws Exception {
        ConsumidorAsincrono p = new ConsumidorAsincrono();
        p.recibeMensajeAsincronoCola();
    }
}
```

De la parte asíncrona, destacar la necesidad de crear un `MessageListener` que será el encargado de recibir el mensaje y realizar las acciones oportunas. Por ello, una vez creado el listener, se le adjunta al consumidor.

A continuación podemos ver el listener de mensajes para la recepción asíncrona, el cual implementa el método `onMessage`:

```
package org.especialistajee.jms;

// imports

public class TextoListener implements MessageListener {

    /**
    * Casts del mensaje a un mensaje de texto y se muestra por
consola
    * @param message mensaje de entrada
    */
    @Override
```

```
    public void onMessage(Message message) {
        TextMessage msg = null;

        try {
            if (message instanceof TextMessage) {
                msg = (TextMessage) message;
                System.out.println("Recibido asincrono [" +
msg.getText() + "]");
            } else {
                System.err.println("El mensaje no es de tipo
texto");
            }
        } catch (JMSException e) {
            System.err.println("JMSException en onMessage(): " +
e.toString());
        } catch (Throwable t) {
            System.err.println("Exception en onMessage():" +
t.getMessage());
        }
    }
}
```

Si lanzamos el consumidor antes del productor entonces éste estará esperando hasta que el productor los envíe y obtendremos la misma salida. Esto sucede si usamos colas, cuando usamos tópicos como destinos las cosas cambian, como veremos en la siguiente sección.

Si lanzamos el consumidor después del productor, el primero recibirá los mensajes dependiendo del tiempo de vida de los mismos (que por defecto es 0).

1.5.2. Ejemplo de Pub/Sub

De igual modo, con el modelo *publish/subscribe* tenemos los siguientes ejemplos:

1.5.2.1. Publicador

```
package org.especialistajee.jms;

// imports

public class Publicador {

    @Resource(mappedName = "jms/ConnectionFactory")
    private static ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/Topic")
    private static Topic topic;

    public void enviaMensajeTopico(String mundo) throws
JMSException {

        Connection connection = null;
        Session session = null;

        MessageProducer publisher = null;
        Message message = null;
```

```
        boolean esTransaccional = false;

        try {
            connection = connectionFactory.createConnection();
            // Recordar llamar a start() para permitir el envio de
mensajes
            connection.start();
            // Creamos una sesion sin transaccionalidad y con envio
de acuse automatico
            session = connection.createSession(esTransaccional,
Session.AUTO_ACKNOWLEDGE);
            // Creamos el publicador a partir de un topico
            publisher = session.createProducer(topic);
            // Creamos un mensaje sencillo de texto
            message = session.createTextMessage(mundo);
            // Mediante el publicador, enviamos el mensaje
            publisher.send(message);

            System.out.println("Enviado mensaje [" + mundo + "]");
        } finally {
            // Cerramos los recursos
            publisher.close();
            session.close();
            connection.close();
        }
    }

    /**
     * Creamos y lanzamos el publicador
     */
    public static void main(String[] args) throws Exception {
        Publicador p = new Publicador();
        p.enviaMensajeTopico("Hola Mundo");
        p.enviaMensajeTopico("Adios Mundo");
    }
}
```

Si comparamos el código del productor con el del publicador podemos ver que es el mismo excepto a la hora de crear el `MessageProducer`. En el caso de *PTP* se utiliza una cola, y para *Pub/Sub* un tópico.

1.5.2.2. Subscriptor

Del mismo modo que con los consumidores, hemos separado los subscriptores en dos:

Subscriptor Síncrono

```
package org.especialistajee.jms;

// imports

public class SubscriptorSincrono {

    @Resource(mappedName = "jms/ConnectionFactory")
    private static ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/Topic")
```

```
    private static Topic topic;

    public void recibeMensajeSincronoTopico() throws JMSException {

        Connection connection = null;
        Session session = null;

        MessageConsumer subscriber = null;
        TextMessage message = null;
        boolean esTransaccional = false;

        try {
            connection = connectionFactory.createConnection();
            // Recordar llamar a start() para permitir el envio de
mensajes
            connection.start();
            // Creamos una sesion sin transaccionalidad y con envio
de acuse automatico
            session = connection.createSession(esTransaccional,
Session.AUTO_ACKNOWLEDGE);
            // Creamos el subscriptor a partir de un topico
            subscriber = session.createConsumer(topic);
            // Recibimos un mensaje de texto
            message = (TextMessage) subscriber.receive();

            // Sacamos el mensaje por consola
            System.out.println("Recibido sincrono [" +
message.getText() + "]");
            System.out.println("Fin sincrono");
        } finally {
            // Cerramos los recursos
            subscriber.close();
            session.close();
            connection.close();
        }
    }

    public static void main(String[] args) throws Exception {
        SubscriptorSincrono p = new SubscriptorSincrono();
        p.recibeMensajeSincronoTopico();
    }
}
```

Al comparar con el consumidor, este subscriptor también es idéntico excepto a la hora de crear el `MessageConsumer` (éste se basa en un tópico).

Subscriptor Asíncrono

```
package org.especialistajee.jms;

// imports

public class SubscriptorAsincrono {

    @Resource(mappedName = "jms/ConnectionFactory")
    private static ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/Topic")
```

```
    private static Topic topic;

    public void recibeMensajeAsincronoTopico() throws JMSException
{

        Connection connection = null;
        Session session = null;

        MessageConsumer subscriber = null;
        TextoListener listener = null;
        boolean esTransaccional = false;

        try {
            connection = connectionFactory.createConnection();
            // Creamos una sesion sin transaccionalidad y con envio
de acuse automatico
            session = connection.createSession(esTransaccional,
Session.AUTO_ACKNOWLEDGE);
            // Creamos el subscriptor a partir de un topico
            subscriber = session.createConsumer(topic);
            // Creamos el listener, y lo vinculamos al subscriptor
-> asincrono
            listener = new TextoListener();
            subscriber.setMessageListener(listener);
            // Llamamos a start() para empezar a consumir
            connection.start();

            // Sacamos el mensaje por consola
            System.out.println("Fin asincrono");
        } finally {
            // Cerramos los recursos
            subscriber.close();
            session.close();
            connection.close();
        }
    }

    public static void main(String[] args) throws Exception {
        SubscriptorAsincrono p = new SubscriptorAsincrono();
        p.recibeMensajeAsincronoTopico();
    }
}
```

Y el asíncrono también es similar, tanto que el listener que utilizamos es semejante al utilizado en el ejemplo del consumidor asíncrono.

Si ejecutamos el subscriptor antes que el publicador, veremos que el subscriptor síncrono se *cuelga* puesto que los mensajes que habían sido enviados antes de lanzar el subscriptor no pueden ser recibidos. Esto se debe al modelo Pub/sub, ya que un cliente subscrito a un tópico puede leer mensajes de él solo *después* de haber realizado la subscripción, y en este caso sucede al contrario. Ocurre lo mismo en el caso asíncrono.

2. Ejercicios de Introducción a JMS

2.1. Dominio JMS

Para que todas las factorias sean independientes del resto de módulos, vamos a crear un nuevo dominio. Para ello, crearemos desde Netbeans un nuevo servidor al que denominaremos **'GlassFish Server 3 - JMS'**. Al crear el servidor crearemos un nuevo dominio personal que llamaremos `domain-jms` el cual situaremos dentro de la carpeta de dominios de Glassfish, es decir, en `/home/especialista/glassfish-v2.1/domains/domain-jms`.

Todos los ejercicios de este módulo utilizarán este dominio.

2.2. Primeros Ejemplos (1p)

Se plantea como ejercicio la creación de los ejemplos PTP y Pub/Sub vistos en los apuntes.

Antes de nada habrá que crear en *Glassfish* tanto la factoría de conexiones como los recursos de cola y tópico vistos en los apuntes.

En cuanto a los clientes, crearemos un proyecto de aplicación cliente para cada elemento de cada dominio. Así pues, tendremos los proyectos `jms-productor`, `jms-consumidor-sync`, `jms-consumidor-async`, `jms-publicador`, `jms-subscriptor-sync` y `jms-subscriptor-async`.

2.3. Ejemplos con Clases Específicas (1p)

Los ejemplos que habéis hecho en el primer ejercicio se basan en los interfaces JMS de más alto nivel. Así, cuando estamos accediendo a una cola, hemos utilizado las clases `ConnectionFactory`, `Connection`, `Session`, `MessageConsumer`, etc...

Se plantea como ejercicio rescribir los 6 proyectos y que cada clase utilice las clases específicas de cada dominio. Es decir, en el caso de la cola, utilizaremos `QueueConnectionFactory`, `QueueConnection`, `QueueSession`, `QueueReceiver`, etc...

Las nuevas clases se situarán en los mismos proyectos, pero nombrando a las clases con el sufijo `Especifico`. Por ejemplo, la nueva clase `Consumidor` se llamará `ConsumidorEspecifico`.

Para tener más claro las relaciones de las clases se recomienda consultar los javadocs del API de JMS: java.sun.com/javaee/6/docs/api/javax/jms/package-summary.html

2.4. Semáforo Hipotecario (2p)

Vamos a hacer un pequeño proyecto para este módulo que va a simular un semáforo hipotecario.

El semáforo va a recibir peticiones sobre hipotecas, y dependiendo del riesgo que conllevan, responderá con Rojo (denegar), Naranja (previo análisis de un consultor financiero) o Verde (aceptar).

El semáforo recibirá los siguientes parámetros:

- Nombre de la entidad financiera
- Cuantía de la hipoteca
- Años de la hipoteca
- Tipo de interés anual hipotecario
- Nómina mensual del cliente (neta, sin retenciones). Suponemos que cobra 12 nominas al año.

Las reglas asociadas al semáforo para responder con uno u otro color dependiendo de la cuota mensual de la hipoteca serán:

- Si la cuota mensual supone menos del 30% de la nomina del cliente, devolverá verde.
- Si la cuota mensual supone entre el 30% y el 40% de la nomina, devolverá naranja.
- Si la cuota mensual supone más del 40% de la nomina, devolverá rojo.

Para calcular la cuota mensual, tenéis la formula en la wikipedia : es.wikipedia.org/wiki/Contrato_de_Hipoteca.

2.4.1. jms-banco

Para realizar esta aplicación necesitaréis crear 2 proyectos. Un proyecto productor de mensajes (`jms-banco`), el cual enviará mensajes a una cola, la cual llamaremos `SemaforoHipotecarioRequestQueue`.

Todo esto lo haremos dentro de una clase `Banco` dentro de `org.especialistajee.jms`.

El mensaje se enviará desde un método con la siguiente firma:

```
void enviaHipoteca(String banco, double cuantia, int anyos, double
interes, double nomina);
```

2.4.2. jms-semaforo

El segundo proyecto (`jms-semaforo`) se encarga de escuchar los mensajes de forma asíncrona y devolver uno de los tres colores comentados anteriormente en la cola de respuesta. La consumición del mensaje se realizará en un método con la siguiente firma:

```
void analizaHipoteca();
```

Este método obtendrá el color de la hipoteca (y de momento, lo sacará por consola).

Para facilitaros el trabajo, se os da comenzada una clase de prueba para comprobar la **lógica de negocio** del semáforo (debéis completar la prueba para que abarque la mayor casuística posible):

```
package es.ua.jtech.jms;

// imports
public class SemaforoBRTest {

    private SemaforoBR sbr = null;

    public SemaforoBRTest() {
        sbr = SemaforoBR.getInstance();
    }

    @Test
    public void devuelveVerde() {
       String color = "Verde";

       Assert.assertTrue("Deberia ser " + color,
        sbr.obtenerColorHipoteca("BBVA", 100000, 30, 5,
1800).equals(color));
       Assert.assertTrue("Deberia ser " + color,
        sbr.obtenerColorHipoteca("BBVA", 50000, 20, 5,
1800).equals(color));

       Assert.assertTrue("Deberia ser " + color,
        sbr.obtenerColorHipoteca("Banesto", 100000, 40, 4,
1500).equals(color));
       Assert.assertTrue("Deberia ser " + color,
        sbr.obtenerColorHipoteca("Banesto", 100000, 35, 3,
1500).equals(color));

    // Resto de métodos de prueba....
}
```

Además, también tenéis el esqueleto para crear la lógica de negocio del semáforo:

```
package es.ua.jtech.jms;

public class SemaforoBR {
    private static SemaforoBR me = new SemaforoBR();

    public final static String ROJO = "Rojo";
    public final static String VERDE = "Verde";
    public final static String NARANJA = "Naranja";

    private SemaforoBR() { }

    public static SemaforoBR getInstance() {
        return me;
    }

    public String obtenerColorHipoteca(
            String banco, double cuantia, int anyos, double
```

```
interes, double nomina) {
        String result = null;

        double interesMensual = interes / 12;
        int plazo = anyos * 12;

        double cuota = (cuantia * interesMensual) /
            (100 * (1 - Math.pow(1 + (interesMensual / 100),
-plazo)));
        double ratio = (cuota / nomina) * 100;

        // TODO Completar logica de negocio
    }
}
```

2.4.3. Enviando y parseando la información

En cuanto al envío y recepción de la información mediante un mensaje de texto, tendremos que agrupar la información para luego parsearla. Para ello, vamos a utilizar el carácter '#' para unir cada campo. Para construir el mensaje haremos algo similar a:

```
String miMensaje = banco + "#" + cuantia + "#" + anyos + "#" +
interes + "#" + nomina;
```

Y para parsearlo, utilizaremos el método `split` de `String`, tal que así:

```
String[] tokens = miMensaje.split("#");

String banco = tokens[0];
double cuantia = Double.parseDouble(tokens[1]);
int anyos = Integer.parseInt(tokens[2]);
double interes = Double.parseDouble(tokens[3]);
double nomina = Double.parseDouble(tokens[4]);
```

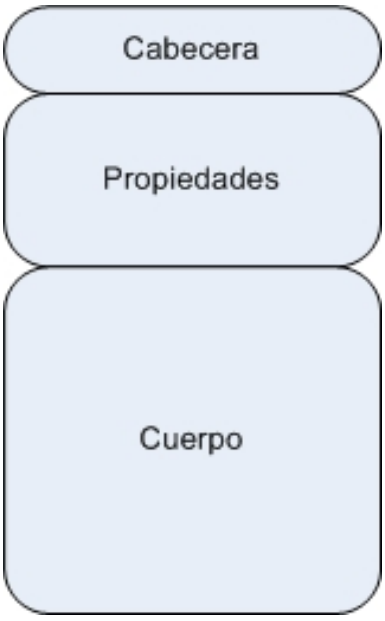
3. Mensajes y Robustez

3.1. Anatomía de un Mensaje

El mensaje JMS es el concepto más importante de la especificación JMS. Cada concepto de la especificación está relacionado con un mensaje porque es el medio mediante el cual los datos y eventos de negocio se transmiten a través de un proveedor JMS.

Debido a que los sistemas de mensajería han sido implementados sin seguir ningún formato de mensaje estándar, existe un gran abanico de formatos de mensaje entre los MOMs. Esto es así, y JMS no lo quiere cambiar. De hecho, JMS no define cual debe ser el formato de los mensajes, sino mediante un modelo de mensaje unificado y abstracto garantiza la portabilidad del código. Esto es, en vez de dictar cómo debe ser el formato de los mensajes, especifica el API a utilizar para construir el mensaje.

El modelo de mensaje unificado bajo JMS especifica que todos los mensajes deben representarse mediante objetos que implementen el interfaz `javax.jms.Message`. Y este interfaz separa un mensaje JMS en tres partes:



Partes de un Mensaje

Las cabeceras ofrecen metadatos sobre el mensaje utilizado por ambos clientes y el proveedor JMS. Las propiedades son campos opcionales dentro del mensaje para añadir información adicional al mensaje. El cuerpo puede contener tanto texto como datos binarios mediante los diferentes tipos de mensajes.

3.1.1. Cabecera

Todos los mensajes JMS soportan la misma lista estándar de cabeceras, y el API JMS ofrece métodos (*getters y setters*) para trabajar con dichas cabeceras. Muchas de las cabeceras se asignan automáticamente, ya sea por el cliente o por el proveedor.

Las cabeceras que automáticamente rellena el proveedor JMS al realizar la llamada al método `send()` del cliente son:

- **JMSDestination** - Destino al que se envía el mensaje. En el caso de un modelo PTP especifica la cola, y el Pub/Sub el tópico. Esta cabecera es muy útil para aquellos clientes que consumen mensajes de más de un destino.
- **JMSDeliveryMode** - JMS soporta 2 tipos de modos de entrega: persistente y no-persistente. El modo de entrega por defecto es persistente. Cada modo de entrega incurre en su propia sobrecarga e implica un nivel particular de fiabilidad.
  - **Persistent** - Informa al proveedor que persista los mensajes para que no se pierdan en el caso de que caiga el proveedor. El proveedor debe entrar un mensaje persistente una única vez. Dicho de otro modo, si el proveedor JMS falla, el mensaje no se pierde y no se enviara dos veces. Los mensajes persistentes incurren en una sobrecarga debido a la necesidad de almacenar el mensaje, de modo que la fiabilidad prevalece sobre el rendimiento.
  - **Non-Persistent** - Informa al proveedor JMS que no persista los mensajes. Un proveedor JMS debe entregar un mensaje persistente como mucho una vez. Dicho de otro modo, si el proveedor JMS falla, el mensaje se puede haber perdido, pero no se entregará dos veces. Los mensajes no persistentes incurren en una menor sobrecarga, de modo que el rendimiento prevalece sobre la fiabilidad.

  El modo de entrega se asigna en el productor y se aplica a todos los mensajes enviados desde dicho productor:

  ```
  producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT)
  ```

  De forma individual, el modo de entrega se puede sobrecarga para cada mensaje:

  ```
  message.setJMSDeliveryMode(DeliveryMode.NON_PERSISTENT)
  ```

- **JMSExpiration** - Previene la entrega una vez que el mensaje ha expirado. Se trata de una cabecera muy útil para los mensajes sensibles al tiempo. Los clientes deberían tener en cuenta la expiración para desechar los mensajes sin procesar que han expirado. Por defecto, el tiempo de vida es 0.
- **JMSMessageID** - Una cadena que identifica unívocamente a un mensaje, se asigna por el proveedor JMS y debe empezar con 'ID:'. Esta cabecera se puede utilizar para el procesamiento de mensajes o para propósito de histórico en un mecanismo de almacenamiento de mensajes.
- **JMSPriority** - Se utiliza para asignar un nivel de importancia. Esta cabecera también la asigna el productor de mensajes. Una vez un productor le asigna la prioridad, se aplica a todos los mensajes enviados desde dicho productor. La prioridad se puede sobrecargar para mensajes individualizados. JMS define 10 niveles de prioridad de

mensajes, siendo 0 el más bajo, 9 el más alto y la prioridad normal de nivel 5. Los proveedores JMS no están obligados a implementar la ordenación de mensajes, aunque la mayoría lo hace. Lo que tienen que hacer es entregar los mensajes con mayor prioridad antes que los de menor prioridad.

- **JMSTimestamp** - Denota el instante en el que se envió el mensaje. Esta cabecera se puede deshabilitar par reducir la carga de trabajo del proveedor JMS.

Las cabeceras asignadas de forma opcional por el cliente son:

- **JMSCorrelationID** - Se utiliza para asociar el mensaje actual con el anterior. Esta cabecera se suele utilizar en la entrega asíncrona de mensajes, y en los mecanismos de *request/reply*.
- **JMSReplyTo** - Se utiliza para especificar un destino al cual debería responder el cliente. Se suele utilizar para estilos de mensajería de *request/reply*.
- **JMSType** - Utilizada para identificar semánticamente al mensaje, es decir, el tipo del cuerpo del mensaje. Esta cabecera la utilizan muy pocos proveedores.

Las cabeceras asignadas de forma opcional por el proveedor:

- **JMSRedelivered** - Se utiliza para indicar que un mensaje ha sido reenviado. Esto puede suceder si un consumidor falla en el acuse de recibo o si el proveedor JMS no ha sido notificado del envío.

3.1.2. Propiedades

Las propiedades son más o menos cabeceras adicionales que pueden especificarse en un mensaje. Al contrario que las cabeceras, son opcionales. Las propiedades son pares de {nombre, valor}, y JMS ofrece métodos genéricos para trabajar con propiedades cuyo tipo de valor sea `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, o `String`. Las propiedades se inicializan cuando se envía un mensaje, y al recibir pasan a ser de sólo lectura.

Existen tres tipos de propiedades: propiedades arbitrarias o de aplicación, propiedades definidas por JMS y propiedades específicas del proveedor.

3.1.2.1. Propiedades de Aplicación

Estas propiedades son arbitrarias y las define una aplicación JMS. Los desarrolladores de aplicaciones JMS pueden definir libremente cualquier propiedad que necesiten mediante los métodos genéricos comentados anteriormente.

Si queremos añadirle a un mensaje un propiedad Anyo e inicializarla a 2008, haremos:

```
message.setIntProperty("Anyo", 2008);
```

3.1.2.2. Propiedades Definidas por JMS

La especificación JMS reserva el prefijo de propiedades 'JMSX' para las propiedades definidas por JMS. El soporte de estas propiedades es opcional:

- **JMSXAppID** - Identifica la aplicación que envía el mensaje.
- **JMSXConsumerTXID** - Identificador de la transacción dentro de la cual el mensaje ha sido consumido.
- **JMSXDeliveryCount** - Número de intentos de entrega del mensaje.
- **JMSXGroupID** - Grupo del mensaje del cual forma parte el mensaje.
- **JMSXGroupSeq** - Número de secuencia del mensaje dentro del grupo.
- **JMSXProducerTXID** - Identificador de la transacción dentro de la cual el mensaje ha sido producido.
- **JMSXRcvTimestamp** - Instante en el que el proveedor JMS entregó el mensaje al consumidor.
- **JMSXState** - Se utiliza para definir un estado específico del proveedor.
- **JMSXUserID** - Identifica al usuario que envía el mensaje.

La única recomendación que ofrece la especificación para el uso de estas propiedades es para las propiedades JMSXGroupID y JMSXGroupSeq, indicando que dichas propiedades deberían utilizarse por los clientes cuando se agrupen mensajes.

3.1.2.3. Propiedades Específicas del Proveedor

La especificación JMS reserva el prefijo de propiedad 'JMS_vendor-name' para propiedades específicas del proveedor. Cada proveedor define sus propios valores para sus propiedades. En su mayoría las utilizan los clientes no-JMS, atados al proveedor, y por tanto, no deberían utilizarse entre mensajerías JMS.

3.1.3. Cuerpo

El cuerpo de los mensajes permite enviar y recibir datos e información con diferentes formatos, ofreciendo compatibilidad con los formatos de mensaje existentes. El cuerpo también se conoce como **carga** (*payload*).

JMS define seis tipos de cuerpo para los mensajes, también conocidos tipos de mensajes:

- **Message** - Tipo de mensaje base. Se utiliza para enviar un mensaje sin cuerpo, solo cabeceras y propiedades. Normalmente se utiliza para notificación simple de eventos.
- **MapMessage** - Compuesto de un conjunto de pares {nombre,valor}. El tipo de los nombres es `String`, y los valores tipos primitivos Java. A los nombres (que no están ordenados) podemos acceder de forma secuencial mediante un enumerador, o por acceso directo por nombre.
- **BytesMessage** - Contiene un array de bytes sin interpretar. Se utiliza para hacer coincidir el cuerpo con un formato de mensaje existente (*legacy*).
- **StreamMessage** - El cuerpo es un flujo de tipos primitivos Java, cuya lectura y escritura se realiza de modo secuencial.
- **TextMessage** - Un mensaje cuya carga es un `String`. Se suele utilizar para enviar

texto simple y datos XML.

- **ObjectMessage** - La carga es un objeto Java `Serializable`. Normalmente se utiliza para trabajar con objetos Java complejos.

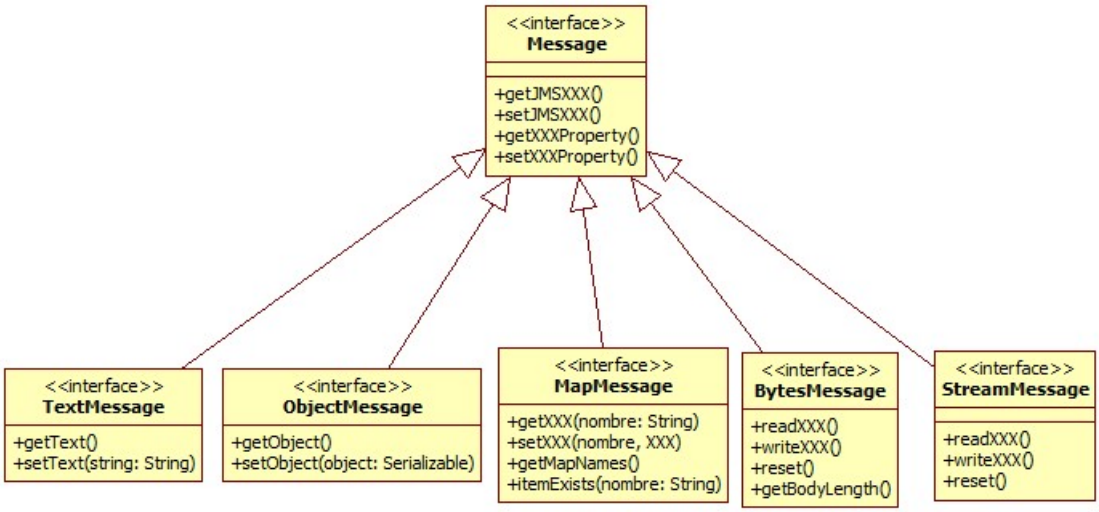


Diagrama de Clases UML con los diferentes tipos de Mensaje

3.1.3.1. Creando Mensajes

El API JMS ofrece diferentes métodos para crear los mensajes de cada tipo (a partir de una `Session`, mediante los métodos `createXXXMessage`, siendo XXX el tipo de mensaje), y para rellenar sus contenidos. Por ejemplo, para crear y enviar un mensaje de texto podríamos hacer:

```
TextMessage mensaje = session.createTextMessage();
mensaje.setText(miTexto);
producer.send(mensaje);
```

Si lo que queremos es enviar un objeto, crearemos un mensaje del siguiente modo:

```
ObjectMessage mensaje = session.createObjectMessage();
mensaje.setObject(miLibroEntity);
// el objeto miLibroEntity debe ser Serializable !!!!
```

Así pues, si al rellenar el mensaje de tipo objeto, el objeto no es `Serializable` obtendremos una excepción del tipo `MessageFormatException`.

3.1.3.2. Recibiendo Mensajes

En el destino, el mensaje llega como un objeto de `Message` genérico, y debemos hacer el *cast* al tipo de mensaje apropiado. Los diferentes tipos de mensaje ofrecen diversos métodos para extraer el contenido del mensaje. Por ejemplo, el código necesario para

extraer el contenido de un mensaje de texto:

```
Message m = consumer.receive();
if (m instanceof TextMessage) {
        TextMessage mensaje = (TextMessage) m;
        System.out.println("Leyendo mensaje: " +
mensaje.getText());
} else {
        // Manejar el error
}
```

3.2. Filtrado de Mensajes

En ocasiones necesitaremos ser más selectivos a la hora de recibir un mensaje de una determinada cola o tópico. El filtrado de mensajes nos va a permitir elegir que mensajes consumir, de modo que no sea obligatorio consumir todos los mensajes de un destino.

3.2.1. Selector de Mensajes

Si queremos filtrar los mensajes que recibe la aplicación lo podemos hacer mediante un selector de mensajes, ya que permite a un receptor definir en que mensajes esta interesado a través del uso de las cabeceras y propiedades. Solo aquellos mensajes cuyas cabeceras o propiedades coincidan con el selector estarán disponibles para su consumición (los selectores **no pueden referenciar al cuerpo** del mensaje).

Un selector de mensajes es un `String` que contiene una expresión, definida mediante un subconjunto de la sintaxis de expresiones condicionales de SQL92. Por lo tanto, mediante una expresión condicional cuyos operandos son las cabeceras o las propiedades y el operador adecuado (operador lógico, `between`, `like`, `in(...)`, `is null`), los clientes puede definir cuales son los mensajes permitidos. Por ejemplo, si nuestro mensaje define las propiedades `Anyo` y `Mes`, las siguientes son expresiones validas: `Anyo = 2011`, `Mes = 'Diciembre'`, `Mes LIKE '%BRE'`, `Anyo BETWEEN 2000 AND 2011`, etc...

Podemos encontrar más información sobre los operadores y las cabeceras aplicables dentro de un selector en el javadoc de la interfaz `Message`: java.sun.com/javaee/5/docs/api/javax/jms/Message.html

El selector de mensajes se utiliza a la hora de crear el consumidor de mensajes (o subscriptor durable), mediante los métodos `createConsumer(Destination destino, String selectorMensaje)` (y su respectivo `createDurableSubscriber`) del interfaz `Session`. Si el selector no es correcto, al crear el consumidor de mensajes obtendremos una excepción del tipo `javax.jms.InvalidSelectorException`. Una vez hemos creado el consumidor de mensajes, podemos consultar cual es su selector mediante el método `getMessageSelector()`. Así pues, por ejemplo, podriamos hacer:

```
MessageConsumer consumer = session.createConsumer(queue, "Anyo =
2011");
String selectorAnyo2008 = consumer.getMessageSelector();
```

Aquellos mensajes que no cumplen las expresiones del selector no se entregan al cliente. Esto significa que aquellos mensajes no entregados dependerán del `MessageConsumer` utilizado. Una vez se establece el selector de mensajes, no puede cambiarse. Tenemos que cerrar el consumidor (o borrarlo si es un subscriptor durable) y crear un nuevo consumidor con su nuevo selector de mensajes.

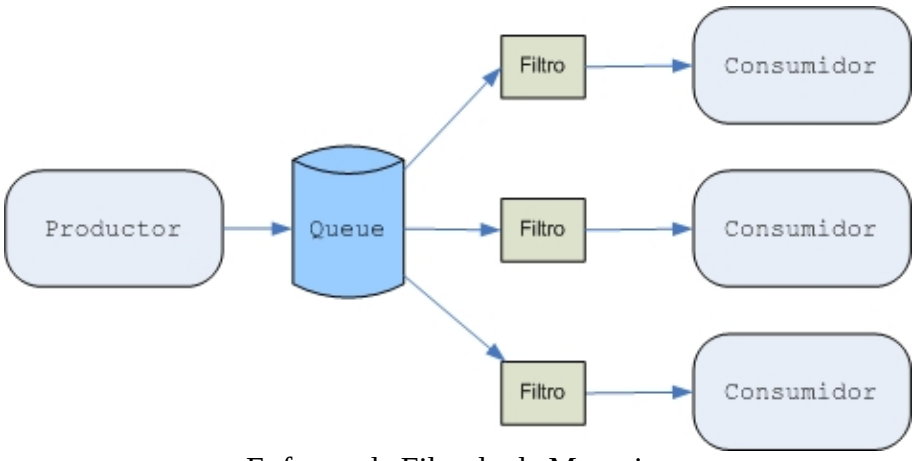
Mucho cuidado

Es muy importante que el selector de mensajes abarque toda la casuística posible.
Si tenemos un selector que hace "Anyo < 2010" y su complementario "Anyo > 2010", ¿Qué pasará con los mensajes cuyo año sea 2010?

3.2.2. Selectores vs Múltiples Destinos

Existen 2 enfoques a la hora de considerar una solución basada en mensajes. Podemos enviar todos los mensajes a un único destino JMS y utilizar filtrado de mensajes para consumir los mensajes específicos (enfoque de **Filtrado de Mensajes**), o podemos utilizar múltiples destinos JMS que contienen los mensajes ya filtrados (enfoque de **Múltiples Destinos**).

Gráficamente, podemos observar más facilmente los diferentes planteamientos:

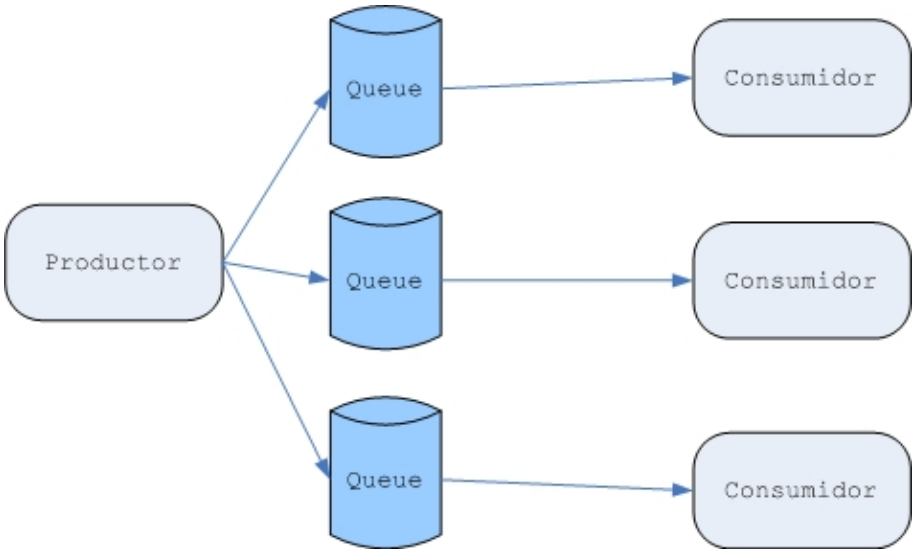


Enfoque de Filtrado de Mensajes

Claramente, se observa que el consumidor del mensaje el que tiene el control sobre el filtrado y sobre los mensajes que quiere recibir. Este enfoque dota de un alto nivel de desacoplamiento entre los componentes que producen mensajes respecto a los consumidores, ya que el productor necesita menos informacion sobre como se van a procesar los mensajes.

En cambio, con el enfoque de múltiples destino, el filtrado se realiza antes de enviar el mensaje al destino adecuado, ya que cada destino contiene los mensajes específicos que

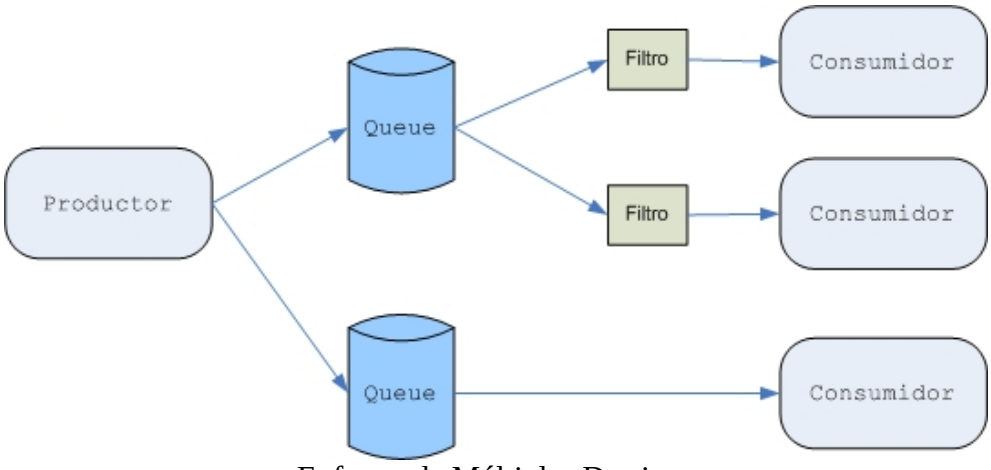
se utilizarán. Este filtrado se realiza mediante código Java para determinar a que destino se debe enviar el mensaje.



Enfoque de Múltiples Destinos

Podemos observar que ahora es el productor de mensajes el que tiene el control sobre el filtrado y decide a que destinos llegaran los mensajes apropiados. Un factor clave a la hora de considerar este enfoque es saber si el productor tiene suficiente información refecto a como se procesan los mensajes para tomar la decisión sobre a que destino enrutar el mensaje. Cuanto más información necesita el productor sobre la consumición del mensaje, mayor será el acoplamiento entre productor y consumidor.

Por último, para beneficiarnos de ámbos enfoques, también se puede hacer un uso de un enfoque combinado:



Enfoque de Múltiples Destinos

3.3. Browser de Mensajes

En la sesión anterior vimos como recibir mensajes de una cola, pero supongamos que solo queremos consultar el contenido de la cola sin *consumir los mensajes*. Para este propósito, en vez de utilizar un `MessageConsumer`, crearemos un `QueueBrowser`.

Para crear un objeto de este tipo, utilizaremos el método `createBrowser` de la `Session`, indicándole la cola a inspeccionar:

```
QueueBrowser browser = session.createBrowser(queue);
```

También podemos crear un *browser* que filtre los mensajes en los que estamos interesados, pasándole como segundo argumento el selector de mensajes:

```
QueueBrowser browser = session.createBrowser(queue, "provincia =
'Alicante'");
```

Browser en PTP

Destacar que el *browser* de mensajes **solo** se puede aplicar sobre **colas**, no sobre tópicos, ya que los mensajes en los tópicos desaparecen tan pronto como aparecen: si no hay ningun subscriptor para consumir el mensaje, el proveedor JMS lo elimina.

La interfaz `QueueBrowser` contempla unos cuantos métodos a destacar:

- `getEnumeration()` para obtener los mensajes
- `getMessageSelector()` para obtener el `String` que actúa como filtro de mensajes (si es que hay alguno para ese consumidor)
- `getQueue()` que devuelve la cola asociada al browser.

Así pues, una vez creado el *browser*, obtenemos una enumeración para iterar sobre ella e ir obteniendo las propiedades del mensaje, tales como la prioridad, el identificador, etc...

```
Enumeration iter = browser.getEnumeration();
if (!iter.hasMoreElements()) {
        System.out.println("No hay mensajes en la cola");
} else {
        while (iter.hasMoreElements()) {
                Message tempMsg = (Message) iter.nextElement();
                System.out.println("Mensaje: " + tempMsg);
        }
}
```

Con el *browser* obtenemos el estado actual de una cola y nos permite visualizar aquellos mensajes que no han sido recogidos por una mala sincronización. El resultado de una consulta como la anterior sería algo así:

```
Mensaje:
Text:        Hola Mundo
Class:
com.sun.messaging.jmq.jmsclient.TextMessageImpl
```

```
getJMSMessageID():
ID:41-172.18.33.103(d6:f1:b9:3e:e:4f)-52900-1229023964087
getJMSTimestamp():        1229023964087
getJMSCorrelationID():        null
JMSReplyTo:                null
JMSDestination:                jmsQueue
getJMSDeliveryMode():        PERSISTENT
getJMSRedelivered():        false
getJMSType():                null
getJMSExpiration():        0
getJMSPriority():        4
Properties:                {Anyo=2008}
```

Podemos observar como se muestra el contenido del cuerpo, el tipo de mensajes, todas las cabeceras, así como aquellas cabeceras con valor asignado. Por ejemplo, podemos ver como aparece la propiedad que hemos añadido en la sección anterior.

3.4. JMS Robusto

El API de JMS está destinado a garantizar la robustez, y al mismo tiempo (si se puede) eficiencia de nuestras aplicaciones distribuidas. En este sentido, es esencial no tolerar situaciones en donde los mensajes no llegan o bien llegan duplicados, lo que incide muy negativamente en la sincronización de la componentes software.

Por eso, JMS garantiza que:

- El mensaje será recibido
- Y solamente será recibido una vez

Implementar esta funcionalidad por parte del servidor de aplicaciones implica, por ejemplo, dotar la infraestructura JMS con una capa de persistencia. De hecho, la forma más fiable de **producir** un mensaje es enviar un mensaje **PERSISTENT** (el modo de creación de mensajes por defecto) dentro de una transacción. También, la forma más robusta de **consumir** un mensaje es dentro de una **transacción** y a partir de una cola o de un *Durable Subscriber* a un tópico.

Sin embargo, tomar las medidas anteriores supone un elevado consumo de recursos y por lo tanto una caída del rendimiento. Así, y como veremos a continuación, se pueden conseguir soluciones menos robustas manipulando el nivel de prioridad de los mensajes (JMS intenta transmitir primero los mensajes con mayor prioridad aunque eso no se garantiza en la práctica) o controlar el tiempo de vida de los mensajes (si se fija a cero eso indica que no caduca nunca). En un caso extremo, para aliviar la carga de la capa de persistencia se pueden enviar mensajes NON_PERSISTENT lo cual libera a JMS de garantizar dicha persistencia si el proveedor de mensajes falla. Por último, también podemos crear colas temporales, que se mantienen mientras dura la conexión, pero con ellas se corre el riesgo de perder los mensajes si hay fallos. Todas estas medidas deberán tomarse solo si la aplicación puede permitírselas mientras garantiza su funcionalidad.

A continuación estudiaremos los diferentes mecanismos para conseguir o que afectan a la

entrega fiable de los mensajes.

3.4.1. Controlar el Acuse de Recibo de los Mensajes

Hasta que no se recibe un acuse de recibo de un mensaje, éste no se considera consumido exitosamente. La consumición exitosa de un mensaje se lleva a cargo en tres fases:

1. El cliente recibe el mensaje
2. El cliente procesa mensaje
3. Se acusa el recibo del mensaje. Este acuse se inicia por parte del proveedor JMS o por el cliente, dependiendo del modo de acuse de la sesión.

En las sesiones transaccionales (mediante las transacciones locales), el acuse ocurre automáticamente cuando se hace el *commit* de la transacción. Si se produce un *rollback* de la transacción, todo los mensajes consumidos se vuelven a re-entregar.

En las sesiones no transaccionales, el cuando y el cómo se acusa depende del valor especificado como segundo argumento del método `createSession`. Los tres posibles valores para este argumento son:

- **Session.AUTO_ACKNOWLEDGE**: La sesión automáticamente realiza el acuse al mensaje de cliente tanto si el cliente ha retornado exitosamente de una llamada `MessageConsumer.receive()`, o cuando el listener de mensaje llamado para procesar el mensaje, `MessageListener.onMessage()`, retorna exitosamente.
  Una recepción síncrona en una sesión AUTO_ACKNOWLEDGE es la excepción a la regla que la consumición del mensaje es un proceso de tres fases como anteriormente comentado.
  En este caso, la recepción y el acuse tienen lugar en un único paso, seguido por el procesamiento del mensaje.
- **Session.CLIENT_ACKNOWLEDGE**: Un cliente acusa un mensaje mediante la llamada al método `acknowledge()` del mensaje. En este modo, el acuse se realiza a nivel de sesión: El acuse de un mensaje consumido automáticamente implica el acuse de la recepción de **todos** los mensajes que se han consumido por dicha sesión.
  Por ejemplo, si un consumidor de mensajes consume diez mensajes y a continuación acusa el quinto mensaje entregado, los diez mensajes son acusados.
- **Session.DUPS_OK_ACKNOWLEDGE**: Esta opción instruye a la sesión a realizar el acuse de recibo de los mensajes de un modo tardío (***lazy***). Esto supone la entrega de mensajes duplicados en el caso de que el proveedor JMS falle, por lo tanto solo debería utilizarse por aquellos consumidores que puede tolerar mensajes duplicados (si el proveedor JMS reenvia un mensaje, debe poner el valor de la cabecera `JMSRedelivered` a `true`).
  Esta opción puede reducir la sobrecarga de la sesión minimizando la cantidad de trabajo que ésta realiza para prevenir duplicados.

Como hemos comentado, especificaremos el tipo de acuse al crear la sesión mediante el *flag* apropiado:

```
TopicSession session = topicConnection.createTopicSeccion(false,
Session.CLIENT_ACKNOWLEDGE);
```

Si se ha recibido un mensaje de una cola pero no se ha realizado el acuse al terminar la sesión, el proveedor JMS los retiene y los re-entrega cuando el consumidor vuelve a acceder a la cola. El proveedor también retiene aquellos mensajes que no han sido acusados para aquellas sesiones realizadas por un `TopicSubscriber`.

En el caso de utilizar una cola o una subscripción duradera, se puede utilizar el método `Session.recover()` para detener una sesión no transaccional y reiniciarla con su primer mensaje sin acuse de recibo realizado. En efecto, las series de sesiones de los mensajes enviados se resetean hasta el punto posterior al último mensaje acusado. El mensaje que ahora se envía puede ser diferente de aquellos que originalmente fueron enviados, si los mensajes han expirado o si han llegado mensajes con una prioridad mayor. Para un `TopicSubscriber` no durable, el proveedor puede perder mensajes sin acusar cuando se recupera su sesión.

3.4.2. Especificar la Persistencia de los Mensajes

JMS soporta dos modos de entrega de mensajes para especificar si los mensajes se pierden al caerse el proveedor. Estos modos de entrega se definen como campos del interfaz `DeliveryMode`:

- El modo **PERSISTENT**, el cual es el modo por defecto, le indica al proveedor JMS a tomar un cuidado extra para asegurar que no se pierde ningún mensaje en el hipotético caso de que fallase el proveedor JMS. Un mensaje enviado con este modo de entrega se almacena es un almacenamiento estable al enviarse.
- El modo **NON_PERSISTENT** no obliga al proveedor a almacenar el mensaje, y por tanto, no garantiza que no se pierda el mensaje en el caso de que falle el proveedor.

Podemos especificar el modo de entrega de dos maneras:

- Mediante el método `MessageProducer.setDeliveryMode()` para indicar el modo de entrega para todos los mensajes enviados por dicho productor. Por ejemplo:

  ```
  producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
  ```

- Mediante el modo largo del método `send()` o `publish()` para cada mensaje de forma individual. El segundo parámetro indica el modo de entrega. Por ejemplo:

  ```
  producer.send(message, DeliveryMode.NON_PERSISTENT, 3, 10000);
  ```

  El tercer y cuarto parámetro indican el nivel de prioridad y el periodo de expiración, los cuales explicaremos en los siguientes apartados.

Como hemos dicho, si no especificamos nada, el modo por defecto es `DeliveryMode.PERSISTENT`. Mediante el modo `DeliveryMode.NON_PERSISTENT` podemos mejorar el rendimiento y reducir la sobrecarga por almacenamiento, pero deberemos cuidarnos de utilizarlo solo en aquellas aplicaciones que puedan permitirse la

pérdida de mensajes.

3.4.3. Indicar el Nivel de Prioridad

Podemos utilizar diferentes niveles de prioridad para que el proveedor JMS envíe primero los mensajes más urgentes. Podemos indicar este nivel de dos maneras:

- Mediante el método `MessageProducer.setPriority()` para indicar el nivel de prioridad de todos los mensajes enviados por dicho productor. Por ejemplo:

```
producer.setPriority(7);
```

- Mediante el modo largo del método `send()` o `publish()` para cada mensaje de forma individual. El tercer parámetro indica el nivel de prioridad de entrega. Por ejemplo:

```
producer.send(message, DeliveryMode.NON_PERSISTENT, 3, 10000);
```

Los diez niveles de prioridad van desde el 0 (el más bajo) al 9 (el más alto). Si no se especifica ningún nivel, el nivel por defecto es 4. El proveedor JMS debería entregar los mensajes de mayor prioridad antes que los de menor prioridad, pero esto no implica que lo tenga que entregar en el orden exacto de prioridad.

3.4.4. Permitir la Expiración de los Mensajes

Por defecto, un mensaje nunca expira. En el caso de necesitar que un mensaje se vuelva obsoleto tras pasar un cierto periodo de tiempo, podemos indicar un periodo de expiración. Podemos hacerlo de dos maneras:

- Mediante el método `MessageProducer.setTimeToLive()` para indicar un periodo de expiración por defecto para todos los mensajes enviados por dicho productor. Por ejemplo:

```
producer.setTimeToLive(60000); // 1 min
```

- Mediante el modo largo del método `send()` o `publish()` para cada mensaje de forma individual. El cuarto parámetro indica el tiempo de vida (`timeToLive`) en milisegundos. Por ejemplo:

```
producer.send(message, DeliveryMode.NON_PERSISTENT, 3, 10000); //
10 seg
```

Si el valor especificado como `timeToLive` es 0, el mensaje nunca expira.

Cuando se envía el mensaje, el periodo especificado como `timeToLive` se añade al tiempo actual para calcular el periodo de expiración. Cualquier mensaje que no se ha enviado antes del periodo de expiración especificado será destruido. La destrucción de los mensajes obsoletos preserva el almacenamiento y los recursos computaciones.

3.4.5. Crear Destinos Temporales

Por norma general, crearemos los destinos JMS (colas y tópicos) de modo administrativo más que programativo. Todo proveedor JMS incluye una herramienta de administración para crear y eliminar destinos, y estos destinos, una vez creados, no suelen borrarse.

JMS también permite crear destinos (`TemporaryQueue` y `TemporaryTopic`) que duran lo que dura la conexión que los ha creado. Así pues, crearemos estos destinos de forma dinámica mediante los métodos `Session.createTemporaryQueue()` y `Session.createTemporaryTopic()`.

Los únicos consumidores de mensajes que pueden consumir de un destino temporal son aquellos creados por la misma conexión que creó los destinos. Cualquier productor de mensajes puede enviar a un destino temporal. Si cerramos la conexión a la que pertenece un destino temporal, el destino se cierra y se pierden sus contenidos.

Podemos utilizar destinos temporales para implementar un mecanismo sencillo de *request/reply*.

3.5. Simulando Llamadas Síncronas Mediante Request/Reply

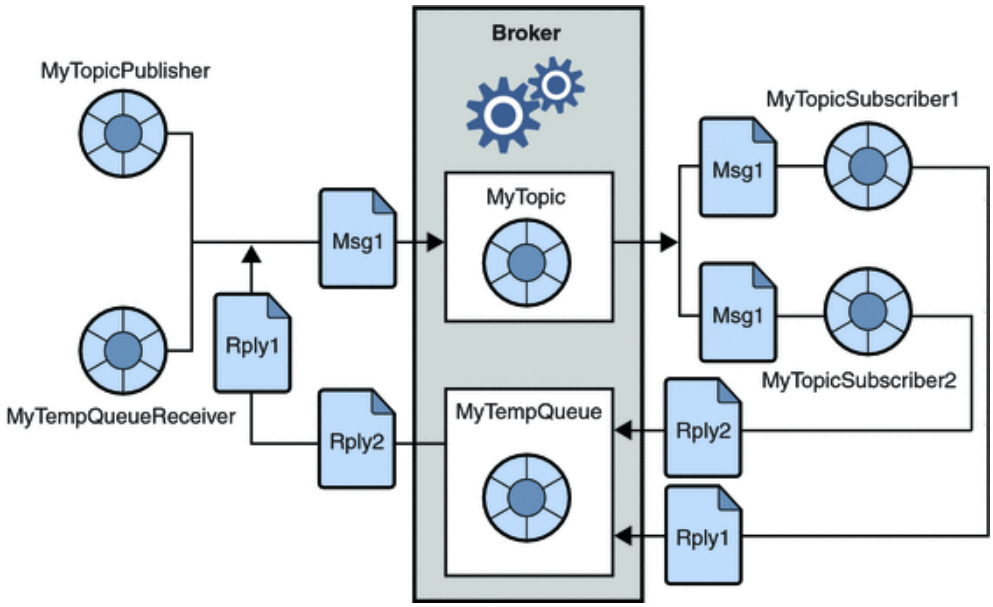
En ciertos casos puede venirnos bien implementar un mecanismo de *petición y respuesta* de tal forma que quedemos a la espera de que la petición se satisfaga. Así pues, mediante este mecanismo simulamos una comunicación síncrona mediante un sistema asíncrono.

Los pasos que necesita un cliente para simular una comunicación síncrona con un dominio punto a punto son:

1. Crear una cola temporal mediante el método `createTemporaryQueue()` de la sesión de la cola
2. Asignar a la cabecera `JMSReplyTo` esta cola temporal (de este modo, el consumidor del mensaje utilizará este cabecera para conocer el destino al que enviar la respuesta).
3. Enviar el mensaje.
4. Ejecutar un receive con bloqueo en la cola temporal, mediante una llamada al método `receive` en la cola del receptor para esta cola temporal

El consumidor también puede referenciar a la petición original asignando a la cabecera `JMSCorrelationID` del mensaje de respuesta con el valor de la cabecera `JMSMessageID` de la petición.

A continuación se muestra un diagrama del mecanismo *request/reply* con el dominio Pub/Sub:



Request/Reply con Pub/Sub

Para reducir la cantidad de trabajo que tienen que hacer los clientes, JMS ofrece una pareja de clases auxiliares (`QueueRequestor` and `TopicRequestor`) que gestionan toda la configuración requerida por los anteriores pasos. Un *requestor* es un objeto que convierte un intercambio de mensajes en una especie de llamada a procedimiento remoto. Existen tanto `QueueRequestor` como `TopicRequestor` aunque este tipo de objetos se usan fundamentalmente en el modelo punto-a-punto, esto es con colas, en donde existe un solo receptor de la petición, que responde al *request*. Si se utilizase el modelo de subscripción solo la respuesta del primer subscriptor que responde se tendría en cuenta por el *requestor* y el resto de ellas se perdería.

3.6. Subscripciones Duraderas

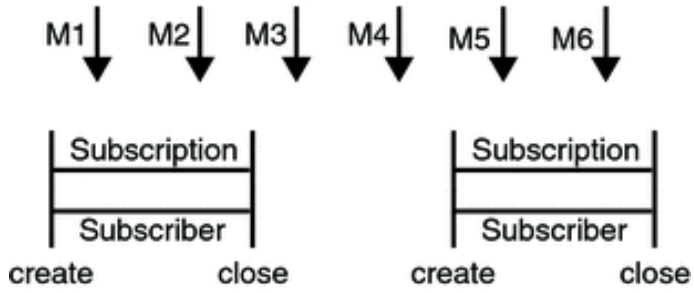
Para estar seguro de que una aplicación Pub/sub recibe todos los mensajes publicados, debe usarse el modo de envío `PERSISTENT` en la construcción del mensaje. Pero adicionalmente, en este caso puede usarse un *durable subscriber* para los subscriptores.

La idea básica de un subscriptor duradero es que una subscripción de este tipo permanece activa a pesar de que el subscriptor cierre su conexión, de modo que recibirá los mensajes publicados cuando el subscriptor no estaba activo. Este mecanismo ofrece la fiabilidad de las colas dentro del dominio pub/sub.

3.6.1. Durable vs Non Durable

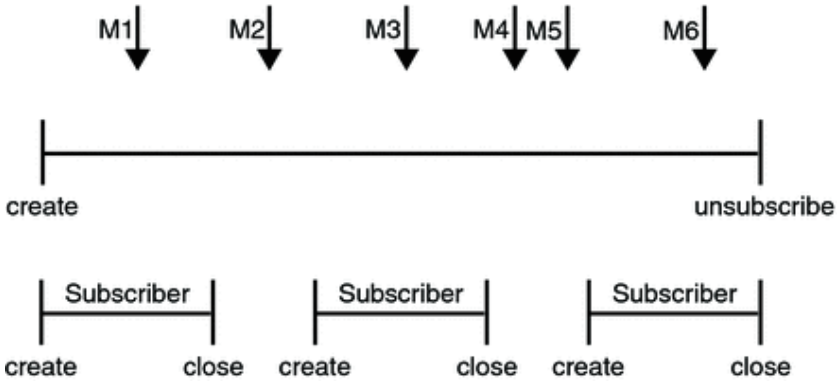
Mediante un subscriptor ordinario, no duradero, tanto el subscriptor como la subscripción comienzan y finalizan al mismo tiempo, y por tanto, su vida es idéntica. Al cerrar un

subscriptor, la subscripción también finaliza. En el gráfico, *create* referencia una llamada a `Session.createConsumer()` pasándole un tópico como parámetro, y *close* a una llamada a `MessageConsumer.close()`. Cualquier mensaje publicado al tópico entre el primer *close* y el segundo *create* no llega a consumirse por ningún subscriptor, de modo que los mensajes M3 y M4 se pierden.



Non-durable Subscribers

Con una subscripción duradera, el subscriptor puede cerrarse y recrearse, pero la subscripción sigue existiendo y mantiene los mensajes hasta que la aplicación realiza una llamada al método *unsubscribe*. En el gráfico, *create* referencia una llamada a `Session.createDurableSubscriber()`, *close* a una llamada a `MessageConsumer.close()` y *unsubscribe* a `Session.unsubscribe()`. Los mensajes publicados mientras el subscriptor esta inactivo se reciben al volver a crear el subscriptor. De este modo, incluso los mensajes M2, M4 y M5 llegan al destino, y por tanto no se pierden.



Durable Subscribers

3.6.2. Creación y Manejo

A coste de incrementar la carga del servidor, podemos utilizar el método `Session.createDurableSubscriber` para crear un subscriptor durable, con la restricción de que una subscripción de este tipo solo puede tener un subscriptor activo a la vez.

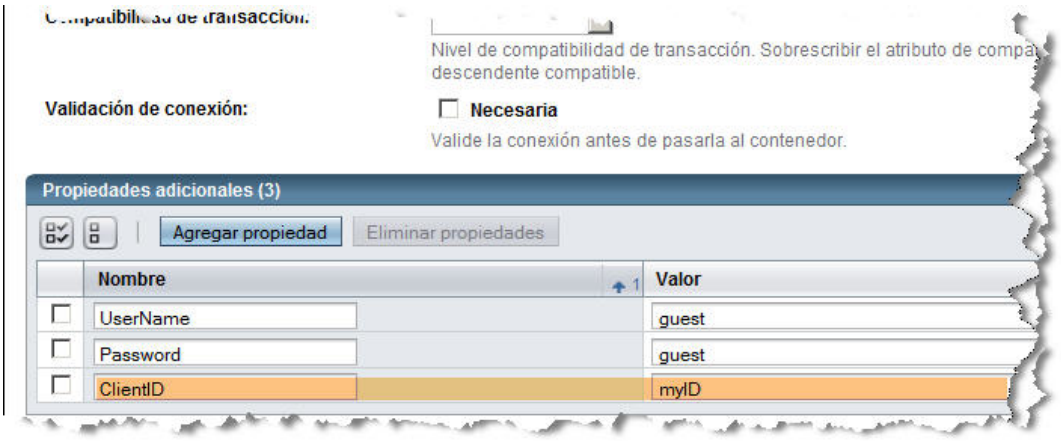
Un subscritor duradero registra una subscripción duradera especificando una identidad única la cual retiene el proveedor JMS. Los siguientes subscriptores que tengan la misma identidad continúan con la subscripción en el mismo estado en el que quedó la subscripción una vez finalizó el subscritor previo. Si una subscripción duradera no tiene ningún subscriptor activo, el proveedor JMS retiene los mensajes hasta que alguien los consume o que expiran.

Para establecer la identidad única de un subscriptor indicaremos:

- Un ID de cliente para la conexión
- Un nombre de tópico y de subscripción para el subscriptor

Para indicar el ID de cliente, lo podemos hacerde forma administrativa para una factoría de conexiones específica de un cliente. Supongamos que llamamos a esa *factoría especial* `jms/DurableConnectionFactory`. La crearemos de la forma habitual desde el servidor de aplicaciones asignándole como tipo `TopicConnectionFactory` y añadiendo en la parte inferior de la ventana una *propiedad* llamada `ClientID` con valor `myID`. Dicha propiedad define la identidad:

Definición de la factoría



Definición de la identidad del cliente

Tras crear la factoría de conexiones para crear la conexión y la sesión, utilizaremos el método `createDurableSubscriber()` con dos parámetros, el tópico y el nombre de la subscripción:

```
String nombreSub = "MiSub";
MessageConsumer topicSubscriber =
session.createDurableSubscriber(topic, nombreSub);
```

El subscriptor se activa tras iniciar la conexión. Si queremos cerrar el subscriptor, llamaremos a:

```
topicSubscriber.close();
```

Para eliminar una subscripción duradera, primero cerraremos el subscriptor, y posteriormente mediante el método `unsuscribe()` (pasándole como parámetro el nombre de la subscripción) eliminamos la subscripción y el estado que mantiene el proveedor para el subscriptor:

```
topicSubscriber.close();
session.unsubscribe(nombreSub);
```

3.6.3. Ejemplo

Veamos como funciona el mecanismo de la subscripción duradera en siguiente ejemplo. El código del ejemplo se encuentra en DurableSubscriberExample.java.

Desde el `main()` se invoca ya al método `runProgram()` que crea, por un lado, un objeto llamado `DurableSubscriber` que se encarga de la gestión del subscriptor duradero (creación, inicio, finalización y sincronización) y, por otro lado, un objeto `MultiplePublisher` que publica varios mensajes en un tópico, pero para ello usa la factoría no duradera. Sin embargo, en ese tópico estará escuchando un durable subscriber. La secuencia es la siguiente:

```
public void runProgram() {
```

```
    // Instanciamos el subscriptor y el publicador
    DurableSubscriber durableSubscriber = new DurableSubscriber();
    MultiplePublisher multiplePublisher = new MultiplePublisher();
    // Iniciamos el subscriptor, publicamos mensajes, y cerramos el
subscriptor
    durableSubscriber.startSubscriber();
    multiplePublisher.publishMessages();
    durableSubscriber.closeSubscriber();
    // Mientras que el subscriptor esta cerrado, el publicador
publica algunos mensajes más
    multiplePublisher.publishMessages();
    // Reiniciamos el subscriptor y recoge estos mensajes
    durableSubscriber.startSubscriber();
    // Cerramos las conexiones
    durableSubscriber.closeSubscriber();
    multiplePublisher.finish();
    durableSubscriber.finish();
}
```

El constructor de `DurableSubscriber()` se encarga de establecer la conexión con la factoría duradera y de iniciar la sesión. Lo mismo ocurre con el constructor de `MultiplePublisher` a través de una factoría normal.

Posteriormente, `startSubscriber()` detiene la conexión, crea un *durable subscriber*, indicando como segundo argumento el nombre de la subscripción duradera (en nuestro ejemplo *'MiSubscripcionDuradera'*). Este nombre se concatena internamente con el identificador del cliente del siguiente modo: `ClientID + "##" + nombreSubscripcion`. A continuación se crea un listener de mensajes de texto y se vincula a dicho listener (recordad que el método `onMessage()` del listener captura de forma asíncrona los mensajes que se envíen al tópico). Finalmente se reanuda la conexión.

```
public void startSubscriber() {
    try {
        System.out.println("Iniciando subscriptor con
MiSubscripcionDuradera");
        connection.stop();
        subscriber = session.createDurableSubscriber(topic,
"MiSubscripcionDuradera");
        listener = new TextListener();
        subscriber.setMessageListener(listener);
        connection.start();
    } catch (JMSException e) {
        System.err.println("startSubscriber -> Exception: " +
e.toString());
    }
}
```

Posteriormente, el método `publishMessages()` publica tres mensajes en el tópico en que está escuchando el subscriptor duradero.

```
public void publishMessages() {
    TextMessage message = null;
    int i;
    final int NUMMSGS = 3;
```

```
    final String MSG_TEXT = "Este es el mensaje num";

    try {
        message = session.createTextMessage();

        for (i = startindex; i < (startindex + NUMMSGS); i++) {
            message.setText(MSG_TEXT + " " + (i + 1));
            System.out.println(" PUBLICADOR -> Publicando mensaje:
"
                + message.getText());
            producer.send(message);
        }

        // Envia un mensaje de control indicando el fin de los
mensajes
        producer.send(session.createMessage());
        startindex = i;
    } catch (JMSException e) {
        System.err.println("publishMessages -> Exception : " +
e.toString());
    }
}
```

Esos mensajes son leídos por dicho subscriptor, mediante el listener comentado anteriormente:

```
public void onMessage(Message message) {
    if (message instanceof TextMessage) {
        TextMessage msg = (TextMessage) message;

        try {
            System.out.println(" SUBSCRIPTOR -> Leyendo mensaje: "
+ msg.getText());
        } catch (JMSException e) {
            System.err.println("Exception en onMessage(): " +
e.toString());
        }
    } else {
        monitor.allDone();
    }
}
```

Posteriormente el subscriptor se cierra con el método `closeSubscriber()`:

```
public void closeSubscriber() {
    try {
        listener.monitor.waitTillDone();
        System.out.println("Cerrando subscriptor");
        subscriber.close();
    } catch (JMSException e) {
        System.err.println("closeSubscriber -> Exception: " +
e.toString());
    }
}
```

Mientras el subscriptor duradero está cerrado, el productor envía 3 mensajes más. Dichos mensajes son recogidos por el subscriptor cuando vuelve a lanzar el método

`startSubscriber()`. Después se vuelve a cerrar y finalmente el publicador cierra la conexión en su método `finish()` y el subscriptor duradero cierra en su `finish()` la conexión y luego abandona la subscripción (debemos especificar su nombre):

```
// Finish del Durable Subscriber
public void finish() {
    if (connection != null) {
        try {
            System.out.println("Unsuscribe de la subscripcion
duradera");
            session.unsubscribe("MiSubscripcionDuradera");
            connection.close();
        } catch (JMSException ee) {}
    }
}
```

El resultado de la ejecución es el siguiente:

```
Factoria de conexiones con ID Cliente es
jms/DurableConnectionFactory
Nombre del topico es jms/Topic
-----------------------------------
Iniciando subscriptor con MiSubscripcionDuradera
 PUBLICADOR -> Publicando mensaje: Este es el mensaje num 1
 PUBLICADOR -> Publicando mensaje: Este es el mensaje num 2
 SUBSCRIPTOR -> Leyendo mensaje: Este es el mensaje num 1
 PUBLICADOR -> Publicando mensaje: Este es el mensaje num 3
 SUBSCRIPTOR -> Leyendo mensaje: Este es el mensaje num 2
 SUBSCRIPTOR -> Leyendo mensaje: Este es el mensaje num 3
Cerrando subscriptor
 PUBLICADOR -> Publicando mensaje: Este es el mensaje num 4
 PUBLICADOR -> Publicando mensaje: Este es el mensaje num 5
 PUBLICADOR -> Publicando mensaje: Este es el mensaje num 6
Iniciando subscriptor con MiSubscripcionDuradera
 SUBSCRIPTOR -> Leyendo mensaje: Este es el mensaje num 4
 SUBSCRIPTOR -> Leyendo mensaje: Este es el mensaje num 5
 SUBSCRIPTOR -> Leyendo mensaje: Este es el mensaje num 6
Cerrando subscriptor
Unsuscribe de la subscripcion duradera
```

4. Ejercicios de Mensajes y Robustez en JMS

Todos los ejercicios se basan en el proyecto del Semáforo Hipotecario, y en los casos que sea posible, se trabajará sobre los proyectos de la sesión anterior.

4.1. Propiedades y Selectores (0.5p)

Queremos que nuestro semáforo sólo analice aquellas hipotecas que provengan de cajas de ahorro, ignorando a los bancos. Para ello, el productor de mensajes deberá añadir la propiedad `BancoCaja` con el valor `Banco` o `Caja`.

A su vez, el semáforo, mediante un selector de mensajes, obtendrá solo aquellos mensajes donde la propiedad `BancoCaja = Banco`.

Tu decides como implementarlo, ya sea añadiendo un nuevo argumento al método de envio con el tipo de entidad, o bien creando un nuevo método y que cada método defina la propiedad con el valor adecuado.

Pregunta

Y si el semáforo sólo aceptase las peticiones de los bancos BBVA y Banesto, ¿qué habría que hacer?

4.2. Datos del Préstamo en Objeto (0.75p)

Crear una clase `DatosHipoteca` (la cual debe ser `Serializable`) que contenga los datos de entrada al semáforo, de modo que el *listener* permita tratar tanto con mensajes de tipo `TextMessage` como `ObjectMessage`.

Para mejorar la legibilidad del código del listener, añade un método a `BibliotecaBR` para que desde el listener pueda obtener el color con una llamada a `BibliotecaBR.getInstance().obtenerColorHipoteca(datos)`, siendo `datos` una instancia de `DatosHipoteca`.

Por cuestiones internas de Netbeans, no podemos crear un proyecto con esta clase y que la compartan tanto el banco como el semaforo. Por tanto, deberás duplicar dicha clase.

4.3. Caducidad de los Mensajes (0.75p)

Nos informan que los tipos de interes del semáforo van a cambiar cada día. Por ello, para no sobrecargar el sistema con peticiones obsoletas, queremos que el productor fije un tiempo máximo de modo que los mensajes caduquen si no se han consumido durante el día de envío del mensaje.

4.4. Browser del Semáforo (0.5p)

Crear un browser que permita visualizar los mensajes que el semáforo tiene pendiente. El nombre del proyecto será `jms-browser-hipotecario`.

5. Transacciones. JMS y JavaEE

5.1. Transacciones Locales

En un cliente JMS se pueden usar transacciones locales para agrupar bien envíos o bien recepciones en operaciones atómicas. Ya desde el principio hemos visto que una sesión se puede crear como transaccional o no solamente con poner respectivamente a `true` o `false` el primer argumento del método `Connection.createSession()` y hasta ahora solamente hemos trabajado con sesiones no transaccionales. De modo, que mediante los siguientes fragmentos de código crearíamos una sesión transaccional:

```
Session session = connection.createSession(true, 0);
QueueSession queueSession =
queueConnection.createQueueSession(true, 0);
TopicSession topicSession =
topicConnection.createTopicSession(true, 0);
```

Buenas Prácticas

Es una buena práctica poner a 0 el segundo argumento del método (tipo de acuse de recibo), ya que al tratarse de una sesión transaccional, será ignorado.
Recordar que el valor 0 es similar a Session.AUTO_ACKNOWLEDGE

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

JMS no ofrece ningún método de inicio de transacción de forma explícita, de modo que nada más crear la sesión transaccional, la transacción ha comenzado.

Además, JMS aporta los métodos `Session.commit()` y `Session.rollback()` que pueden usarse en un cliente, y que supondrán el final de la transacción. El *commit* significa que *todos los mensajes producidos son enviados y se envía acuse de recibo de todos los consumidos*. Sin embargo, el *rollback* en JMS implica que *se destruyen todos los mensajes enviados y se recuperan todos los mensajes consumidos y re-enviados aunque hayan expirado*.

Toda transacción forma parte de una sesión transaccional. Tan pronto como se llama a `commit` o `rollback`, finaliza una transacción y comienza otra (esto se conoce como *transaction chaining*). Cerrar una sesión transaccional implica un rollback automático de la transacción, incluyendo los envíos y recepciones pendientes.

Finalmente, los métodos anteriores no pueden usarse en EJBs ya que, como veremos en el módulo correspondiente, se usan transacciones distribuidas.

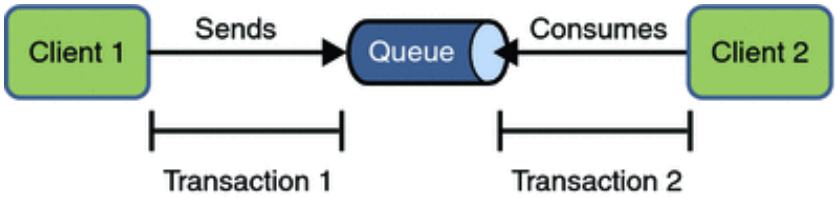
Un aspecto muy importante es que podemos combinar varios envíos y recepciones en una transacción local (no distribuída), pero en ese caso *debemos tener en cuenta el orden de las operaciones*. Si una transacción consiste solo en *sends*, solo en *receives* o la recepción se realiza antes de enviar no hay problema, pero si intentamos usar un mecanismo

request/reply entonces siempre que enviemos un mensaje y esperemos recibirlo dentro de la misma transacción el programa se colgará, ya que el envío no se hace efectivo hasta que no se hace un commit.

```
// No hacer esto
outMsg.setJMSReplyTo(replyQueue);
producer.send(outQueue, outMsg);
consumer = session.createConsumer(replyQueue);
inMsg = consumer.receive();
session.commit();
```

Esto se debe a que al enviar un mensaje dentro de una transacción, realmente no se envía hasta que no se realiza el commit de la transacción, por lo tanto, **la transacción no puede contener ninguna recepción que dependa de un mensaje enviado previamente**.

Es más, la producción y el consumo de un mensaje no puede ser parte de la misma transacción ya que el intermediario es JMS, el cual interviene entre la producción y la consumición del mensaje. En ese caso debemos hacer una transacción desde el productor al recurso JMS y otra desde éste al consumidor. Analicemos el siguiente gráfico:



JMS como intermediario

El envío de uno o más mensajes a uno o más destinos por parte del cliente 1 puede formar una transacción (*producción transaccional*), ya que conforma un conjunto de interacciones con el proveedor JMS mediante una única sesión. Del mismo modo, la recepción de uno o más mensajes (*consumición transaccional*) desde uno o más destinos por parte del cliente 2 también forma una única transacción, y que se realiza desde una sola sesión. Pero como ambos clientes no tienen interacciones directas y están utilizando dos sesiones diferentes, no se puede producir una transacción entre ellos.

En Resumen

El hecho de producir y/o consumir mensajes dentro de **una sesión puede ser transaccional**, pero el hecho de producir y consumir un mensaje específico entre **diferentes sesiones no puede ser transaccional**.

5.1.1. Ejemplos

Como hemos comentado, los métodos de *commit* y *rollback* de una transacción local están asociados con la sesión. Podemos combinar operaciones con colas y tópicos dentro de una única transacción siempre y cuando utilicemos la misma sesión. Por ejemplo, podemos utilizar la misma sesión para recibir un mensaje desde una cola y enviar un

mensaje a un tópico dentro de la misma transacción:

```
public void recibirSincronoPublicarCommit() throws JMSException {
    Connection connection = null;
    Session session = null;
    QueueReceiver receiver = null;
    TopicPublisher publisher = null;

    try {
        connection = connectionFactory.createConnection();
        connection.start();
        // Creamos una sesion transaccional
        session = connection.createSession(true, 0);

        // Creamos el consumidor a partir de una cola
        receiver = (QueueReceiver) session.createConsumer(queue);
        // Creamos el productor a partir de un topico
        publisher = (TopicPublisher) session.createProducer(topic);

        // Consumimos y luego publicamos
        TextMessage message = (TextMessage) receiver.receive();
        System.out.println("Recibido mensaje [" + message.getText()
+ "]");
        publisher.publish(message);

        session.commit();
    } catch (JMSException jmse) {
        System.err.println("Rollback por " + jmse.getMessage());
        session.rollback();
    } catch (Exception e) {
        System.err.println("Rollback por " + e.getMessage());
        session.rollback();
    } finally {
        publisher.close();
        receiver.close();
        session.close();
        connection.close();
    }
}
```

Además, podemos pasar la sesión de un cliente al constructor de un listener de mensajes y utilizarla para crear un productor de mensajes. De este modo, podemos utilizar la misma sesión para recibir y enviar mensajes dentro de un consumidor de mensajes asíncrono.

```
public void recibirAsincronoPublicarCommit() throws JMSException {
    Connection connection = null;
    Session session = null;
    QueueReceiver receiver = null;
    TextListener listener = null;

    try {
        connection = connectionFactory.createConnection();
        // Creamos una sesion transaccional
        session = connection.createSession(true, 0);

        // Creamos el consumidor a partir de una cola
        receiver = (QueueReceiver) session.createConsumer(queue);
```

```
        listener = new TextListener(session);
        receiver.setMessageListener(listener);
        // Llamamos a start() para empezar a consumir
        connection.start();
        // Sacamos el mensaje por consola
        System.out.println("Fin asincrono");
    } catch (JMSException jmse) {
        System.err.println("Rollback por " + jmse.getMessage());
        session.rollback();
    } catch (Exception e) {
        System.err.println("Rollback por " + e.getMessage());
        session.rollback();
    } finally {
        receiver.close();
        session.close();
        connection.close();
    }
}
```

De modo que el listener puede hacer commit o rollback conforme necesite:

```
private class TextListener implements MessageListener {

    private Session session;

    public TextListener(Session session) {
        this.session = session;
    }

    public void onMessage(Message message) {
        TopicPublisher publisher = null;
        TextMessage msg = null;

        // Consumimos y luego publicamos
        try {
            msg = (TextMessage) message;
            System.out.println("Recibido mensaje asincrono [" +
msg.getText() + "]");
            publisher = (TopicPublisher)
session.createProducer(topic);
            publisher.publish(message);

            session.commit();
        } catch (JMSException e) {
            System.err.println("Rollback en onMessage(): " +
e.toString());
            try {
                session.rollback();
            } catch (JMSException ex) {
            }
        }
    }
}
```

5.2. Transacciones Distribuidas

Los sistemas distribuidos en ocasiones utilizan un proceso de *two-phase commit* (2PC)

que permite a múltiples recursos distribuidos participar en una transacción. Esto implica el uso de un gestor de transacciones que se encarga de cordinar la preparación, commit o rollback de cada recurso que participa en la transacción. Lo más común es que estos recursos sean BBDD, pero también pueden ser proveedores de mensajes.

El proceso de 2PC se realiza bajo el interfaz XA (*eXtended Architecture*), y en JavaEE lo implementa JTA (*Java Transaction API*) y los interfaces XA (`javax.transaction` y `javax.transaction.xa`). Cualquier recurso que implementa estos interfaces puede unirse a una transacción global mediante un gestor de transacciones que soporte estos interfaces.

Los proveedor JMS que implementan los interfaces XA puede participar en transacciones distribuidas. La especificación JMS ofrece versiones XA de los siguientes objetos:`XAConnectionFactory`, `XAQueueConnection`, `XAQueueConnectionFactory`, `XAQueueSession`, `XASession`, `XATopicConnectionFactory`, `XATopicConnection` y `XATopicSession`.

Cada uno de estos objetos trabajar de modo similar a los no-XA. El gestor de transacciones de un servidor de aplicaciones utiliza los interfaces XA directamente, pero el cliente JMS solo ve las versiones no-transaccionales. Así pues, los interfaces XA no estan pensados para que lo utilicen los desarrolladores, sino que los provedores JMS son los que deben implementarlos. En resumen, no debemos preocuparnos en usar estos interfaces, sólo de si nuestro servidor de aplicaciones soporta 2PC, y éste ya se encargará de incluir nuestra operación dentro de una transacción global.

5.3. Conexiones Perdidas

Cuando lo conexión de red entre el cliente y el servidor se pierde, el proveedor JMS intentará restablecer la conexión. Si el proveedor no consiguiese la reconexión, debe notificar al cliente de la situación, mediante el lanzamiento de una excepción.

El problema viene cuando tenemos un consumidor asíncrono, el cual no realiza ninguna llamada de envío o recepción. Este consumidor no esta invocando ningún método JMS, sólo está escuchando mensajes, por lo que puede no llegar a detectar la pérdida de la conexión.

JMS ofrece la interfaz `ExceptionListener` para capturar todas las conexiones perdidas y notificar a los clientes de dicha situación. Este listener se asocia a la conexión. La defínción del listener es la siguiente:

```
public interface ExceptionListener {
   void onException(JMSException exception);
}
```

El proveedor JMS se responsabilizará de llamar a este método de todos los listeners registrados cuando no pueda realizar la reconexión automática. Por lo tanto, el consumidor asíncrono podrá implementar este interfaz para poder actuar en esta situación,

e intentar la reconexión de modo manual.

5.3.1. Ejemplo

Por ejemplo, podremos modificar nuestro consumidor para que en el caso de perder la conexión, realice una reconexión manual:

```
public class ConsumidorAsincrono implements ExceptionListener {

    @Resource(mappedName = "jms/ConnectionFactory")
    private static ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/Queue")
    private static Queue queue;

    private Connection connection = null;

    private void estableceConexion() {
        try {
            connection = connectionFactory.createConnection();
            connection.setExceptionListener(this);
        } catch (JMSException ex) {
            ex.printStackTrace(System.err);
        }
    }

    @Override
    public void onException(JMSException exception) {
        System.err.println("Ha ocurrido un error con la conexion");
        exception.printStackTrace(System.err);

        this.estableceConexion();
    }

    public void recibeMensajeAsincronoCola() throws JMSException {
        Session session = null;
        MessageConsumer consumer = null;
        TextoListener listener = null;

        try {
            this.estableceConexion();
            session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);
            consumer = session.createConsumer(queue);

            listener = new TextoListener();
            consumer.setMessageListener(listener);

            connection.start();
        } finally {
            consumer.close();
            session.close();
            connection.close();
        }
    }

    public static void main(String[] args) throws Exception {
        ConsumidorAsincrono p = new ConsumidorAsincrono();
```

```
            p.recibeMensajeAsincronoCola();
        }
}
```

5.4. Uso de JMS en Aplicaciones JavaEE

En esta sección veremos las diferentes maneras de usar JMS desde una aplicación JavaEE, y sus diferencias respecto a una aplicación cliente.

Importante

Los componentes web y EJBs no deben crear más de una sesión activa (sin cerrar) por conexión

5.4.1. Anotaciones en Componentes JavaEE

Cuando utilizamos la anotación `@Resource` en una aplicación cliente, normalmente la declaramos como un recurso estático:

```
@Resource(mappedName="jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;

@Resource(mappedName="jms/Queue")
private static Queue queue;
```

Sin embargo, cuando utilizamos estas anotaciones en un ejb de sesión, un MDB o un componente web, **no** declararemos el recurso estático:

```
@Resource(mappedName="jms/ConnectionFactory")
private ConnectionFactory connectionFactory;

@Resource(mappedName="jms/Topic")
private Topic topic;
```

Si lo declaramos estáticos, obtendremos errores de tiempo de ejecución.

5.4.2. EJBs de Sesión para Producir y Recibir Mensajes Síncronos

Una aplicación JavaEE que produce mensajes o que los consume de manera síncrona *puede* utilizar un EJB de sesión para realizar estas tareas. Pero como la consumición dentro de un bloque síncrono reduce los recursos del servidor, no es una buena práctica realizar un `receive` dentro de un EJB. Por ello, se suele utilizar un `receive` con timeout, o un MDB para recibir los mensajes de una manera asíncrona.

El uso de JMS dentro de una aplicación JavaEE es muy similar al realizado en una aplicación cliente, salvando las diferencias en cuanto a la gestión de los recursos y las transacciones.

5.4.2.1. Gestión de Recursos

Los recursos JMS son la conexión y la sesión. Normalmente, es importante liberar los recursos JMS cuando dejan de utilizarse. Algunas prácticas útiles son:

- Si queremos mantener un recurso únicamente durante la vida de un método de negocio, es una buena idea cerrar el recurso en el bloque `finally` dentro del método.
- Si queremos mantener un recurso durante la vida de una instancia EJB, es conveniente utilizar un método anotado con `@PostConstruct` para crear el recurso y otro método anotado con `@PreDestroy` para cerrarlo. Si utilizásemos un EJB de sesión con estado, para mantener el recurso JMS en un estado cacheado, deberíamos cerrarlo y poner su valor a `null` mediante un método anotado con `@PrePassivate` , y volver a crearlo en un método anotado como `@PostActive`.

5.4.2.2. Transacciones

En vez de usar transacciones locales, utilizamos transacciones CMT para métodos de los ejbs que realizan envíos o recepciones de mensajes, permitiendo que el contenedor EJB gestione la demarcación de las transacciones. Ya que las transacciones CMT son las que se usan por defecto, no tenemos que utilizar ninguna anotación para especificarlas.

También podemos utilizar transacciones BMT y el interfaz `javax.transaction.UserTransaction` para demarcar las transacciones de un modo programativo, pero solo debemos hacerlo así si nuestros requisitos son muy complejos y dominamos muy bien los conceptos sobre transacciones. Normalmente, CMT produce el comportamiento más eficiente y correcto.

5.4.3. Ejemplo de un EJB que Envía Mensajes

A continuación mostramos un sencillo EJB que realiza la misma función que el productor de mensajes creado en la primera sesión.

El interfaz remoto es el siguiente:

```
package es.ua.jtech.jms;

// imports

@Remote
public interface ProductorSLSBRemote {
    void enviaMensajeJMS(String mensaje) throws JMSException;
}
```

Y en cuanto al bean, el código es muy similar al visto como aplicación cliente:

```
package es.ua.jtech.jms;

// imports

@Stateless
public class ProductorSLSBBean implements ProductorSLSBRemote {
```

```
    @Resource(name = "jms/ConnectionFactory")
    private ConnectionFactory connectionFactory;
    @Resource(name = "jms/Queue")
    private Queue queue;

    public void enviaMensajeJMS(String mensaje) throws JMSException
{
        Connection connection = null;
        Session session = null;
        try {
            connection = connectionFactory.createConnection();
            connection.start();
            session = connection.createSession(true, 0);

            TextMessage tm = session.createTextMessage(mensaje);

            MessageProducer messageProducer =
session.createProducer(queue);
            messageProducer.send(tm);
        } finally {
            if (session != null) {
                session.close();
            }
            if (connection != null) {
                connection.close();
            }
        }
    }
}
```

Tal como hemos comentado antes, destacar como las instancias de los recursos administrados ya no son estáticas, así como el cierre de los recursos dentro del bloque *finally*.

5.4.4. Ejemplo de Aplicación Web que Envía Mensajes

A continuación tenemos un *Servlet* que envía un mensaje, igual que el *EJB* anterior:

```
package es.ua.jtech.jms;

// imports

public class ProductorJMSServlet extends HttpServlet {

    @Resource(name = "jms/ConnectionFactory")
    private ConnectionFactory connectionFactory;
    @Resource(name = "jms/Queue")
    private Queue queue;

    @Override
    protected void doGet(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();

        String mensaje = "Este es un mensaje enviado desde un
```

```
Servlet";

        try {
            this.enviaMensajeJMS(mensaje);
            out.println("Enviado mensaje: " + mensaje);
        } catch (JMSException jmse) {
            jmse.printStackTrace(System.err);
        } finally {
            out.close();
        }
    }

    private void enviaMensajeJMS(String mensaje) throws
JMSException {
        Connection connection = null;
        Session session = null;
        try {
            connection = connectionFactory.createConnection();
            connection.start();
            session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);

            TextMessage tm = session.createTextMessage(mensaje);

            MessageProducer messageProducer =
session.createProducer(queue);
            messageProducer.send(tm);
        } finally {
            if (session != null) {
                session.close();
            }
            if (connection != null) {
                connection.close();
            }
        }
    }
}
```

Como podéis observar, el método de envío de mensajes es exactamente igual (sólo hemos cambiado el método de `public` a `private`).

Si lo que queremos es que el Servlet envíe el mensaje a través del EJB anterior, podríamos hacer esto:

```
package es.ua.jtech.jms;

// imports

public class ProductorEJBServlet extends HttpServlet {

    @EJB
    private ProductorSLSBRemote productorSLSBBean;

    @Override
    protected void doGet(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
```

```
        PrintWriter out = response.getWriter();

        String mensaje = "Este es un mensaje enviado desde un
Servlet que llama a un EJB";

        try {
            productorSLSBBean.enviaMensajeJMS(mensaje);
            out.println("Enviado mensaje: " + mensaje);
        } catch (JMSException jmse) {
            jmse.printStackTrace(System.err);
        } finally {
            out.close();
        }
    }
}
```

En la siguiente sesión repasaremos el ciclo de vida de los EJBs, y veremos como podríamos mejorar el EJB para que tanto la creación como el cierre de la conexiones se realizase únicamente en un sitio, y no una vez por cada método de negocio que tengamos.

6. Ejercicios de Subscripciones Duraderas, Transacciones Locales y Uso de JMS en JavaEE

6.1. Transacciones Locales (1p)

Las hipotecas con color naranja requieren de un estudio específico por parte de analistas financieros externos a nuestra entidad. Por lo tanto, al recibir el color naranja, le reenviaremos toda la información de la hipoteca a un sistemas externo.

Para ello, modificaremos el proyecto del semáforo para que en el caso de que el semáforo devuelva `Naranja`, antes de sacar el color por la consola, envíe un mensaje con `DatosHipoteca` a una nueva cola (`ConsultorExternoQueue`).

Así pues, tanto la recepción asíncrona como el envío síncrono del mensaje deberán formar parte de una transacción JMS.

6.2. Semáforo EAR

Vamos a crear una nueva aplicación EAR (`jms-banco-semaforo-ear`) que se base en nuestro semáforo hipotecario.

Consejo

A la hora de crear el EAR mediante Netbeans, es conveniente también crear los proyectos ejb y web

6.2.1. jms-banco-ejb (1p)

Crearemos un nuevo proyecto `jms-banco-ejb`, de modo que construyamos un EJB de sesión sin estado y con únicamente el interfaz local (`ConsultorHipotecaSLSBean`) que haga la función del banco, es decir, que envíe mensajes a la cola. Para ello, crearemos el siguiente método:

```
String enviaHipotecaSemaforo(String banco, double cuantia, int
anyos, double interes, double nomina)
```

Este EJB enviará un mensaje a la misma cola (`SemaforoHipotecarioRequestQueue`) que lo enviaba el proyecto `jms-banco`, con la misma propiedad (`BancoCaja`).

De momento, el EJB siempre devolverá `null`. En la siguiente sesión cambiaremos su comportamiento.

6.2.2. jms-banco-web (1p)

Además, crearemos un aplicación web que contendrá dos Servlets:

1. El Servlet `SolicitarHipotecaEJBServlet`, el cual llamará al EJB.
2. Y el Servlet `SolicitarHipotecaJMSServlet`, el cual enviará directamente el mensaje a la cola desde el Servlet, sin hacer uso del EJB. Es decir, tendrá código JMS dentro del Servlet.

Para facilitar el trabajo, disponemos del siguiente formulario para comprobar el resultado:

```
<form action="SolicitarHipotecaEJBServlet">
   Banco: <input type="text" name="banco" value="Banesto" /> <br />
   Cuantia: <input type="text" name="cuantia" value="150000.0" />
<br />
   Años: <input type="text" name="anyos" value="30" /> <br />
   Interes: <input type="text" name="interes" value="4.5" /> <br />
   Nómina: <input type="text" name="nomina" value="40000.0" /> <br
/>
   <input type="submit" value="Preguntar Semaforo" />
</form>
```

7. Message Driven Beans (MDBs)

7.1. Introducción

Un *Message-Driven Bean* o MDB (EJB dirigido por mensajes) es un oyente de mensajes que puede consumir mensajes de una cola o de una *durable subscription*. Dichos mensajes pueden ser enviados por cualquier componente JavaEE (cliente, otro EJB o una componente Web como un servlet). Incluso desde una aplicación o sistema que no use tecnología JavaEE.

Conceptualmente se diseñaron para que el servidor de aplicaciones proporcionase facilidades de *multi-threading*, esto es que múltiples consumidores procesen mensajes concurrentemente sin necesidad de desarrollar código adicional. Así, los MDBs proporcionan dicha facilidad al manejar los mensajes entrantes mediante múltiples instancias de beans alojados en el *pool* del servidor de aplicaciones.

Al igual que en el caso de los clientes JMS *standalone* que se basan en el método `onMessage()`, un MDB también contiene este método que se invoca automáticamente a la llegada de un mensaje. Sin embargo, los MDBs difieren de estos clientes en que el contenedor EJB realiza automáticamente varias tareas de inicialización que implementamos a mano en el cliente, como:

- Crear un consumidor asíncrono (`MessageConsumer`/`QueueReceiver`/`TopicSubscriber`) para recibir el mensaje. En vez de crear el consumidor en el código fuente, con un MDB asociamos el destino y la factoría de conexiones durante el despliegue. Si se quiere especificar una subscripción duradera o un selector de mensajes también se puede hacer en tiempo de despliegue.
- Registrar el listener de mensajes. El MDB registra el listener automáticamente sin que haya que codificar una llamada a `setMessageListener`.
- Especificar el modo de acuse de recibo. El modo de acuse de recibo por defecto es `AUTO_ACKNOWLEDGE`, y se utiliza a no ser que se cambien mediante una propiedad de configuración.

Para ello el MDB usa la anotación `@MessageDriven` para especificar las propiedades del bean o de la factoría de conexión, tales como el tipo de destino, la subscripción duradera, el selector de mensajes, o el modo de acuse de recibo.

Por defecto, el contenedor iniciará una transacción justo ANTES de llamar al método `onMessage()` y hará un commit de esta transacción cuando dicho método haga el return, a menos que la transacción esté marcada como rollback en el contexto del MDB.

Con respecto a otros EJBs, la diferencia fundamental con cualquier otro EJB es que el MDB no tiene interface local o remota. Solo la clase bean. Se parece a un *Stateless*

Session Bean (SSB) porque sus instancias son short-lived y no retienen estado para un cliente específico. Pero sus variables pueden contener información de estado entre los diferentes mensajes de cliente: por ejemplo, un conexión a una base de datos, o una referencia a un EJB, etc...

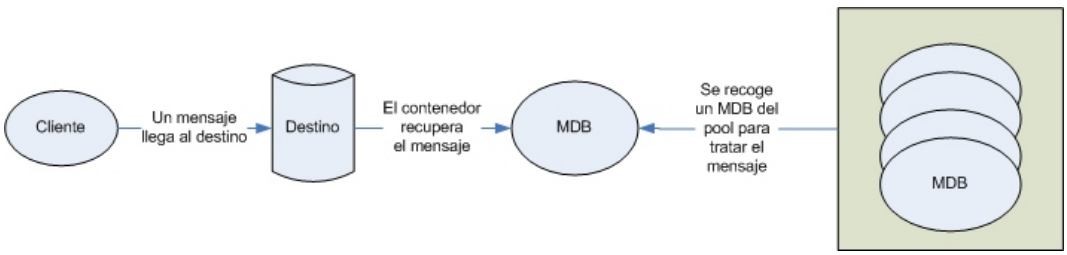
Como un SSB, un MDB puede tener varias instancias intercambiables ejecutándose al mismo tiempo. El contenedor puede hacer un pooling de instancias para permitir que los mensajes se procesen concurrentemente, lo cual puede afectar al orden en que se reciben los mensajes. Por lo tanto, como la concurrencia puede afectar el orden en que se entregan los mensajes, nuestras aplicaciones deberán gestionar los mensajes que llegan en cadena, por ejemplo, mediante un control conversacional a nivel de aplicación que vaya cacheando los mensajes hasta que se reciban todos.

7.2. Por Qué Utilizar MDBs

Frente a la mala fama, en términos de escalabilidad, que ha acompañado hasta ahora a los EJBs (antes de EJB3 claro!) los MDBs siempre han *mantenido el tipo*. Esto se debe a una serie de características que tienen los MDBs.

7.2.1. Multihilo

Las aplicaciones de negocio pueden necesitar consumidores de mensajes multihilo que puedan procesar los mensajes de modo concurrente. Los MDBs evitan esta complejidad ya que soporta el multihilo sin necesidad de código adicional. Los MDBs gestionan los mensajes entrantes mediante múltiples instancias de beans (dentro de un pool), y tan pronto como un nuevo mensaje llega al destino, una instancia MDB sale del pool para manejar el mensaje.



MDBs y Multihilo

7.2.2. Código de Mensajería Simplicado

Los MDBs evitan la necesidad de codificar los aspectos mecánicos asociados al procesamiento de mensajes (como buscar las factorías de conexiones o los destinos, crear las conexiones, abrir sesiones, crear consumidores y adjuntar *listeners*). Mediante EJB 3, el uso de situaciones por defecto para las circunstancias más comunes elimina gran parte de la configuración. En el peor caso, tendremos que ofrecer la información de

configuración via anotaciones o mediante el descriptor de despliegue.

7.2.3. Inicio de Consumo de Mensajes

Para comenzar a recoger mensajes de la cola de peticiones, alguien necesita invocar el método apropiado dentro del código. En un entorno de producción, no queda claro quien y donde recae esta responsabilidad. El inicio de consumo de mensajes mediante un proceso manual claramente no es deseable. En un entorno de servidor, casi cualquier manera de ejecutar el método al inicio del servidor es altamente dependiente del sistema; del mismo modo ocurre para detener la recepción de mensajes de manera manual.

Mediante los MDBs registrados, podremos iniciar o detener estos componentes de una manera sencilla cuando se arranque o detenga el servidor.

7.3. Reglas de Programación

Igual que los EJBs, los MDBs son POJOs que siguen un sencillo conjunto de reglas y que en ocasiones tienen anotaciones:

1. La clase MDB debe directamente (mediante la palabra clave `implements` en la declaración de la clase) o indirectamente (mediante anotaciones o descriptores) implementar un interfaz de *listener* de mensajes.
2. La clase MDB debe ser concreta, ni abstracta ni final.
3. La clase MDB debe ser un POJO y no una subclase de otro MDB.
4. La clase MDB debe declararse pública.
5. El constructor de la clase MDB no debe tener argumentos. Si no se tiene un constructor, el compilador implementará un constructor por defecto. El contenedor usa ese constructor para crear instancias de MDBs.
6. No se puede definir un método `finalize`. Si es necesario alguno código de limpieza, se debería definir un método designado como `PreDestroy`.
7. Los MDBs deben implementar los métodos de la interfaz `MessageListener` y esos métodos deben ser públicos, nunca estáticos o finales.
8. Esta prohibido lanzar `javax.rmi.RemoteException` o cualquier excepción de ejecución. Si se lanza un `RuntimeException`, la instancia MDB finalizará.

7.4. Ejemplo de un Consumidor con MDBs

A continuación vamos a realizar el mismo consumidor que hicimos en la primera sesión, pero únicamente de forma asíncrona via MDBs.

```
@MessageDriven(mappedName = "jms/Queue", activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType",
propertyValue = "javax.jms.Queue")
})
public class ConsumidorMDBBean implements MessageListener {

    public ConsumidorMDBBean() {
```

```
        System.out.println("Constructor del MDB");
    }

    public void onMessage(Message message) {
        TextMessage msg = null;

        try {
            if (message instanceof TextMessage) {
                msg = (TextMessage) message;
                System.out.println("Recibido MDB [" + msg.getText()
+ "]");
            } else {
                System.err.println("El mensaje no es de tipo
texto");
            }
        } catch (JMSException e) {
            System.err.println("JMSException en onMessage(): " +
e.toString());
        } catch (Throwable t) {
            System.err.println("Exception en onMessage():" +
t.getMessage());
        }
    }
}
```

7.5. Anotaciones de un MDB

7.5.1. Uso de la Anotación con @MessageDriven

Los MDBS son sin duda los EJBs más sencillos de desarrollar, y por tanto soportan muy pocas anotaciones. De hecho, la anotación `@MessageDriven` y su anidada `@ActivationConfigProperty` con las únicas anotaciones específicas de los MDBs.

La anotacion `@MessageDriven` utilizada en el ejemplo representa el caso típico que se utilizará la mayoría de las veces. La anotación se define así:

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface MessageDriven {
    String name() default "";
    Class messageListenerInterface default Object.class;
    ActivationConfigProperty[] activationConfig() default {};
    String mappedName();
    String description();
}
```

Destacar que todos los argumentos son opcionales, de modo, que en su uso más minimalista la anotación quedará así:

```
@MessageDriven
public class GestorPeticionesCompraMDB
```

dejando todo los detalles para el descriptor de despliegue.

El primer elemento, `name`, especifica el nombre del MDB (en nuestro caso `GestorPeticionesCompraMDB`). Si el nombre del elemento se omite, el código utiliza el nombre de la clase como nombre del MDB. El segundo parámetro, `messageListenerInterface`, especifica que listener de mensajes implementa el MDB. El parámetro `actionConfig` se utiliza para especificar propiedades de configuración específicas del listener. Finalmente, mediante `mappedName` indicaremos la ruta JNDI del destino.

7.5.2. Implementado el MessageListener

Un MDB implementa un interfaz de listener de mensaje por la misma razón que los consumidores JMS implementan el interfaz `javax.jms.MessageListener`. El contenedor utiliza el listener para registrar el MDB en el proveedor de mensajes y pasar los mensajes de entrada a los métodos implementados en el listener.

Utilizar el parámetro `messageListenerInterface` de la anotación `@MessageDriven` es solo una manera de especificar un listener de mensajes. Podríamos hacer lo mismo de este otro modo:

```
@MessageDriven(
    name="MiGestorPeticionesCompra",
    messageListenerInterface="javax.jms.MessageListener")
public class GestorPeticionesCompraMDB {
```

Sin embargo, es más cómodo omitir el parámetro y especificar el interfaz con la palabra clave `implements`:

```
public class GestorPeticionesCompraMDB implements MessageListener {
```

Otra opción es especificar el interfaz del listener mediante el descriptor de despliegue, y dejar los detalles fuera del código. La elección entre un modo u otro suele ser cuestión de gustos, aunque algunas herramientas se sienten más cómodas con el segundo enfoque, ya que facilita la generación de los métodos del bean a partir del interfaz.

7.5.3. Uso de ActivationConfigProperty

La propiedad `activationConfig` de la anotacion `@MessageDriven` nos permite especificar la configuración específica de nuestro sistema de mensajería mediante un array de instancia de `ActivationConfigProperty`. La definición de `ActivationConfigProperty` es:

```
public @interface ActivationConfigProperty {
    String propertyName();
    String propertyValue();
}
```

Cada propiedad de activación es un par (nombre, valor) que el proveedor conoce, entiende y utiliza para iniciar el MDB. El mejor modo de ver como funcionan estas

propiedades es mediante ejemplo. En el siguiente ejemplo podemos ver como configuramos tres de las propiedades más comunes:

```
@MessageDriven(
    name="MiGestorPeticionesCompra",
    activationConfig = {
        @ActivationConfigProperty(
            propertyName="destinationType",
            propertyValue="javax.jms.Queue"),
        @ActivationConfigProperty(
            propertyName="connectionFactoryJndiName",
            propertyValue="jms/QueueConnectionFactory"),
        @ActivationConfigProperty(
            propertyName="destinationName",
            propertyValue="jms/PeticionesCompraQueue")
    }
)
```

La propiedad `destinationType` le indica al contenedor que este MDB está escuchando de una cola. A continuación, `connectionFactoryJndiName` le indica el nombre JNDI de la factoría de conexiones que debe utilizarse para crear las conexiones JMS para el MDB. Y el `destinationName` indica que estamos escuchando mensajes que llegan a un destino cuyo nombre JNDI es `jms/PeticionesCompraQueue`.

A continuación veremos otras propiedades JMS que ya hemos estudiado en las sesiones anteriores.

7.5.3.1. acknowledgeMode

Ya sabemos que los mensajes no se eliminan de la cola hasta que el consumidor no envía el acuse de recibo. Para sesiones no transaccionales, hay que elegir el modo más apropiado para el proyecto. El más común y conveniente es `AUTO_ACKNOWLEDGE`. El otro modo soportado por los MDBs es `DUPS_OK_ACKNOWLEDGE`.

Para cambiar el modo de acuse de recibo lo haríamos del siguiente modo:

```
@ActivationConfigProperty(
    propertyName="acknowledgeMode",
    propertyValue="DUPS_OK_ACKNOWLEDGE")
```

El tratamiento del acuse de recibo por parte de JMS ya lo vimos en la segunda sesión: sesion02-apuntes.html#Controlar+el+Acuse+de+Recibo+de+los+Mensajes

7.5.3.2. subscriptionDurability

Si queremos que un MDB sea un subscritor duradero, tendremos algo así:

```
@ActivationConfigProperty(
        propertyName="destinationType",
        propertyValue="javax.jms.Topic"),
@ActivationConfigProperty(
        propertyName="subscriptionDurability",
```

```
        propertyValue="Durable")
```

Para subscripciones no duraderas, podemos fijar el valor de la propiedad `subscriptionDurability` a `NonDurable`, aunque este es el valor por defecto.

7.5.3.3. messageSelector

La propiedad messageSelector es el homonimo MDB a aplicar un selector a un consumidor JMS. Si queremos hacer un selector con MDBs el cual obtenga aquellos mensajes del año 2008 haríamos:

```
@ActivationConfigProperty(
        propertyName="messageSelector",
        propertyValue="Anyo = 2008")
```

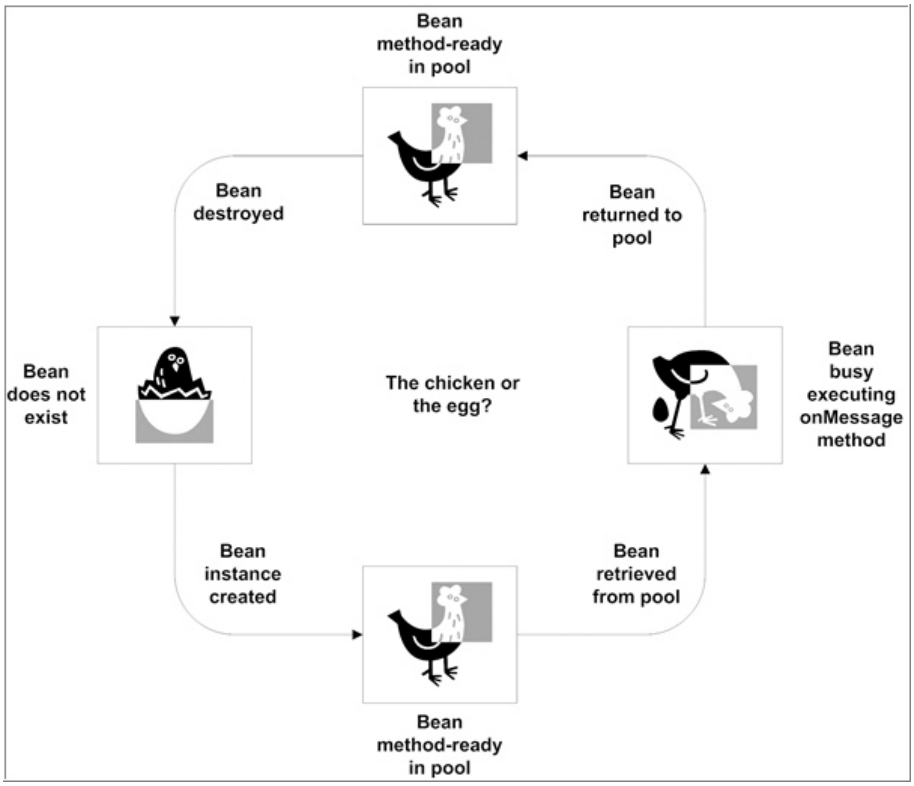
Los selectores de mensajes ya lo vimos en la segunda sesión: sesion02-apuntes.html#Selector+de+Mensajes

7.6. Uso de los Callbacks del Ciclo de Vida del Bean

El contenedor es responsable de:

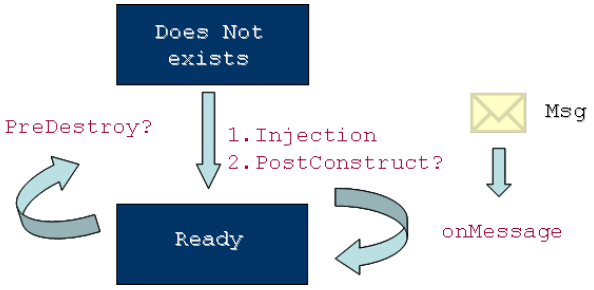
- Crear instancias MDBs y configurarlas.
- Inyectar recursos, incluyendo el contexto 'message-driven'.
- Colocar las instancias en un pool gestionado.
- Cuando llega un mensaje, sacar un bean inactivo del pool (en este punto, el contenedor puede que tenga que incrementar el tamaño del pool.
- Ejecutar el método de listener de mensajes (método `onMessage`)
- Al finalizar la ejecución del método `onMessage`, devolver al pool el bean.
- Conforme sea necesario, retirar (o destruir) beans del pool.

Un MDB es un EJB, y como tal, tiene un ciclo de vida, el cual tiene 3 estados: no existe, disponible y ocupado.



Ciclo de vida de un MDB

Los dos *callbacks* de ciclo de vida de un MDBs son (1) `PostContruct`, el cual se llama inmediatamente un vez el MDB se ha creado, iniciado y se le han inyectado todo los recursos, y (2) `PreDestroy`, que se llama antes de quitar y eliminar las instancias bean del pool. Estos *callbacks* se suelen utilizan para reservar y liberar recursos inyectados que se usan en el método `onMessage`, que es lo que hemos hecho en nuestro ejemplo.



Callbacks de un MDB

Método Callback

Para poder marcar un método como *callback* éste debe ser público, no puede ser ni final ni estático, debe devolver `void` y puede recibir 0 o 1 argumento.

Para demostrar el uso de los *callbacks*, vamos a modificar el MDB de modo que cuando le llegue un mensaje, realice un acceso a la base de datos.

```
@MessageDriven(mappedName = "jms/Queue", activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType",
propertyValue = "javax.jms.Queue")
})
public class ConsumidorMDBJDBCBean implements MessageListener {

    private java.sql.Connection connection;
    private DataSource dataSource;
    @Resource
    private MessageDrivenContext context;

    public ConsumidorMDBJDBCBean() {
        System.out.println("Constructor del MDB");
    }

    @Resource(name = "jdbc/biblioteca")
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @PostConstruct
    public void initialize() {
        try {
            connection = dataSource.getConnection();
        } catch (SQLException sqle) {
            sqle.printStackTrace();
        }
    }

    @PreDestroy
    public void cleanup() {
        try {
            connection.close();
            connection = null;
        } catch (SQLException sqle) {
            sqle.printStackTrace();
        }
    }

    public void onMessage(Message message) {
        TextMessage msg = null;

        try {
            if (message instanceof TextMessage) {
                msg = (TextMessage) message;
                System.out.println("Recibido MDB [" + msg.getText()
+ "]");
            } else {
                System.err.println("El mensaje no es de tipo
texto");
            }
            // Accedemos a la base de datos;
            this.preguntaBBDD("Total de Libros");
        } catch (JMSException jmse) {
            jmse.printStackTrace();
```

```
            context.setRollbackOnly();
        } catch (SQLException sqle) {
            sqle.printStackTrace();
            context.setRollbackOnly();
        } catch (Throwable t) {
            System.err.println("Exception en onMessage():" +
t.getMessage());
            context.setRollbackOnly();
        }
    }

    private int preguntaBBDD(String mensaje) throws SQLException {
        int result = -1;

        Statement stmt = connection.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT COUNT(*) FROM
LIBRO");
        if (rs.next()) {
            result = rs.getInt(0);
            System.out.println(mensaje + " " + result);
        }

        if (stmt != null) {
            stmt.close();
            stmt = null;
        }
        if (rs != null) {
            rs.close();
            rs = null;
        }

        return result;
    }
}
```

Ejemplo con JDBC

Aunque en el ejemplo hayamos utilizado JDBC, es recomendable considerar JPA para acceder a la base de datos

En el ejemplo vamos a utilizar el datasource del proyecto web (`jdbc/biblioteca`), el cual se supone que ya tenéis creado, así como la cola `jms/Queue` creada en la primera sesión.

Para evitar crear y destruir sesiones por cada llegada de un mensaje, utilizamos los métodos callbacks y la inyección de recursos. Primero inyectamos el *datasource* mediante la anotación `@Resource` en el método `setDataSource`. Esta anotación le indica al contenedor EJB que debería buscar el datasource `jdbc/biblioteca` vía *JNDI* y pasársela al método *set*. Tras inyectar los recursos, el contenedor comprueba si hay algún método anotado con `@PostConstruct` que deba invocarse antes de colocar el MDB en el pool. En nuestro caso, el método `initialize`, el cual se encarga de crear un conexión a partir de la fuente de datos.

En algún punto, el contenedor decir que nuestro bean debe salir del pool y ser destruido

(puede que al para el servidor). El callback `@PreDestroy` ofrece esta posibilidad para limpiar los recursos de un bean. Nosotros lo hemos utilizado en nuestro método `cleanup`, que se encarga de cerrar la conexión con la base de datos.

7.7. Envío de Mensajes JMS desde MDBs

Además de los recursos de base de datos, los *callbacks* también se utilizan para gestionar los objetos administrados de JMS (los destinos y la factoría de conexiones). Y aunque parezca un poco irónico, la tarea que más se realiza dentro de un MDB es enviar mensajes JMS. Por ejemplo, cuando un MDB recibe una petición puede que algo funcione mal o que la petición sea incompleta, y por tanto, la mejor manera de notificar esto es vía JMS a una cola de error sobre la que estará escuchando el productor del mensaje.

Vamos a modificar el primer ejemplo para que si el mensaje que recibe no es de tipo texto escriba en una cola de error. Para ello, aprovecharemos las ventajas que ofrece el MDB para inyectar las recursos y el uso de los *callbacks* para la creación y destrucción de los recursos.

```
@MessageDriven(mappedName = "jms/Queue", activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType",
propertyValue = "javax.jms.Queue")
})
public class ConsumidorMDBJMSBean implements MessageListener {

    private javax.jms.Connection jmsConnection;
    @Resource(name = "jms/ErrorQueue")
    private javax.jms.Destination errorQueue;
    @Resource(name = "jms/QueueConnectionFactory")
    private javax.jms.ConnectionFactory connectionFactory;
    @Resource
    private MessageDrivenContext context;

    @PostConstruct
    public void initialize() {
        try {
            jmsConnection = connectionFactory.createConnection();
        } catch (JMSException jmse) {
            jmse.printStackTrace();
        }
    }

    @PreDestroy
    public void cleanup() {
        try {
            jmsConnection.close();
        } catch (JMSException jmse) {
            jmse.printStackTrace();
        }
    }

    public void onMessage(Message message) {
        TextMessage msg = null;
```

```
        try {
            if (message instanceof TextMessage) {
                msg = (TextMessage) message;
                System.out.println("Recibido JMS-MDB [" +
msg.getText() + "]");
            } else {
                System.err.println("El mensaje no es de tipo texto.
Enviando mensaje a cola de error");
                this.enviaMensajeError();
            }
        } catch (JMSException jmse) {
            jmse.printStackTrace();
            context.setRollbackOnly();
        } catch (Throwable t) {
            System.err.println("Exception en onMessage():" +
t.getMessage());
            context.setRollbackOnly();
        }
    }

    private void enviaMensajeError() throws JMSException {
        Session session = jmsConnection.createSession(true,
                Session.AUTO_ACKNOWLEDGE);
        MessageProducer producer =
session.createProducer(errorQueue);
        TextMessage message = session.createTextMessage("El mensaje
recibido debía ser de tipo texto");
        producer.send(message);
        session.close();
    }
}
```

Siguiendo el esquema del ejemplo anterior, hemos creado variables de instancia para almacenar la factoría de conexiones y la cola para los mensajes de error. También hemos utilizado los callbacks para crear y destruir la conexión con la cola.

7.8. Gestionando las Transacciones Distribuidas

En los ejemplos JMS de las primeras sesiones, indicábamos si la sesión JMS era o no transaccional. Por otro lado, si revisamos los ejemplos MDB no indicamos en ningún sitio nada sobre la transaccionalidad. De hecho, le cedemos la decisión al contenedor para que utilice el comportamiento por defecto para los MDBs.

En concreto, por defecto, el contenedor comenzará una transacción antes de iniciar el método `onMessage` y realizará el *commit* tras el *return* del método, a no ser que se marque como *rollback* mediante el contexto *message-driven*.

Una aplicación JavaEE que utiliza JMS puede utilizar transacciones para combinar el envío o recepción de mensajes con modificaciones a una base de datos y cualquier otra operación con un gestor de recursos. Las transacciones distribuidas nos permiten acceder a múltiples componentes de una aplicación dentro de una única transacción. Por ejemplo, un Servlet puede empezar una transacción, acceder a varias bases de datos, invocar a un EJB que envía un mensaje JMS y realizar un commit de la transacción. Sin embargo,

persiste la restricción vista en la sesión anterior respecto a que la aplicación no puede enviar un mensaje JMS y recibir una respuesta dentro de la misma transacción.

7.8.1. Transacciones CMT y BMT dentro de un MDB

Con los MDBs, podemos utilizar transacciones tanto CMT como BMT. Para asegurar que todos los mensajes son recibidos y manejados dentro del contexto de una transacción, utilizaremos transacciones CMT, anotando el método `onMessage` con el atributo transaccional a `Required` (es el atributo por defecto). Esto significa que si no hay ninguna transacción en progreso, se comenzará una nueva antes de la llamada al método y se realizará el commit al finalizar el mismo.

Al utilizar transacciones CMT podemos emplear los siguientes métodos del `MessageDrivenContext`:

- `setRollbackOnly`: utiliza este método para el manejo de los errores. Si salta una excepción, `setRollbackOnly` marca la transacción actual para que la única salida de la transacción sea un *rollback*.
- `getRollbackOnly`: utiliza este método para comprobar si la transacción actual está marcada para hacer *rollback*.

Si utilizamos BMT, la entrega de un mensaje en el método `onMessage` tiene lugar fuera del contexto de la transacción distribuida. La transacción comienza cuando se llama al método `UserTransaction.begin` dentro del método `onMessage`, y finaliza al llamar a `UserTransaction.commit` o `UserTransaction.rollback`. Cualquier llamada al método `Connection.createSession` debe tener lugar dentro de la transacción. También hay que destacar que si en un BMT llamamos a `UserTransaction.rollback`, no se realiza la re-entrega del mensaje, mientras que si en una transacción CMT llamamos a `setRollbackOnly`, si que provoca la re-entrega del mensaje.

7.9. Mejores Prácticas

Igual que todas las tecnologías, los MDB tienen algunos escollos que hay que evitar, y algunas mejores prácticas que debemos tener en mente.

7.9.1. Elige con Cuidado el Modelo de Mensajería

Antes de empezar a codificar código como un loco, considera la elección del modelo de mensaje de un modo cuidadoso. Lo más normal es que el modelo PTP resuelva tus problemas más del 90% de los casos. En ocasiones, aunque contadas, el enfoque Pub/Sub es mejor, sobretodo si envías el mismo mensaje a más de un receptor.

Por suerte, la mayoría del código es independiente del dominio, y deberías codificar así, mediante los interfaces de más alto nivel. De este modo, cambiar el dominio debería ser una cuestión de configuración.

7.9.2. Modulariza

Debido a que los MDBs son muy similares a los beans de sesión, es natural empezar a colocar lógica de negocio dentro de los métodos del listener de mensajes. La lógica de negocio debe esta modularizada y desacoplada de los aspectos específicos de la mensajería.

En el ejemplos hemos creado un método privado para realizar la lógica de negocio, aunque mejor habría sido colocarlo en una clase aparte, en concreto, dentro de nuestros objetos de negocio, y desde el `onMessage` del MDB invocar al método de negocio adecuado.

7.9.3. Bueno Uso de los Filtros de Mensajes

Existen pocas razones para utilizar un único destino de mensaje para múltiples propósitos. Los selectores de mensajes son muy útiles en estas circunstancias. Por ejemplo, si utilizamos la misma cola tanto para enviar peticiones de compra como notificar las cancelaciones de los pedidos, podemos hacer que el cliente utilice una propiedad de mensaje para identificar el tipo de petición. En el MDB utilizaremos un selector de mensaje para separar y manejar cada tipo de petición.

A la inversa, mejoraremos de forma dramática el rendimiento y mantendremos el código más sencillo mediante destinos separados (en contra de los *selectors*). Así, utilizaríamos una cola para realizar las peticiones y otra para las cancelaciones de pedidos, siendo el cliente el que decide a que cola envía el mensaje.

7.9.4. Elige el Tipo de Mensajes con Cuidado

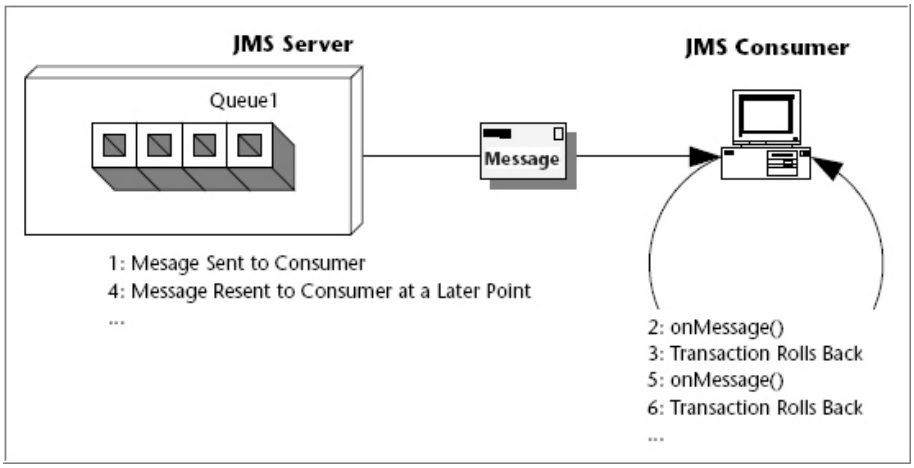
La elección del tipo de mensaje no siempre es tan obvia como pueda parecer. Por ejemplo, una idea atractiva es utilizar cadenas de texto XML; esto hace que el acoplamiento entre sistemas sea débil. Si hubiésemos utilizado un objeto, estaremos obligando que al otro lado conozcan dicho objeto.

El problema de XML es que hincha el tamaño de los mensajes de un modo significativo, degradando el rendimiento del MOM. En algunas ocasiones, la elección correcta será utilizar flujos binarios dentro del cuerpo, ya que éste tipo implica la mejor demanda de procesamiento MOM y consumo de memoria.

7.9.5. Cuidado con los Mensajes Venenosos

Un mensaje venenoso es un mensaje que recibe el MDB para el cual no está preparado. Imagina que uno de nuestros MDBs no puede manejar uno de los mensajes recibidos, por ejemplo, porque esperamos un mensaje tipo `ObjectMessage` y recibimos uno de texto. La recepción, sin control, de este tipo de mensajes causa que la transacción en la que está

inmerso el MDB haga un *rollback*. En nuestros ejemplos hemos protegido al MDB de los mensajes venenosos ya que la recepción se realiza dentro de un *try/catch* y se invoca al *rollback* si se detecta la situación. Sin embargo esto implica que como el MDB sigue escuchando, el mensaje venenoso se le enviará una y otra vez y entraremos en un bucle infinito.



Mensajes Venenosos

Afortunadamente muchos MOMs y contenedores EJB ofrecen mecanismos para tratar estos mensajes venenosos, incluyendo que: (a) no se intente el reenvío más de un determinado número de veces y (b) se habilite una cola de *dead messages*. Una vez un mensaje se ha intentado enviar el número de veces que hemos fijado en la configuración de la cola, éste pasa a almacenar en la cola de *dead messages*. Esto se puede controlar configurando el destino adecuadamente.

Las malas noticias es que estos mecanismos no están estandarizados, y dependen del proveedor de mensajería.

Mensajes Venenosos en Glassfish

Glassfish utiliza las propiedades `endpointExceptionRedeliveryAttempts` y `sendUndeliverableMsgsToDMQ`. Más información en docs.sun.com/app/docs/doc/820-6740/aeoon.

7.9.6. Configura el Tamaño del Pool MDB

La mayoría de contenedores EJB permiten especificar el número máximo de instancias de un determinado MDB que puede crear el contenedor. De hecho, esto controla el nivel de concurrencia. Si hay 5 mensajes concurrentes a procesar y el tamaño del pool es 3, el contenedor esperará hasta que los 3 primeros mensajes hayan sido procesados para asignar más instancias.

Esta es una espada de doble filo y requiere mucho cuidado en su manejo. Si el tamaño del

pool es muy pequeño, los mensajes se procesarán lentamente. Al mismo tiempo, es deseable fijar un límite razonable al tamaño del pool para que muchas instancias MDB concurrentes no ahoguen la máquina.

Para la configuración del pool mediante *Glassfish*, podéis acceder a la siguiente información: docs.sun.com/app/docs/doc/820-4336/beaiu

8. Ejercicios de Message Driven Beans

8.1. Semáforo MDB (1p)

Ahora queremos que el semáforo se implemente mediante un MDB. Para ello, dentro de la aplicación EAR creada en la sesión anterior, crearemos el proyecto `jms-semaforo-mdb`, que será un proyecto de tipo EJB.

Dentro del proyecto crearemos un MDB que nombraremos como `SemaforoMDB`, el cual se encargará únicamente de sacar el color por la consola.

8.2. Semaforo MDB Pregunta al Consultor (1p)

A continuación, modificaremos el MDB para seguir todas las restricciones impuestas en los ejercicios anteriores, como que solo reciba mensajes de bancos, manteniendo el envío de mensajes al consultor externo en el caso de que `SemaforoBR` devuelva `Naranja`, pero teniendo en cuenta la gestión de transacciones que realiza el MDB, etc...

Recuerda crear y destruir la conexión JMS con el consultor mediante los métodos del MDB anotados con `@PostConstruct` y `@PreDestroy`.

8.3. Semáforo MDB Responde al Banco (1p)

Modificar el proyecto anterior para que, además de sacar el color por consola, responda con el color en una nueva cola `SemaforoHipotecarioReplyQueue`.

A su vez, se debe modificar el EJB creado en la sesión anterior, de modo que una vez enviado el mensaje, se quede a la escucha en esta nueva cola durante 1 segundo (consumición síncrona). El EJB del banco devolverá el color recibido, o `null` si no ha recibido nada durante ese segundo.

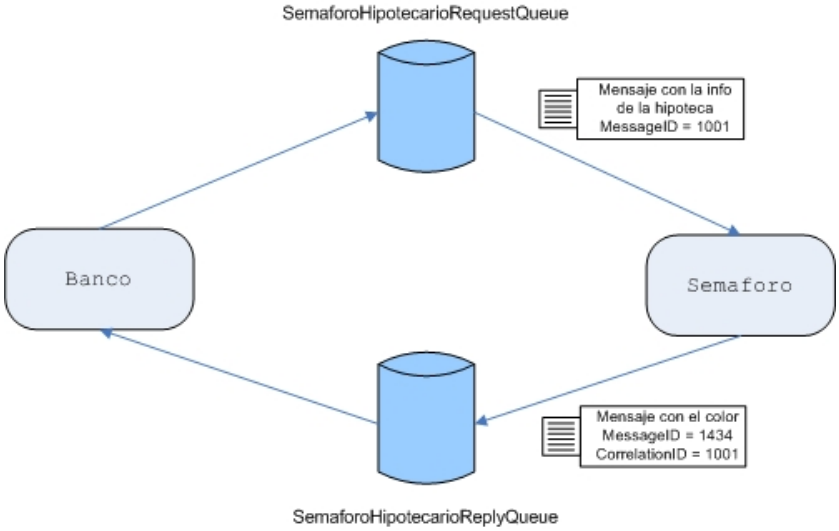
Para que el EJB no escuche mensajes de un banco diferente, utilizaremos las propiedades JMS `MessageID` y `CorrelationID`. Para ello, cuando el MDB responda al mensaje debe poner como `CorrelationID` del mensaje de respuesta, el `MessageID` del mensaje de origen. Es decir, cuando vayamos a enviar el color y creemos un mensaje de texto, haremos:

```
TextMessage respuesta = session.createTextMessage(color);
respuesta.setJMSCorrelationID(mensajeRecibido.getJMSMessageID());
```

A su vez, en el banco, tras enviar el mensaje, cuando no quedemos a la espera durante un segundo, utilzaremos un selector de mensaje de modo que filtremos mediante el `CorrelationID` cuyo valor será el del MessageID del mensaje que desencadeno el MDB.

```
String filtro = "JMSCorrelationID = '" + message.getJMSMessageID()
```

```
+ "'";
MessageConsumer consumer = session.createReceiver(queueRespuesta,
filtro);
// A continuación, consumimos esperando un segundo
```



Mecanimos Request/Reply con el Semaforo

Cuidado

Si tenemos corriendo 2 MDBs en el mismo servidor de aplicaciones, y los dos escuchando sobre la misma cola ¿qué pasa con los mensajes?