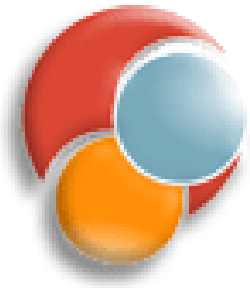


Servicios de Mensajes con JMS

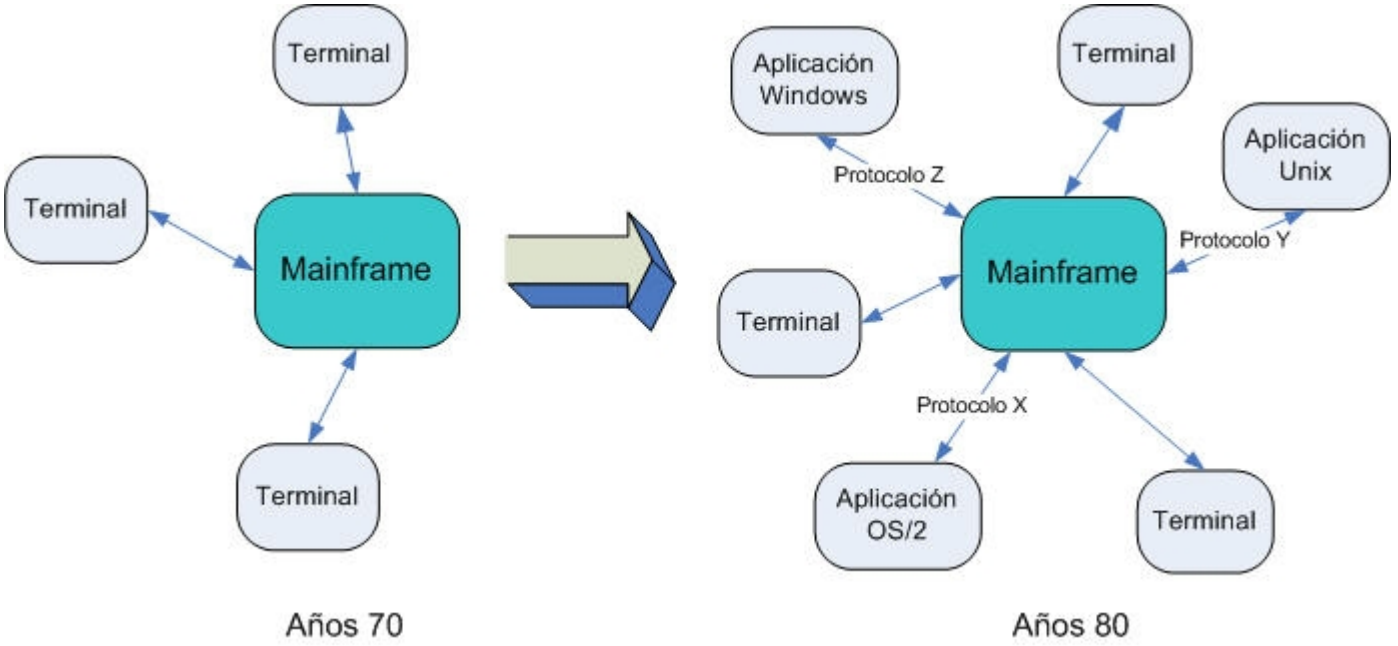
Sesión 1: Introducción a JMS

Puntos a Tratar

- Un Poco de Historia
- Dominios de Mensajería
- JMS
  - Arquitectura
  - Modelo de Programación
- Recursos JMS en *Glassfish*
- Una Aplicación JMS
  - PTP
  - Pub/Sub

Un Poco de Historia

- 60: mainframes
- 70: terminales + mainframes
- 80: PCs + múltiples protocolos



Mensajería Empresarial

- Objetivo: transferir información entre sistemas heterogéneos mediante el envío de mensajes.
- Diferentes soluciones de mensajería
  - **RPCs** que hacen funciones de *middleware* mediante una cola de mensajes
    Ej: COM y *CORBA*
  - **Notificación de eventos**, comunicación entre procesos y colas de mensajes, incluidos en los SSOO
    Ej: buffers FIFO, colas de mensajes, *pipes*, señales, *sockets*, ...
  - Soluciones para una categoría de ***middleware*** que ofrezca una mensajería fiable y asíncrona
    Ej: WebShpereMQ, *SonicMQ*, *TIBCO*, *Apache ActiveMQ*, etc...
- Se necesitaba un *sw* para comunicar y transferir datos entre diferentes aplicaciones, SSOO, protocolos e incluso diferentes lenguajes de programación.

¿Por Qué Mensajería?

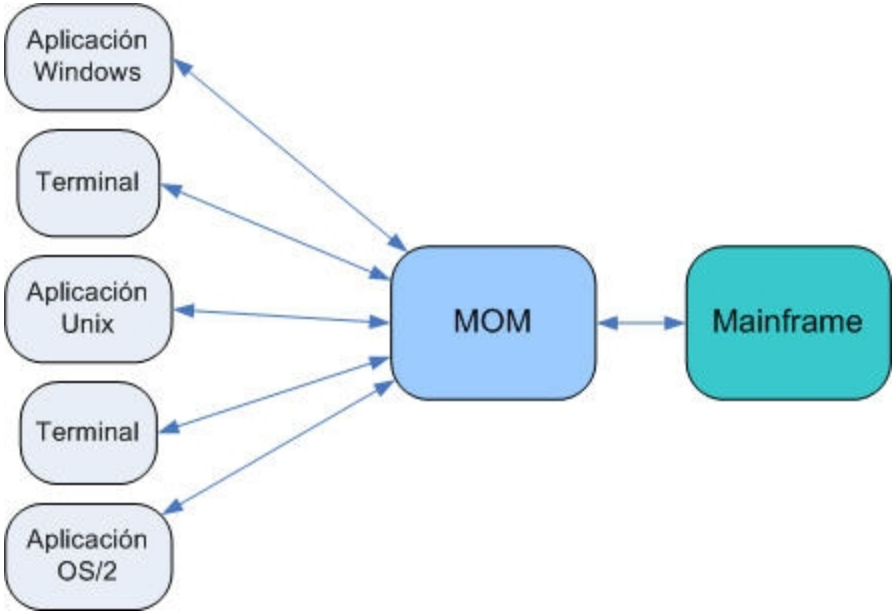
- Método de comunicación entre componentes software o aplicaciones.
- Comunicación distribuída **débilmente acoplada**
  - un componente envía un mensaje a un *destino* y el receptor lo recoge de dicho destino.
- Ni emisor ni receptor tienen que estar disponibles al mismo tiempo para comunicarse (**independencia y asíncronia**)
  - 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 mismo.

MOM

- El **M**idleware **O**rientado a **M**ensajes (MOM) es un categoría de software para la intercomunicación de sistemas.
- Mecanismo seguro, escalable, confiable y con bajo acoplamiento.
- Permite la comunicación entre aplicaciones mediante un conjunto de APIs ofrecidas por cada proveedor y lenguaje
  - tendremos un API propietaria y diferente por cada MOM existente.
- Actúa como un mediador entre los emisores y los receptores de mensajes.
  - Esta mediación ofrece un nuevo nivel de desacoplamiento.

MOM Como Mediador

- Se utiliza para mediar en la conectividad y la mensajería, no solo entre las aplicaciones y el *mainframe*, sino de una aplicación a otra.



Ventajas de Usar MOM

- **Persistencia de mensajes**: para conexiones lentas/poco fiables o donde el fallo de un receptor no afecta al emisor.
- **Enrutamiento complejo**: un mensaje a muchos receptores, enrutamientos basados en propiedades, …
- **Transformación de mensajes**: permite la comunicación entre dos aplicaciones que trabajan con diferentes formatos de mensajes.
- Soporte para **diversos protocolos** de conectividad
  - HTTP/S, SSL, TCP/IP, UDP, etc...
- Soporte para **múltiples lenguajes de programación**.

- Este gran abanico de protocolos, lenguajes y APIs provoca la aparición de JMS como estándar Java.

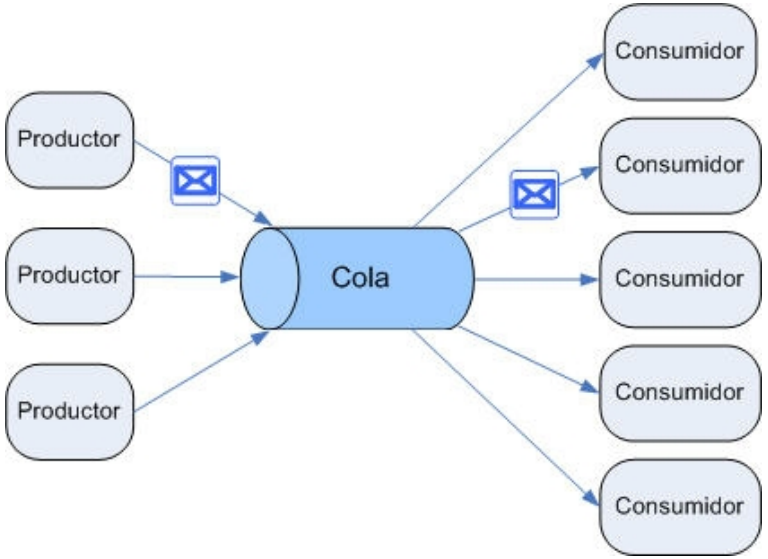
Dominios de Mensajería

- Existen 2 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.

- **No** todos los proveedores implementan ambos.

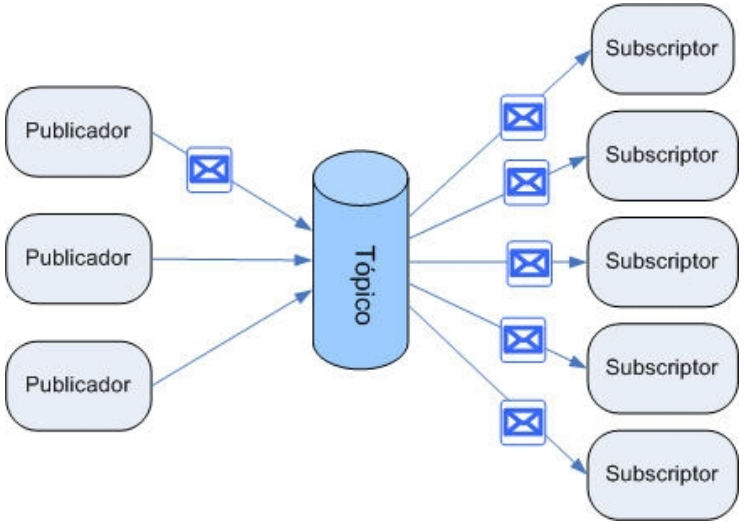
Punto a Punto

- **Un mensaje lo consume un único consumidor (1:1)**
  - Pueden haber varios emisores.
- El destino del mensaje es una **cola** definida y con un nombre
  - modelo FIFO: el 1er mensaje encolado será el 1º en salir.



- Cuando el receptor extrae el mensaje, envía un acuse de recibo a la cola para confirmar su correcta recepción.
- Mecanismo **pull à** las colas retienen todos los mensajes enviados hasta que son consumidos o hasta que expiren.
- Una aplicación PTP se construye bajo el concepto de **colas** de mensajes, **productores** y **consumidores**.

Pub/Sub



- **Un mensaje puede consumirse por múltiples consumidores (1:N).**
- El destino es un **tópico**.
  - Los mensajes en los tópicos no se encolan. Un nuevo mensaje en el tópico sobrescribirá cualquier mensaje existente.
- *Modelo de difusión*: los **publicadores** publican en un tópico, y los **subscriptores** se subscriben al tópico.
- El MOM se encarga de difundir los mensajes que llegan al tópico mediante un **mecanismo Push** -> los mensajes se envían automáticamente a los subscriptores.

Características Pub/Sub

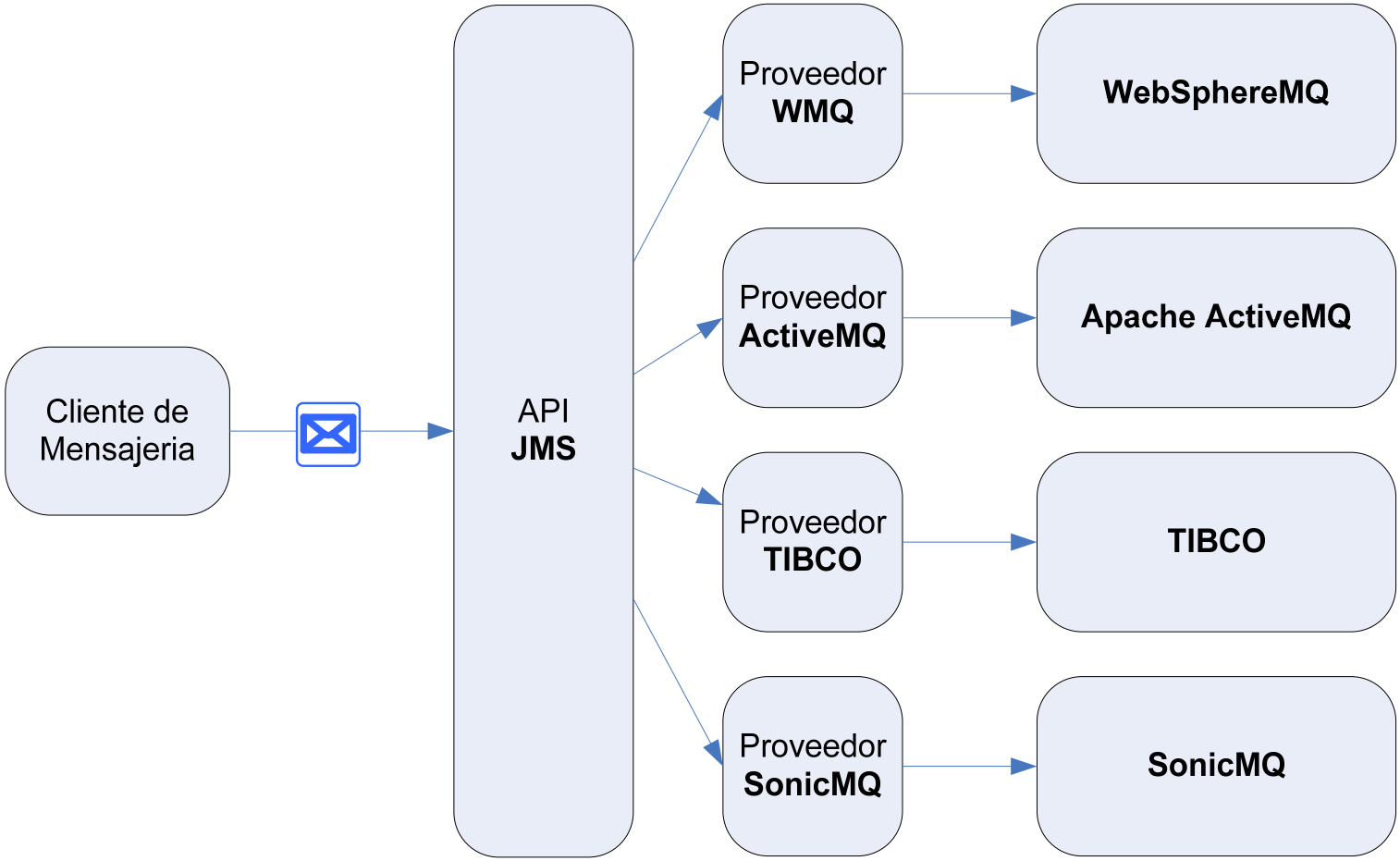
- Cada mensaje puede tener múltiples consumidores
- Existe un dependencia temporal entre los publicadores y los subscritores perecederos (*non-durable*)
  - Un cliente que se subscribe a un tópico puede consumir los mensajes publicados después de la subscripción, y el subscriptor debe continuar conectado para consumir los posteriores mensajes.
- El API JMS disminuye esta dependencia temporal permitiendo crear subscripciones duraderas (*durable*).
  - Permiten recibir mensajes que fueron enviados cuando los subscriptores no estaban conectados.
- Útil en situaciones donde un grupo de aplicaciones quiere notificar a otras de un evento particular.
  - Ejemplo: una aplicación de CRM, al crear un cliente, puede necesitar comunicar a otras aplicaciones la creación de éste.

JMS

- Especificación que ofrece un API estándar (mediante un conjunto de interfaces) para poder enviar y recibir mensajes sin atarnos a ningún proveedor.
- Minimiza el conocimiento de mensajería empresarial que debe tener un programador Java para desarrollar complejas aplicaciones de mensajería.
- Mantiene la portabilidad entre las diferentes implementaciones de proveedores JMS.

- **¡ JMS no es un MOM !.**
- Abstrae la interacción entre los clientes de mensajería y los MOMs
  - Igual que JDBC abstrae la comunicación con las BBDD relacionales.

JMS Como API Estándar



Historia de JMS

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

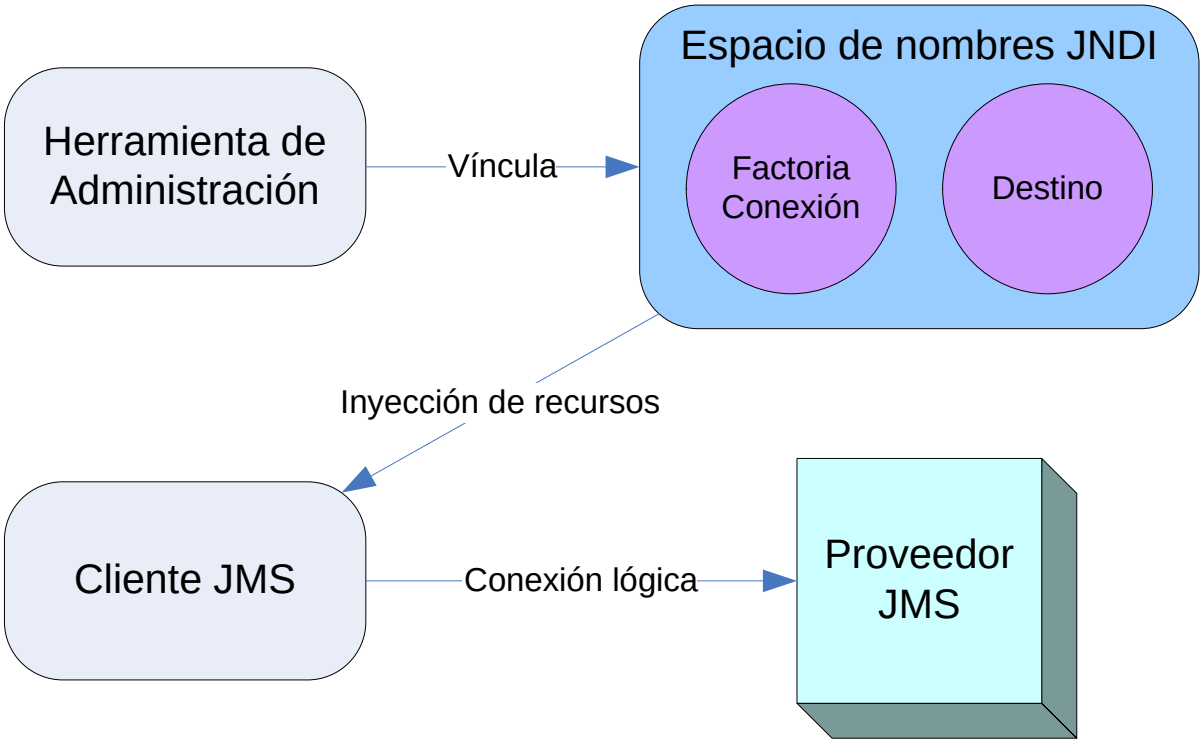
JMS y JavaEE

- JMS forma parte de JavaEE desde la v1.3 de JavaEE
- El API JMS dentro de la plataforma JavaEE aporta:
  - Aplicaciones cliente, componentes EJB y componentes Web que pueden enviar o recibir de forma síncrona/asíncrona mensajes JMS.
  - *Message-driven Beans* – MDBs: tipo de EJB que puede consumir mensajes de forma asíncrona.
- JMS mejora 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 también mejora JMS
  - con el soporte de **transacciones distribuidas**, y el procesamiento de mensajes de manera **concurrente**.

Conceptos JMS

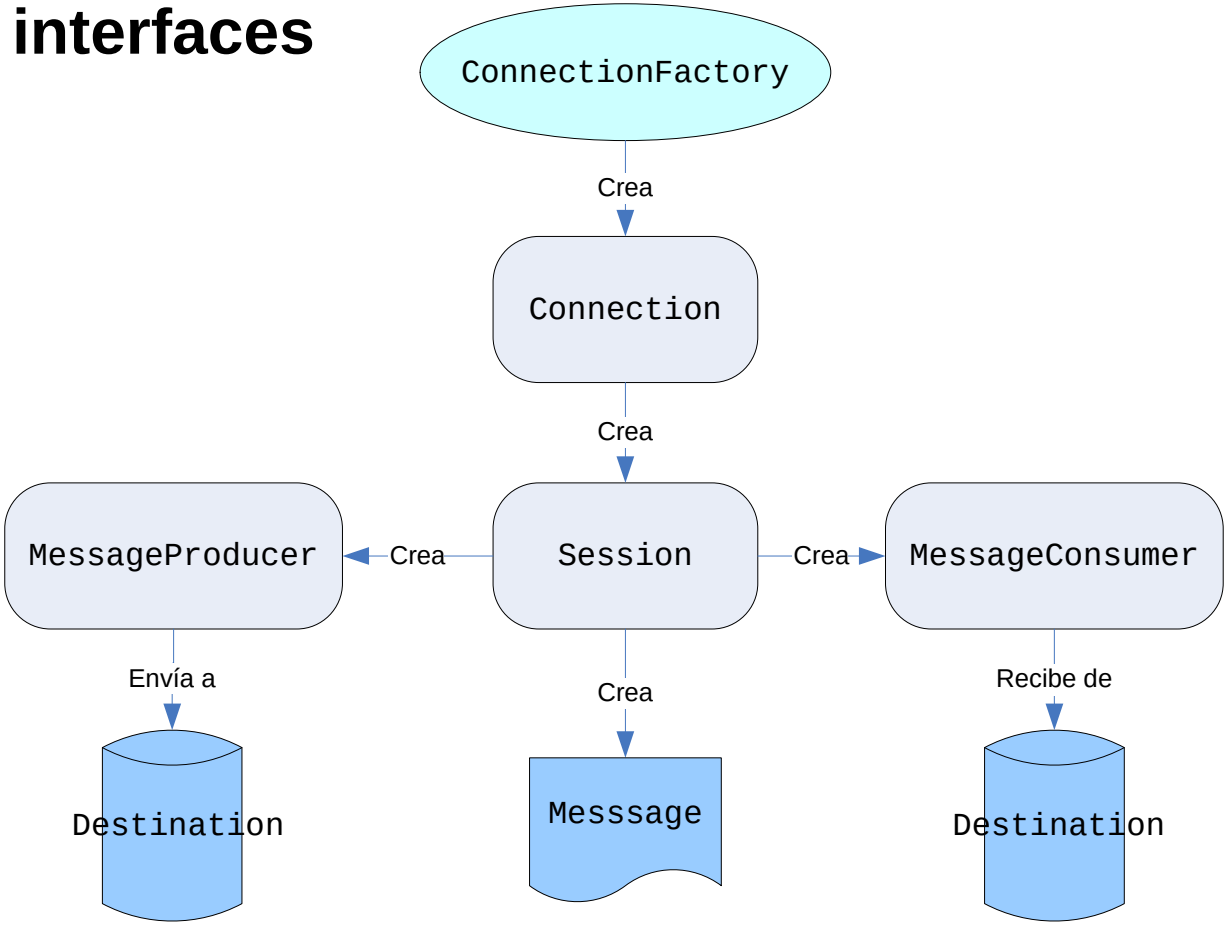
- ***Cliente JMS*:** Una aplicación 100% Java que envía y recibe mensajes.
  - Cualquier componente JavaEE puede actuar como un cliente JMS.
- ***Cliente No-JMS*:** aplicación no Java que envía y recibe mensajes.
- ***Productor JMS***: un cliente que crea y envía mensajes JMS.
- ***Consumidor JMS***: un 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***: 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***: permite la creación de conexiones con el proveedor JMS.
  - ***Destino***: objeto (cola/tópico) al cual se direccionan y envían los mensajes, y desde donde se reciben los mensajes.

Interacción de Elementos JMS

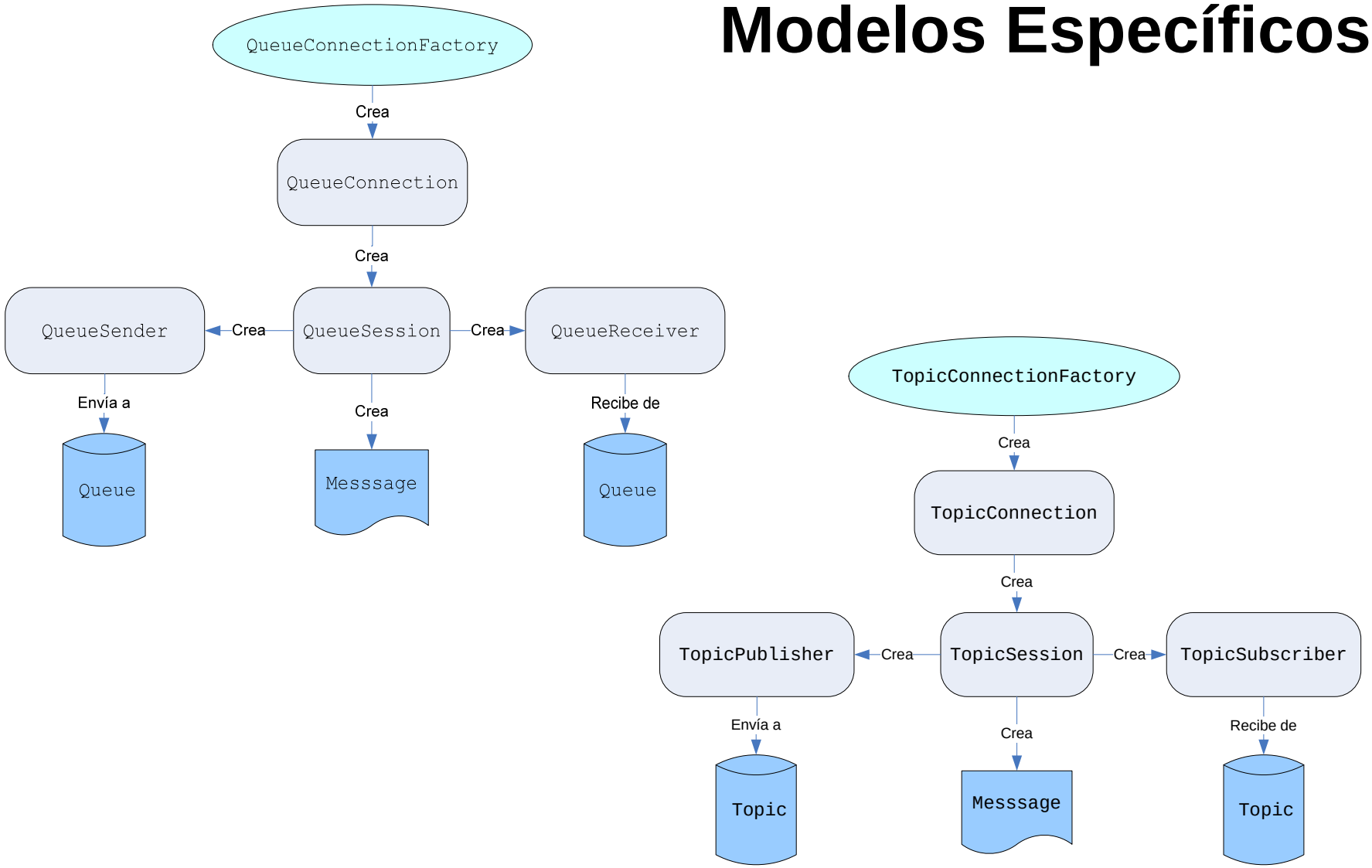


Modelo de Programación JMS

- Basado en **interfaces**



Modelos Específicos



Objetos Administrados

- Los dos extremos de las aplicación JMS, la factoría de conexiones y los destinos, son mantenidos vía administración (*vs* forma programativa)
  - 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
  - Un cliente pueda cambiar de implementación JMS sin necesidad de ninguna modificación.
- La administración de estos objetos se realiza dentro de un espacio de nombre JNDI
  - a través de la consola de administración de *Glassfish*
- Los clientes acceden a ellos mediante la inyección de recursos vía anotaciones.

Factorías de Conexión

- Es el objeto que utiliza el cliente para crear una conexión con el proveedor
- Encapsula 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`.

Ejemplos de Factorías de Conexión

- Factoría común:

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

- Factorías especificas:

```
@Resource(mappedName="jms/QueueConnectionFactory")
private static QueueConnectionFactory queueConnectionFactory;
@Resource(mappedName="jms/TopicConnectionFactory")
private static TopicConnectionFactory topicConnectionFactory;
```

Destinos

- 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.
- 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 `javax.jms.Destination`.

Ejemplos de Destinos e Intercambio

- Cola y Tópico

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

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

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

Conexiones

- Una conexión encapsula una conexión virtual con el proveedor JMS, y puede representar un socket TCP/IP entre el cliente y un demonio del proveedor.
- Al crear una conexión, se crean objetos tanto en la parte del cliente como en la del servidor
  - gestionan el trasiego de mensajes entre el cliente y el MOM.
- 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`.

Ejemplos de Conexiones

- Creando conexiones:

```
Connection connection = connectionFactory.createConnection();
QueueConnection queueConnection = queueConnectionFactory.createQueueConnection();
TopicConnection topicConnection = topicConnectionFactory.createTopicConnection();
```

- Al finalizar la aplicación, tenemos que cerrar toda conexión.
  - 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 detener el envío de mensajes de forma temporal sin cerrar la conexión, podemos utilizar el método `stop`.

Sesiones

- Una sesión es un contexto monohilo para producir y consumir mensajes. Mediante las sesiones crearemos:
  - Productores de mensajes.
  - Consumidores de mensajes.
  - Mensajes.
  - Navegadores de colas (*Queue Browser*).
  - Colas y tópicos temporales.
- Existen dos tipos de sesiones.
  - Transaccionales: sujeta al protocolo *commit/rollback*.

    En estas sesiones no es necesario enviar acuse de recibo.
  - No-transaccionales: hay que elegir un tipo de acuse de recibo.

Ejemplos de Sesiones

- 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);
```

- Para crear una sesión transaccional:

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

- También existen diferentes interfaces para cada dominio (colas con `QueueSession`, y para los tópicos `TopicSession`):

```
int acuse =  Session.AUTO_ACKNOWLEDGE;
QueueSession queueSession = queueConnection.createQueueSession(esTransaccional, acuse);
TopicSession topicSession = topicConnection.createTopicSession(esTransaccional, acuse);
```

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 productores:

```
MessageProducer producerD = session.createProducer(dest);
MessageProducer producerQ = session.createProducer(queue);
MessageProducer producerT = session.createProducer(topic);
```

Productores Específicos

- Al utilizar las sesiones específicas de cada dominio, los métodos también tienen sus nombres específicos, devolviendo productores específicos:

```
QueueSender sender = queueSession.createSender(queue);
TopicPublisher publisher = topicSession.createPublisher(topic);
```

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

```
producer.send(message);
```

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

```
MessageProducer anonProd = session.createProducer(null);
anonProd.send(dest, message);
```

Consumidores de Mensajes

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

```
MessageConsumer consumer = session.createConsumer(dest);
MessageConsumer consumer = session.createConsumer(queue);
MessageConsumer consumer = session.createConsumer(topic);
```

- En el caso de utilizar las sesiones específicas de cada dominio, los métodos también tienen sus nombres específicos, devolviendo consumidores específicos:

```
QueueReceiver receiver = queueSession.createReceiver(queue);
TopicSubscriber subscriber = topicSession.createSubscriber(topic);
```

Consumidores de Mensajes

- 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 siempre **llamar al método `start`**

  - es uno de los errores más comunes dentro de la programación JMS.
- Para consumir de forma síncrona utilizaremos el método `receive`.
  - Esta operación se puede realizar en cualquier momento si previamente se ha iniciado la conexión (mediante el método `start`):

```
connection.start();
Message m = consumer.receive();
connection.start();
Message m = consumer.receive(1000); // timeout tras un seg
```

Listener de Mensajes

- Es un objeto que actúa como un manejador de eventos asíncronos para mensajes.
- 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`.

```
MiListener myListener = new MiListener();
consumer.setMessageListener(myListener);
```

Funcionamiento del *Listener*

- 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 llama al método `onMessage` del listener.
- El método `onMessage` recibe un objeto `Message` con los datos.
- Nuestro método `onMessage` debería capturar todas las excepciones.
  - No debe lanzar excepciones *checked*
  - Relanzar excepciones *unchecked* (RuntimeException) se considera un error de programación.

- Un *listener* no es específico para un tipo de destino en particular.
  - El mismo *listener* puede obtener mensajes de colas y tópicos dependiendo del tipo de destino para el cual se creó el consumidor de mensajes.
  - Normalmente espera un tipo y formato específico de mensaje.
- En un instante cualquiera, uno y solo uno de los *listener* de mensajes de la sesión está en ejecución

Mensajes

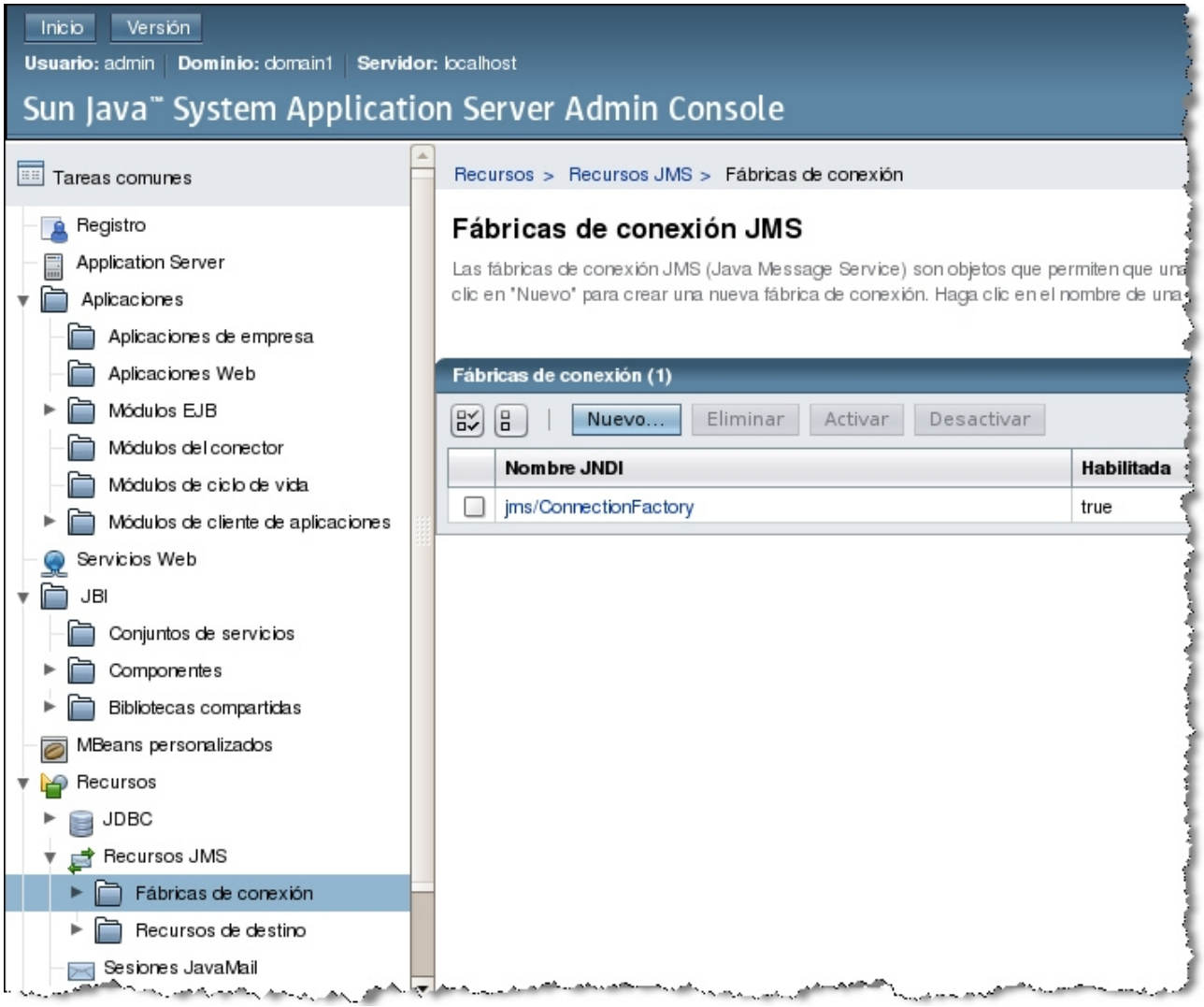
- Los mensajes también se crean a partir de objetos de sesión.

```
TextMessage message = session.createTextMessage();
```

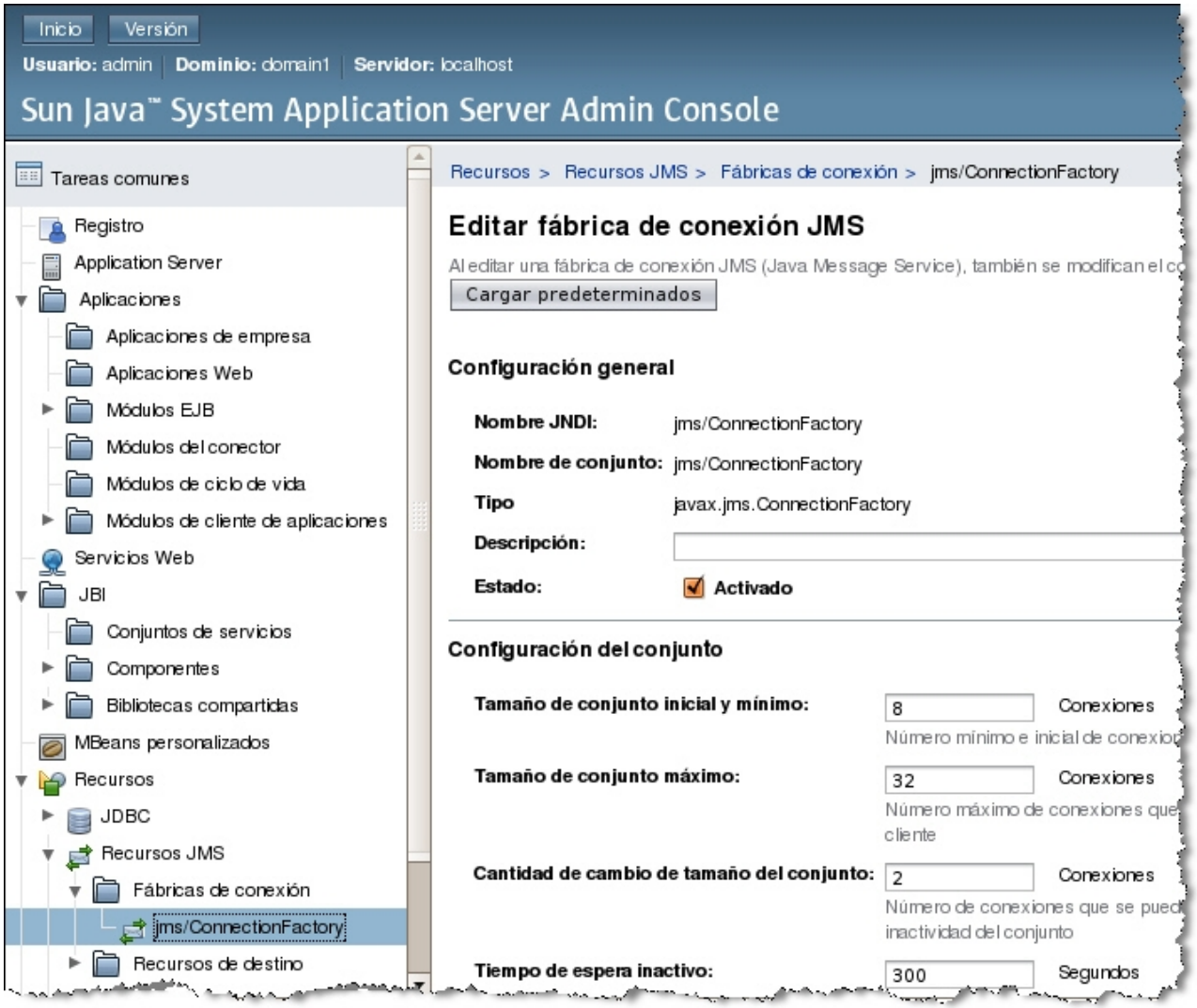
- Los mensajes encapsulan información a intercambiar entre aplicaciones.
- Un mensaje contiene tres componentes: los campos de la *cabecera*, las *propiedades* específicas de la aplicación y el *cuerpo* del mensaje.

Recursos JMS en *Glassfish*

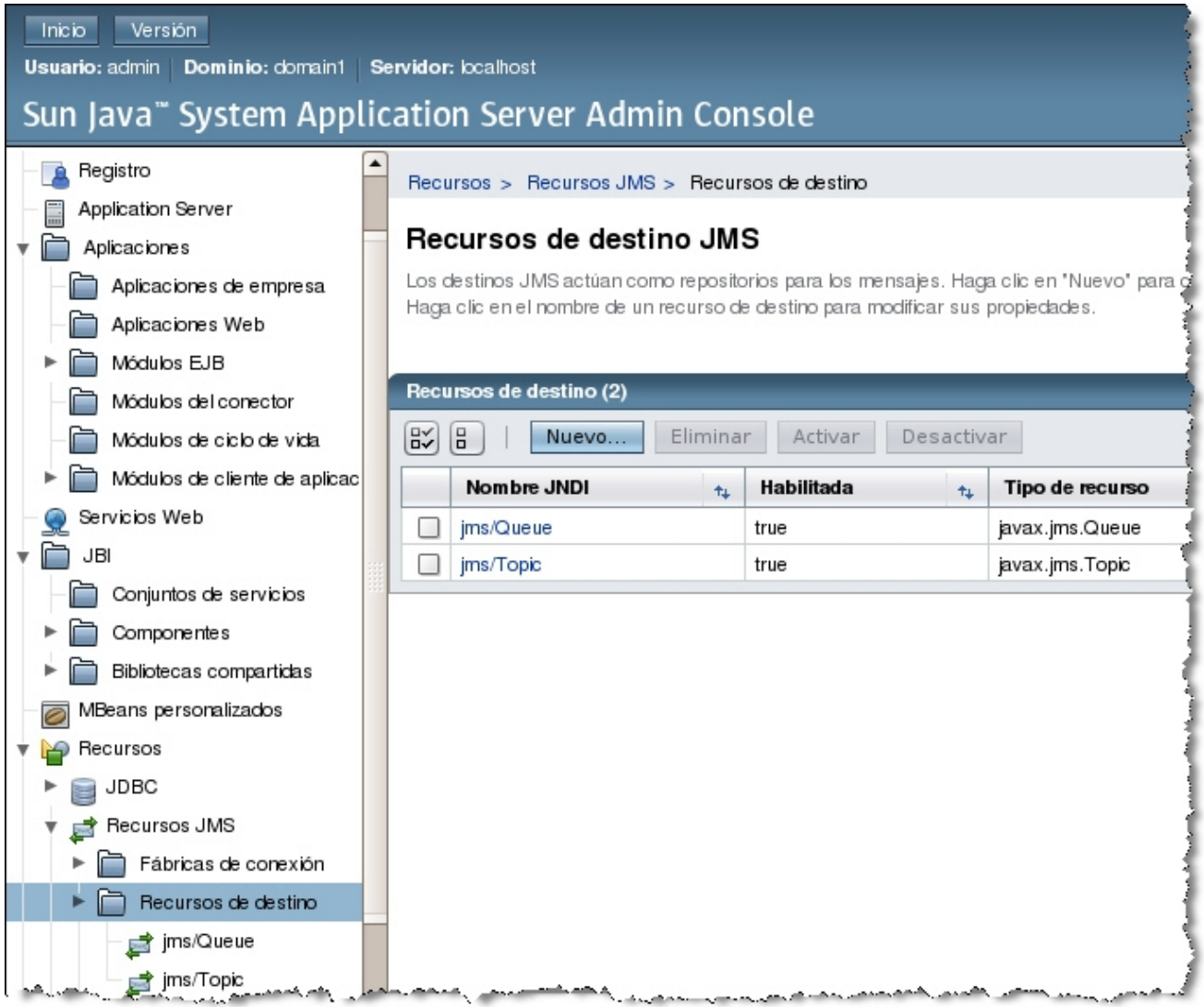
Factorías de Conexión en *Glassfish*



Factorías de Conexión en *Glassfish* (II)



Destinos en *Glassfish*



Pasos en Una Aplicación JMS

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
    1. Crear un productor.
        1. Crear un productor.
        2. Crear un mensaje y adjuntarlo a su destino.
    2. 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)

Ejemplo PTP - Productor

```
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 el productor, enviamos el mensaje
            producer.send(message);
            System.out.println("Enviado mensaje [" + mundo + "]");
        } finally {
            // Cerramos los recursos
            producer.close();
            session.close();
            connection.close();
        }
    }
```

Ejemplo PTP – Consumidor Síncrono

```
public class Consumidor {
    @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 productor a partir de una cola
            consumer = session.createConsumer(queue);
            // Obtenemos 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 {
            consumer.close();
            session.close();
            connection.close();
        }
    }
```

Ejemplo PTP – Consumidor Asíncrono

```
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();
       }
   }
```

Ejemplo PTP – *Listener* de Mensajes

```
public class TextoListener implements MessageListener {

    /**
    * Casts del mensaje a un mensaje de texto y se muestra por consola
    * @param message mensaje de entrada
    */
    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());
        }
    }
}
```

Ejemplo Pub/Sub - Publicador

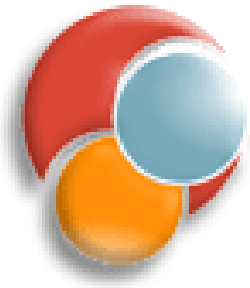
```
public class Publicador {
    @Resource(mappedName = "jms/ConnectionFactory")
    private static ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/Topic")
    private static Topic topic;

    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();
            connection.start();
            session = connection.createSession(esTransaccional, Session.AUTO_ACKNOWLEDGE);
            // Creamos el productor a partir de un topico
            publisher = session.createProducer(topic);
            message = session.createTextMessage(mundo);
            publisher.send(message);

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

Ejemplo Pub/Sub – Subscriptor Síncrono

```
public class Subscriptor {
    @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();
            connection.start();
            session = connection.createSession(esTransaccional, Session.AUTO_ACKNOWLEDGE);
            // Creamos el consumidor a partir de una cola
            subscriber = session.createConsumer(topic);
            // Obtenemos el mensaje
            message = (TextMessage) subscriber.receive();

            System.out.println("Recibido sincrono [" + message.getText() + "]");
            System.out.println("Fin sincrono");
        } finally {
            subscriber.close();
            session.close();
            connection.close();
        }
    }
```

Ejemplo Pub/Sub – Subscriptor Asíncrono

```
public void recibeMensajeAsincronoTopico() throws JMSException {
       Connection connection = null;
       Session session = null;
       MessageConsumer subscriber = null;
       TextoListener listener = null;
       boolean esTransaccional = false;

       try {
           connection = connectionFactory.createConnection();
           session = connection.createSession(esTransaccional, Session.AUTO_ACKNOWLEDGE);
           subscriber = session.createConsumer(topic);
           // Creamos el listener, y lo vinculamos al subscriptor -> asincrono
           listener = new TextoListener();
           subscriber.setMessageListener(listener);

           connection.start();
           System.out.println("Fin asincrono");
       } finally {
           subscriber.close();
           session.close();
           connection.close();
       }
   }
```



¿Preguntas...?