



5.USING JAVA MESSAGE SERVICE API

5.1. Using Java Message Service API

Objectives

After completing this lesson, you should be able to:

- Describe Java Message Service (JMS) API messaging models
- Implement Java SE and Java EE message producers and consumers
- Use durable and shared topic consumer subscriptions
- Create message-driven beans
- Use transactions with JMS



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

API de servicio de mensajes de Java. En este capítulo, veremos los modelos de mensajería de la API de JMS y cómo producimos y consumimos mensajes, y veremos los dos tipos de consumidores y productores: los que estaban en Java EE pero también los que no lo están. , así que fuera del contenedor Java EE también. Analizamos tipos específicos de consumo de mensajes, como, por ejemplo, suscripciones duraderas o compartidas, y también analizaremos beans controlados por mensajes como un tipo de consumidor de mensajes.

Ah, y una cosa más, por supuesto, si estamos hablando de transacciones, los componentes controlados por mensajes también pueden participar en las transacciones. Entonces, ¿cómo usamos las transacciones con JMS? Y ese es otro punto.

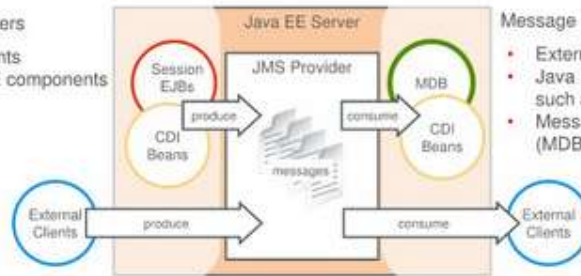
Java Message Service (JMS) API

JMS organizes message exchanges between producers and consumers via an intermediary.

- A JMS provider is a messaging system that implements the JMS interfaces and provides administrative and control features.
- An implementation of the Full Java EE platform includes a JMS provider.

Message Producers

- External clients
- Any Java EE components



Copyright © 2018 Oracle and/or its affiliates. All rights reserved.

The primary reason for using Java Message Service is to define a service for asynchronous request processing. Further, JMS provides a mechanism to allow multiple parties to retrieve messages based on subscription.

Despite the word Java in Java Message Service, the other benefit of message services is that there is no requirement that both the producers and consumers be written in the same language, or run on the same platform.

JMS represents a very loosely coupled type of interaction between components - generally producers and consumer don't have to be aware of each other.

A JMS application is composed of the following parts:

- **JMS provider:** A messaging system that implements the JMS API in addition to other administrative and control functionality required of a full-featured messaging product
- **JMS clients:** Programs that send and receive messages
- **Messages:** Objects that are used to communicate information between the clients of an application
- **Administered objects:** Provider-specific objects that clients look up and use to interact portably with a JMS provider

A Message-Driven Bean is a Java EE EJB component that is specifically designed to represent an Asynchronous Message Consumer:

- Can be subscribed to receive messages from a Queue or a Topic
- Can never be directly invoked by a client
- Typically, responsible for acquiring, validating, and preparing messages before passing them on to the Business Logic handling classes

Message-producer clients create and send messages to destinations.

Message-consumer clients consume messages from destinations. Message consumers can be either asynchronous or synchronous:

- **Asynchronous consumer clients** register with the message destination. When a destination receives a message, it notifies the asynchronous consumer, which then collects the message.
- **Synchronous message consumer clients** collect messages from the destination. If the destination is empty, the client blocks until a message arrives.

JMS API nos permite organizar intercambios de mensajes entre diferentes partes. Cualquier parte puede publicar y suscribirse a los mensajes. Pueden ser clientes externos. Podrían ser beans de sesión. Podrían ser frijoles CDI. Como consumidor del mensaje, puede usar un bean controlado por mensajes, que es un tipo especial de bean diseñado específicamente para consumir mensajes, pero también puede consumir mensajes de otros componentes.

Los mensajes se intercambian a través del servidor JMS. Es parte de su Java EE. Por cierto, requiere que su servidor Java EE admita el perfil completo, por lo que es una especificación completa para poder ejecutar JMS. Y permite que todo tipo de componentes intercambien estos mensajes.

JMS Destination Types

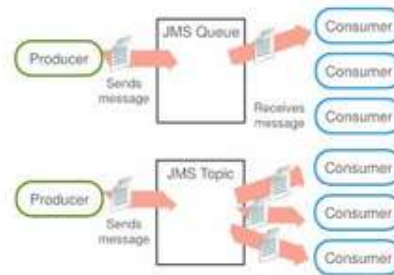
JMS Destinations are Queues and Topics through which messages are exchanged

Point-to-Point (Queue) Model

- Queue retains messages until they are consumed.
- Each message is consumed by one consumer.
- Messages may be "peeked at" by Queue Browser

Publish-Subscribe (Topic) Model

- Each message can have multiple consumers.
- Message may have no consumers, unless consumer uses durable subscription (identified by subscription name).
 - Durable subscriptions guarantee that all messages sent to the topic are received, even if there are no subscribers to the topic.
 - Nondurable subscriptions exist only for as long as there is an active consumer on the subscription. Any message sent to the topic when there are no subscribers is lost.



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

JMS supports two styles of messaging:

- **Point-to-point (PTP):** Messaging by using queues
- **Publish/subscribe:** Messaging by using topics

In a point-to-point scenario, producers address messages to a queue. Point-to-point messaging has the following characteristics:

- A message queue retains all messages until they are consumed.
- There are no timing dependencies between the sender and receiver.
- A receiver gets all the messages that were sent to the queue—even those that were sent before the creation of the receiver.
- The queue then deletes the messages on acknowledgement of successful processing from the message consumer.
- The distinguishing factor between the point-to-point and publish/subscribe architectures is that in most cases, each point-to-point message queue has only one message consumer. However, a message queue can have multiple consumers. When one consumer consumes a message from the queue, the message is marked as consumed. Other consumers cannot also consume the same message.

In a publish/subscribe scenario, producers address messages to a topic, which functions somewhat like a bulletin board. Publishers and subscribers can dynamically publish or subscribe to the topic. The system takes care of distributing the messages that arrive from a topic's multiple publishers to its multiple subscribers. Topics retain messages only as long as it takes to distribute them to subscribers.

Publish/subscribe messaging has the following characteristics:

- Each message can have multiple consumers.
- A client that subscribes to a topic will only receive messages sent to that topic after the client creates the subscription. Messages sent to the topic before the client subscribes are lost. However, JMS API relaxes this requirement to some extent by allowing applications to create durable subscriptions, which receive messages sent while the consumers are not active.

Ahora los intercambios de mensajes podrían seguir dos tipos de modelos, conocidos como punto a punto, también conocido como cola, y publicación-suscripción, también conocido como tema. El modelo Queue le permite entregar el mensaje a la cola, poner el mensaje en la cola y actúa como una pila de cartas donde el consumidor del mensaje toma el mensaje de la cola y eso es todo. Se fue. Sacas la carta de la pila. Se ha ido de la cola. Entonces, a medida que los mensajes se consumen de la cola, desaparecen. Cada mensaje será consumido por un consumidor en un modelo de cola.

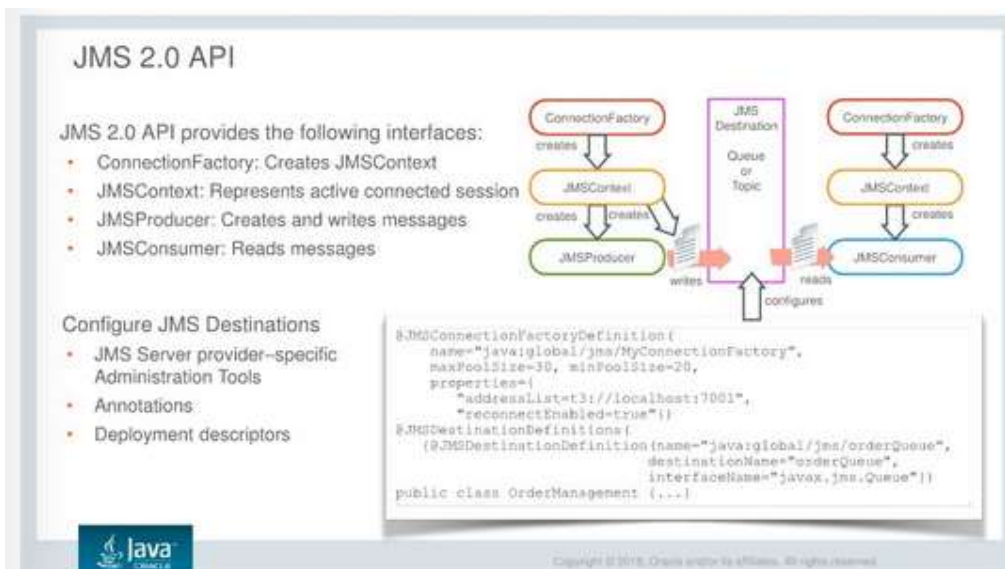
Hay una permutación de ese tipo de consumo de mensajes, que se llama navegador de cola, que es un tipo de consumidor que en realidad no saca mensajes de la cola. Solo está mirando los mensajes en una cola. Realmente no sacarlos. Eso es un navegador. Pero de lo contrario, si consume el mensaje de la cola, se habrá ido de la cola. Eso es todo.

Un tema es más como una valla publicitaria. Pega el mensaje en la cartelera, en el tema, para que lo vean todos los que están mirando el tema. ¿Derecha? Entonces el mensaje es visible para todos los consumidores de ese tema en particular.

Sin embargo, hay un poco complicado con un tema. Si coloca el mensaje en la cola y no hay un consumidor inmediato disponible que esté escuchando lo que entra en esa cola, el mensaje permanecerá en la cola hasta que haya un consumidor. Está bien. Tarde o temprano, el consumidor de la cola surgirá y captará el mensaje.

Pero con un tema, el mensaje no se queda indefinidamente. Hay un período de tiempo en el que se retiene un mensaje y luego tiene que desaparecer. Entonces, si no hay consumidores activos en un tema, lo que teóricamente podría suceder es que pones un mensaje en el tema, nadie lo miró, el mensaje desapareció. Bueno, después de un tiempo, eventualmente desaparecerá.

Para evitar perder mensajes temáticos como estos, puede crear una suscripción duradera. Si hay al menos un consumidor en un tema que dice el tema, soy un consumidor duradero, soy un suscriptor duradero, eso le indicará al tema que mantenga este mensaje hasta que ese suscriptor duradero lo reciba. Entonces, una especie de garantía de que hay al menos un suscriptor que lo obtendrá, incluso si ese suscriptor en ese momento estaba ocupado con otra cosa, inactivo o no disponible. De lo contrario, si se trata de una suscripción no duradera, es posible que pierda mensajes en un tema si no tiene cuidado al respecto.



JMS API provides following interfaces:

- **ConnectionFactory**: An administered object that is used by a client to create a JMSContext
- **JMSContext**: An active connection to a JMS provider and a single-threaded context for sending and receiving messages
- **JMSProducer**: An object created by a JMSContext that is used for sending messages to a queue or topic
- **JMSConsumer**: An object created by a JMSContext that is used for receiving messages sent to a queue or topic

Administered objects are objects that are configured administratively (as opposed to programmatically) for each messaging application. A JMS provider supplies the administered objects.

There are two types of JMS-administered objects:

- **ConnectionFactory**: The object that a client uses to create a connection with a JMS provider
- **Destination**: The object that a client uses to specify the destination of the messages that it is sending and the source of the messages that it receives

JMS Connection Factories, Queues, and Topics can be configured via server administrative tools, annotations, and deployment descriptors.

JMS providers differ significantly in their implementations of the underlying messaging technology. There are also major differences in how a JMS provider's system is installed and administered.

For JMS clients to be portable, they must be isolated from the proprietary aspects of a provider. This is done by defining JMS-administered objects that are created and customized by a provider's administrator and later used by clients. The client uses them through JMS interfaces that are portable. The administrator creates them by using provider-specific facilities.

Administered objects are placed in a Java Naming and Directory Interface (JNDI) namespace by an administrator. A JMS client typically notes in its documentation the JMS-administered objects it requires and how the JNDI names of these objects should be provided to it.

The two types of administered objects are:

- **Destinations:** They are message distribution points. Destinations receive, hold, and distribute messages. Destinations fall into two categories: queue and topic destinations.
 - Queue destinations implement the point-to-point messaging protocol.
 - Topic destinations implement the publish/subscribe messaging protocol.
- **Connection factories:** They are used by a JMS API client (JMS client) to create a connection to a JMS API destination (JMS destination).

Ha habido muchos cambios entre la versión anterior de Java EE y Java EE 7 con respecto a JMS. Se ha introducido una nueva API JMS 2.0 como un cambio significativo. En primer lugar, si alguna vez trabajó con la versión anterior, por si acaso, estaba estableciendo su conexión, estaba estableciendo la sesión, luego estaba creando el consumidor o productor de mensajes.

En la nueva API, y este curso nos presenta una nueva API, los ejemplos de API antiguos están disponibles en las notas, pero no se presentan como parte del curso. Entonces, la nueva API lo simplifica drásticamente. Básicamente, tiene un solo objeto que controla todos los aspectos del manejo de mensajes JMS y se llama Contexto JMS. Con un objeto de contexto, puede producir mensajes. Con un objeto de contexto, puede crear productores y consumidores.

Además del objeto de contexto, es posible que también necesite una fábrica de conexiones, pero, curiosamente, un nuevo estándar JMS, el estándar Java EE 7, en realidad define una fábrica de conexiones predeterminada para que no necesite crear una. Si desea personalizar lo que es la fábrica de conexiones, seguramente puede crear una, pero hay una llamada Fábrica de conexiones predeterminada que se proporciona de fábrica. Así que es posible que no te molestes en crear uno.

Connection Factory puede ser útil si desea personalizar la forma en que se conecta al servidor JMS, como crear y administrar grupos de conexiones, etc., pero eso es solo si desea personalizarlo. Además, tenga en cuenta que ahora tenemos una forma estándar de describir fábricas de conexiones, colas y temas, todos estos artefactos, a través de anotaciones. No tiene que hacerlo a través de la configuración específica del proveedor, aunque aún podría hacerlo a través de la configuración específica del proveedor. Podrías ir a la consola de WebLogic Server y configurar estas cosas. Pero en realidad, podría simplemente anotar su código. Ese es un ejemplo. Creación de colas, temas y fábricas de conexiones solo con anotaciones.

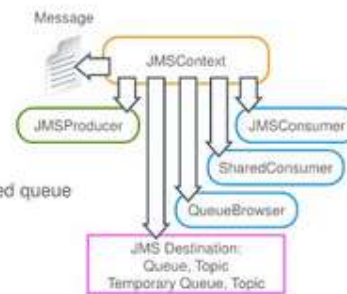
Depende de lo que quieras lograr. Si desea preconfigurar estos objetos en un servidor, está bien. Si desea proporcionar la configuración con su implementación a través de la notación o el descriptor de implementación, también está bien. Puedes hacerlo de cualquier manera.

JMS Context

JMSContext Object represents connected JMS session.

It is used to:

- Create Byte, Map, Object, Stream, and Text messages
- Create JMS Producers, Consumers and Queue Browsers
 - JMS Producer: To post messages
 - JMS Consumer: To receive messages
 - Shared Consumer
 - Queue Browser - To peek at the messages on the specified queue
- Create JMS Destinations
 - Queue or Topic
 - Temporary Queue or Topic
- Set up Exception Handlers
- Control Bean Managed Transactions
- Control subscriptions and delivery of messages



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

JMSContext object provides a number of message creation methods:

- `createMessage()`
- `createBytesMessage()`
- `createMapMessage()`
- `createObjectMessage()`
- `createObjectMessage(Serializable object)`
- `createStreamMessage()`
- `createTextMessage()`
- `createTextMessage(String text)`

For more information on JMS Messages see:

<https://docs.oracle.com/javasee/7/api/javax/jms/Message.html>

JMSContext object creates JMS Producer objects:

- `createProducer()`

JMSContext object creates JMS Consumer objects:

- `createConsumer(Destination destination)`
- `createConsumer(Destination destination, String messageSelector)`
- `createConsumer(Destination destination, String messageSelector, boolean noLocal)`
- `createDurableConsumer(Topic topic, String subscriptionName)`
- `createDurableConsumer(Topic topic, String subscriptionName, String messageSelector, boolean noLocal)`

JMSContext object creates Shared Consumer objects:

- `createSharedConsumer(Topic topic, String sharedSubscriptionName)`
- `createSharedConsumer(Topic topic, String sharedSubscriptionName, String messageSelector)`
- `createSharedDurableConsumer(Topic topic, String sharedSubscriptionName)`
- `createSharedDurableConsumer(Topic topic, String sharedSubscriptionName, String messageSelector)`

JMSContext object creates QueueBrowser objects to peek at the messages on the specified queue:

- `createBrowser(Queue queue)`
- `createBrowser(Queue queue, String messageSelector)`

JMSContext object creates JMS Destinations:

- `createQueue(String queueName)`
- `createTopic(String topicName)`
- `createTemporaryQueue()`
- `createTemporaryTopic()`

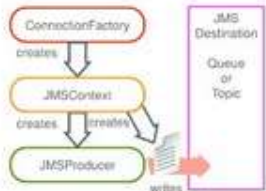
Ahora ese objeto clave crucial llamado Contexto JMS, a partir del objeto Contexto JMS, crea todas las demás cosas en la API JMS 2.0. Usted crea consumidores, productores, diferentes tipos de consumidores, como consumidores compartidos para el tema o navegador de colas. Recuerde, ese es el que le permite echar un vistazo a la cola sin sacar mensajes de la cola. Podría describir destinos JMS. Puede crear mensajes reales.

Los mensajes pueden estar en diferentes formatos: byte, mapa, objetos, texto, cualquier cosa. Y luego, por supuesto, puede publicar mensajes, eso es a través del productor JMS, recibir mensajes, eso es a través de varios tipos de consumidores, y manejar excepciones, manejar transacciones y controlar esa suscripción y entrega de mensajes.

Java SE Message Producer

Java SE Message Producer:

- Obtains JNDI Initial Context object
- Looks up **ConnectionFactory**, **Queue**, or **Topic** resources
- Creates **JMSContext** object
- Creates **JMSProducer** object
- Optionally, sets producer properties (see notes)
- Prepares and sends messages



```

public class OrderProducer {
    public static void main(String[] args) {
        Context ctx = new InitialContext();
        ConnectionFactory cf = (ConnectionFactory)ctx.lookup("java:comp/DefaultJMSConnectionFactory");
        Queue queue = (Queue)ctx.lookup("java:global/jms/orderQueue");
        try {
            JMSContext context = cf.createContext();
            JMSProducer producer = context.createProducer(queue);
            Message msg = context.createMessage();
            producer.setDeliveryMode(DeliveryMode.PERSISTENT).setTimeToLive(1000).send(msg);
        } catch (JMSException e) { ... }
    }
}

```

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Java SE client require server-specific libraries in their classpath.

- Glassfish JMS client uses `gfclient.jar` and `app client.jar`.
- WebLogic JMS client uses `wlthint3client.jar`

There are several overloaded versions of the send method available in the JMSProducer:

`send(Destination destination, byte[] body):` Sends a `BytesMessage`

`send(Destination destination, Map<String, Object> body):` Sends a `MapMessage`

`send(Destination destination, Message message):` Sends a `javax.jms.Message` object (it can contain headers and body parts, where headers contain values used by both clients and providers to identify and route messages and body could be a `Stream`, `Map`, `Text`, `Object` or `Bytes` content)

`send(Destination destination, Serializable body):` Sends an `ObjectMessage`

`send(Destination destination, String body):` Sends a `TextMessage`

You can set various standard as well as custom properties for the JMSProducer, such as:

JMS supports two modes of message delivery.

- The `NON_PERSISTENT` mode is the lowest overhead delivery mode because it does not require that the message be logged to stable storage. A JMS provider failure can cause a `NON_PERSISTENT` message to be lost.
- The `PERSISTENT` mode instructs the JMS provider to take extra care to ensure the message is not lost in transit due to a JMS provider failure.

`setTimeToLive(long timeout)` is used to determine the expiration time of a message.

- Specifies the time to live of messages that are sent using this JMSProducer. Value is in milliseconds; a value of zero (default) means that a message never expires.
- The expiration time of a message is the sum of the message's time to live and the time it is sent. For transacted sends, this is the time the client sends the message, not the time the transaction is committed.
- Clients should not receive messages that have expired; however, JMS does not guarantee that this will not happen.
- A JMS provider should do its best to accurately expire messages; however, JMS does not define the accuracy provided. It is not acceptable to simply ignore time-to-live.

setDeliveryDelay(long deliveryDelay): Sets the minimum length of time in milliseconds that must elapse after a message is sent before the JMS provider may deliver the message to a consumer.

setPriority(int priority): Specifies the priority of messages that are sent using this JMSProducer

The JMS API defines 10 levels of priority value, with 0 as the lowest priority and 9 as the highest.

Clients should consider priorities 0-4 as gradations of normal priority and priorities 5-9 as gradations of expedited priority. Priority is set to 4 by default.

You can also reduce message processing overhead by disabling Message Ids and Message Timestamps, in case your application does not need to use these properties to handle messages.

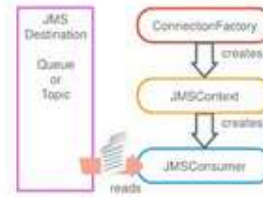
Echemos un vistazo a un productor. Esta página nos muestra el productor de mensajes Java SE. Habrá una página ligeramente diferente un poco más tarde mostrando Java EE Producer. Entonces, SE Producer necesita establecer el contexto JNDI y, con eso, buscar Connection Factory y los objetos de cola. Pues cola o tema. Lo que sea. Este ejemplo es con una cola. Cree un objeto JMS Context, cree un mensaje y prodúzcalo. Ahí tienes

Cuando produce el mensaje, puede controlar cómo lo hace exactamente con esta imitación de método de cadena, punto, punto, punto. Entonces, cuando envía ese mensaje, puede decir, ¿cuál es el modo de entrega? Como persistente, transitorio. ¿Quiere que JMS guarde el mensaje? ¿Y cuál es el tiempo para que el mensaje salga en este sistema? Tiempo fuera para ello, etcétera. OK, así que publicar el mensaje.

Java SE Message Consumer

Java SE Message Consumer:

- Obtains JNDI Initial Context object
- Looks up **ConnectionFactory**, **Queue**, or **Topic** resources
- Creates **JMSContext** object
- Creates **JMSConsumer** object
- Receives messages



```
public class OrderConsumer {
    public static void main(String[] args) {
        Context ctx = new InitialContext();
        ConnectionFactory cf = (ConnectionFactory)ctx.lookup("java:comp/DefaultJMSConnectionFactory");
        Queue queue = (Queue)ctx.lookup("java:global/jms/orderQueue");
        try {
            JMSContext context = cf.createContext();
            JMSConsumer consumer = context.createConsumer(queue);
            Message message = consumer.receive(1000);
        } catch (JMSException e) { ... }
    }
}
```

JMS consumer provides several ways of receiving messages from JMS Queues and Topics:

- Method `receive()`: Receives the next message produced for this message consumer
 - This call blocks indefinitely until a message is produced or until this message consumer is closed.
 - If this receive is done within a transaction, the consumer retains the message until the transaction commits.
- Method `receive(long timeout)`: Receives the next message that arrives within the specified timeout interval
 - Parameter `timeout` sets the timeout value in milliseconds.
 - This call blocks until a message arrives, the timeout expires, or this message consumer is closed.
 - A timeout of zero never expires, and the call is blocked indefinitely.
- Method `receiveNoWait()`: Receives the next message if one is immediately available
 - It returns the next message produced for this JMSConsumer, or null if one is not available.

These operations return the next message produced for this message consumer, or null if this message consumer is concurrently closed.

They could throw `JMSExceptions`, if the JMS provider fails to receive the next message due to some internal error.

JMSConsumer defines three equivalent overloaded operations called `receiveBody` that allow retrieving just the message body, without analyzing headers or properties of the message.

Consumidor. Este es el consumidor de Java SE. El Consumidor Java EE es un bean controlado por mensajes, descubierto un poco más tarde. Ahora Java SE Message Consumer hará lo mismo: establecer un objeto de contexto inicial, buscar Connection Factory, buscar la cola, crear contexto JMS, crear un consumidor y recibir un mensaje. Esta propiedad de recepción describe un tiempo de espera.

Le está pidiendo a este consumidor en particular que espere un período de tiempo para que llegue un mensaje a esa cola. Si no llega, bueno, no llega. Así que deja de esperarlo. Entonces, el método Recibir es básicamente una forma de consumir mensajes sincrónicamente. ¿OK?

Probablemente, un enfoque más flexible será la producción de mensajes asíncronos y el consumo de mensajes asíncronos. Eso es ciertamente posible. Así es como se hace.

Java SE Asynchronous Producers and Consumers

Java SE JMS Producer may send messages asynchronously

- While JMSProducer handles the send operation, the program can perform other actions.
- CompletionListener operations will be invoked when the send operation finishes.

```
CompletionListener listener = new CompletionListener() {  
    public void onCompletion(Message message) {...}  
    public void onException(Message message, Exception exception) {...}  
};  
context.createProducer().setAsync(listener).send(jmsDestination, message);
```

Java SE JMS Consumer may receive messages asynchronously.

- While JMSConsumer handles the receive operation, the program can perform other actions.
- MessageListener onMessage operation will be invoked when the message arrives.

```
MessageListener listener = new MessageListener() {  
    public void onMessage(Message message) {...}  
};  
context.createConsumer().setMessageListener(listener);
```



Copyright © 2015. Oracle and/or its affiliates. All rights reserved.

Java SE application must prepare all JMS resources (ConnectionFactory, JMSContext, JMSProducer) in the same way no matter if it is going to use synchronous or asynchronous message delivery mode.

Sending messages asynchronously allows you application not to wait for send method to return, confirming that the message has been delivered successfully, or throw an exception if there was a problem. This enables you application to perform other tasks, or send more messages, while given message delivery is still in-progress. In order to find out if the message has been delivered successfully or not you must override two methods defined by the `javax.jms.CompletionListener`

- **onCompletion** - this method notifies the application that the message has been successfully sent
- **onException** - this method notifies the application that the exception was thrown while attempting to send specified message

In order to prevent asynchronous JMS producer from getting into indefinite loop trying to redeliver a faulty message, you should also check how many times the message has been redelivered:

```
int deliveryCount = message.getIntProperty("JMSXDeliveryCount");
```

Si desea producir un mensaje de forma asíncrona, debe crear un objeto de escucha de finalización, con dos métodos: `onCompletion`, `onException`. Y cuando registra el productor contra el objeto JMS Context, básicamente configura ese oyente asíncrono aquí como un argumento y luego llama al método `Enviar`. Entonces, lo que sucede es que el método `Enviar` envía el mensaje a ese destino, a esa cola, tema, lo que sea, y permite que su código continúe. No está esperando que un mensaje enviado realmente termine de enviarlo. Solo dispara y olvida.

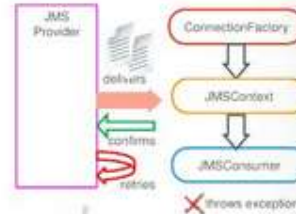
Y luego, más adelante, cuando el servidor JMS realmente averigüe qué sucede con ese mensaje, si se produce, se consume, lo que sea, así que pase lo que pase, llamará a uno de estos métodos. `onCompletion` si todo salió bien o `onException` si tuvo un problema. Para que pueda averiguar más tarde lo que está sucediendo. ¿OK?

Ahora el consumo de mensajes asíncronos, registra un escucha de mensajes, anula todos los métodos de mensajes y lo establece en el objeto JMS Context. Ahí está. Sí, configura el detector de mensajes. Eso es todo. Se le llamará, ese método `onMessage` se invocará cuando el mensaje llegue a cualquier cola o tema con el que haya asociado ese objeto JMS Context. ¿OK?

JMS Session Modes and Message Acknowledgments

JMSContext session modes:

- AUTO_ACKNOWLEDGE (default)
- DUPS_OK_ACKNOWLEDGE
- CLIENT_ACKNOWLEDGE
- SESSION_TRANSACTED



```

ConnectionFactory cf = ...
JMSContext context = cf.createContext(JMSContext.SESSION_TRANSACTED);
  
```

Message delivery can be confirmed explicitly, or in transacted session scenario messages are acknowledged implicitly with transaction commit. Throwing runtime exception may cause message to be redelivered depending on the session mode (see notes).

```

context.acknowledge();
context.recover();
context.commit();
context.rollback();
  
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

Message delivery can be acknowledged explicitly. However, for transacted sessions commit would implicitly acknowledge processed messages.

- **acknowledge()** - Acknowledges all messages consumed by the JMSContext's session.
- **commit()** - Commits all messages done in this transaction and releases any locks currently held.
- **recover()** - Stops message delivery in the JMSContext's session, and restarts message delivery with the oldest unacknowledged message.
- **rollback()** - Rolls back any messages done in this transaction and releases any locks currently held.

The result of a listener throwing a RuntimeException depends on the session's acknowledgment mode.

- **AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE** - the message will be immediately redelivered. The number of times a JMS provider will redeliver the same message before giving up is provider-dependent. The JMSRedelivered message header field will be set, and the JMSXDeliveryCount message property incremented, for a message redelivered under these circumstances.
- **DUPS_OK_ACKNOWLEDGE** - instructs the JMSContext's session to lazily acknowledge the delivery of messages. This is likely to result in the delivery of some duplicate messages if the JMS provider fails, so it should only be used by consumers that can tolerate duplicate messages. Use of this mode can reduce session overhead by minimizing the work the session does to prevent duplicates.

- **CLIENT_ACKNOWLEDGE** - the next message for the listener is delivered. If a client wishes to have the previous unacknowledged message redelivered, it must manually recover the session.
- **Transacted Session** - the next message for the listener is delivered. The client can either commit or roll back the session (in other words, a RuntimeException does not automatically rollback the session).

JMS providers should flag clients with message listeners that are throwing RuntimeException as possibly malfunctioning.

Cuando consume mensajes, puede reconocer que los ha consumido y hay varias maneras diferentes en las que puede administrar ese proceso. El modo predeterminado en el que funciona el objeto JMS Context se llama AUTO_ACKNOWLEDGE, por lo que realiza automáticamente los reconocimientos de mensajes.

Pero si desea controlar el proceso desde su código, puede cambiar ese modo a CLIENT_ACKNOWLEDGMENT.

CLIENT_ACKNOWLEDGMENT significa que puede usar dos métodos llamados Reconocimiento y Recuperación. Si llama al método Reconocimiento, le está diciendo al servidor que todo está bien. Consumió el mensaje. Bien. Si llama al método Recover, entonces le está diciendo al servidor JMS, ¿puedo intentar consumir ese mensaje

nuevamente, por favor? tengo un problema Quiero intentarlo de nuevo. ¿Sí? Esa es la idea de Recuperar. Es básicamente un reintento, ¿sí?

Si le dice al contexto JMS que desea establecer una sesión transaccionada, no reconoce los mensajes individuales: confirma la transacción. Al confirmar la transacción, reconoce automáticamente cualquier otra cosa que haya sucedido en esa transacción, cualquier mensaje que haya logrado consumir hasta ahora. Todo de una vez. O si revierte la transacción, obviamente eso es un equivalente de recuperación, si lo desea. Sí, estás tratando de decirle al servidor JMS que tengo un problema. ¿Puedo intentarlo de nuevo? Así que esa es la sesión transaccional que vincula la confirmación del mensaje a la transacción.

DUPS_OK es una ligera variación en el modo AUTO_ACKNOWLEDGE, y básicamente, le dice al servidor que no rastree, no rastree las confirmaciones de mensajes individuales. Puede conducir a una circunstancia extraña. Si el servidor falla y luego se reinicia, en teoría, lo que podría suceder es que podría entregar el mensaje dos veces. Entonces, un determinado mensaje, el servidor no se dará cuenta de que ya ha sido reconocido. En el caso de un accidente, esa podría ser la situación. Entonces, si su código es tolerante a los mensajes duplicados, si no sucederá nada malo si procesa el mismo mensaje dos veces, DUPS_OK mejorará el rendimiento del servidor al no indicarme que rastree información adicional. De lo contrario, AUTO_ACKNOWLEDGE es la opción más segura, que es la predeterminada de todos modos.

Handle JMS Messages

JMS Message Producer

- Create Message of specific type:
 - ByteMessage
 - MapMessage
 - ObjectMessage
 - StreamMessage
 - TextMessage
- Set message properties

```
ObjectMessage msg = context.createObjectMessage(product);
msg.setStringProperty("name", product.getName());
msg.setFloatProperty("price", product.getPrice());
producer.send(orderQueue, msg);
```

JMS Message Consumer

- Retrieves properties
- Checks and casts message type to:
 - ByteMessage
 - MapMessage
 - ObjectMessage
 - StreamMessage
 - TextMessage

```
public void onMessage(Message msg) {
    try {
        String name = msg.getStringProperty("product");
        float price = msg.getFloatProperty("price");
        if (msg instanceof ObjectMessage) {
            ObjectMessage objMsg = (ObjectMessage)msg;
            Product product = (Product)objMsg.getObject();
        }
        catch (Exception e) {...}
    }
}
```

Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Ahora, cuando maneja mensajes JMS, lo que hace con los mensajes depende de su tipo. Puede tener mensajes binarios, de objeto, flujo, texto, todo tipo de mensajes. También puede redactar mensajes con cualquier cantidad de propiedades arbitrarias. Entonces, en este caso particular, tengo aquí el par de propiedades: algunas cadenas y algunas propiedades flotantes. no sé, cualquier cosa.

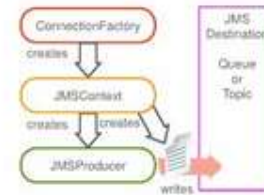
Parece que es un mensaje de objeto. Estoy teniendo un producto aquí. Objeto de producto. Y estoy usando algunos de estos productos como propiedades. ESTÁ BIEN. Puedo establecer las propiedades de un mensaje... Puede obtener las propiedades del mensaje. El mensaje contendrá el producto completo de todos modos, pero la propiedad es como un par de valores de nombre separados que le gustaría asociar con un mensaje. Hay algunas propiedades JMS estándar mantenidas por el servidor JMS y luego hay otras personalizadas, como la que observa aquí, donde puede crear simplemente porque desea propiedades adicionales.

Cómo puede utilizar las propiedades es bastante interesante. Veremos el ejemplo de la utilización de la propiedad un poco más adelante.

Java EE Message Producer

Java EE Message Producer:

- May simply inject **JMSContext** object
 - **ConnectionFactory** can be automatically provided by container.
- Injects **Queue or Topic** Resource
- Creates **JMSProducer** object
- Optionally, sets delivery properties and headers, such as delay
- Uses JMS Producer to send messages



```
public class OrderManagement {
    @Inject
    private JMSContext context;
    @Resource(lookup="jms/orderQueue");
    private Queue orderQueue;

    public void sendOrderMessage(Order order) {
        ObjectMessage msg = context.createObjectMessage(order);
        context.createProducer().setDeliveryDelay(20000).send(orderQueue, msg);
    }
}
```



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

The `@Inject` annotation tells the container to create the `JMSContext` when it is needed.

Use `@Inject` to inject the `JMSContext`, specifying the connection factory that you want to use. The container automatically looks up this connection factory for you and uses it to create a `JMSContext`. Your code simply must use it. At the end of the transaction, the `JMSContext` is closed automatically for you by the container.

If you want to specify a connection factory, you do so via annotation:

```
@Inject
@JMSConnectionFactory("java:comp/MyJMSConnectionFactory")
private JMSContext context;
```

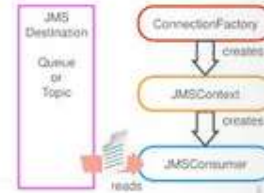
Productor de mensajes Java EE. Hemos visto el productor de mensajes de Java SE. Eso es un productor de mensajes Java EE. ¿Ver la diferencia? Esa es la diferencia.

Simplemente inyecta contexto JMS y dice a qué cola o tema desea suscribirse. El contenedor proporciona automáticamente Connection Factory. Realmente no necesitas hacer nada más. El resto del código es igual. ESTÁ BIEN. Para que puedas enviar tus mensajes.

Java EE Message Consumer

Java EE Message Consumer:

- May simply inject **JMSContext** object
 - **ConnectionFactory** can be automatically provided by container.
- Injects **Queue or Topic** Resource
- Creates **JMSConsumer** object
- Uses JMSConsumer to receive messages



```
@RequestScoped
public class InvoiceManagement {
    @Inject
    private JMSContext context;
    @Resource(lookup="jms/orderQueue");
    private Queue orderQueue;
    public void receiveOrderMessage() {
        try {
            JMSConsumer consumer = context.createConsumer(orderQueue);
            Message msg = consumer.receive(1000);
        } catch (JMSException e) { ... }
    }
}
```

```
createDurableConsumer(Topic topic, String name)
```

Creates an unshared durable subscription on the specified topic (if one does not already exist) and creates a consumer on that durable subscription

```
JMSConsumer createDurableConsumer(Topic topic, String name, String
messageSelector, boolean noLocal)
```

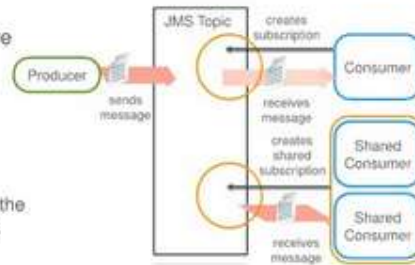
Creates an unshared durable subscription on the specified topic (if one does not already exist), specifying a message selector and the `noLocal` parameter, and creates a consumer on that durable subscription

Consumidor de mensajes Java EE. Normalmente usaríamos un bean controlado por mensajes. Veremos el bean controlado por mensajes en un momento, pero en este caso, estamos consumiendo el mensaje del bean CDI. La misma idea. Inyecta en contexto JMS, inyecta en una cola o tema en particular, y luego recibe el mensaje. Un consumo de mensajes simple y síncrono. Pero como digo, la mayoría de las veces, probablemente usaría un bean controlado por mensajes en su lugar, así que tendremos que ver eso en un momento.

Topics Shared/Unshared Subscriptions

You can have many subscriptions on a topic. Each message is copied to every **subscription** (unless there is a message selector).

- Normal (Unshared) subscription represents a single consumer.
- If the subscription is shared, it can have many consumers. Each message from the **subscription** is delivered to only one of these consumers.
- A shared **subscription** is identified by a name (and by the client ID if it is set), and can have multiple consumers.



```
...
JMSConsumer consumer=context.createSharedConsumer(topic,"SharedSubscriptionName");
...
```



Copyright © 2018, Oracle and/or its affiliates. All rights reserved.

JMSTopic object creates Shared Durable or Non-Durable Topic Consumer objects, with or without message selectors:

- `createSharedConsumer(Topic topic, String sharedSubscriptionName)`
- `createSharedConsumer(Topic topic, String sharedSubscriptionName, String messageSelector)`
- `createSharedDurableConsumer(Topic topic, String sharedSubscriptionName)`
- `createSharedDurableConsumer(Topic topic, String sharedSubscriptionName, String messageSelector)`

Ahora suscripción compartida o no compartida. La suscripción compartida es una característica de un tema. Lo que nos permite hacer, nos permite decir que, normalmente, en un escenario de suscripción no compartida, se muestra un mensaje que trata sobre el tema a cada consumidor, pero si dos consumidores usan una suscripción compartida, básicamente, dos consumidores identifican lo mismo. nombre de suscripción. Eso es todo lo que necesitan hacer. Por lo que dicen, estamos compartiendo el mismo nombre de suscripción.

Entonces se entregará un mensaje no a cada uno de estos consumidores sino a uno de ellos. Así que es un poco como una cola-ish. [? Bueno, dije, ?] mi mensaje se entrega solo a uno pero no a todos los componentes. Así que este es un truco que puedes jugar con un tema si quieres una suscripción compartida. Tan compartido por este nombre.

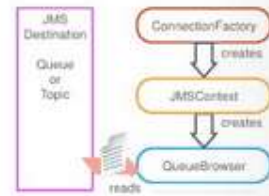
Queue Message Browser

Java EE Queue Message Browser:

- Injects **JMSContext** object
- Injects **Queue** Resource
- Creates **QueueBrowser** object
- Uses QueueBrowser to look at queue messages (they stay in the Queue)

```
@RequestScoped
public class InvoiceManagement {
    @Inject
    private JMSContext context;
    @Resource(lookup="jms/orderQueue");
    private Queue orderQueue;

    public void browseOrderMessages() {
        try {
            QueueBrowser browser = context.createBrowser(queue);
            Enumeration<Message> msgs = browser.getEnumeration();
            while (msgs.hasMoreElements()) {
                Message msg = msgs.nextElement();
                ...
            }
        } catch (JMSException e) { ... }
    }
}
```



Java SE Queue Message browser example:

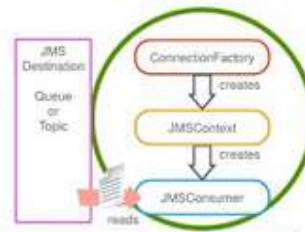
```
public class OrderProducer {
    public static void main(String[] args) {
        Context ctx = new InitialContext();
        ConnectionFactory cf =
        (ConnectionFactory) ctx.lookup("java:comp/DefaultJMSConnectionFactory");
        Queue queue = (Queue) ctx.lookup("java:global/jms/orderQueue");
        try {
            JMSContext context = connFactory.createContext();
            QueueBrowser browser = context.createBrowser(queue); {
                Enumeration msgs = browser.getEnumeration();
                while (msgs.hasMoreElements()) {
                    Message tempMsg = (Message) msgs.nextElement();
                    ...
                }
            }
        } catch (JMSException ex) { ... }
    }
}
```

El otro truco es hacer que una cola funcione un poco como un tema. Si consume el mensaje en una cola, lo está sacando de la cola. Se ha ido de la cola, excepto cuando usa un navegador de cola. Un navegador de cola le permite ver qué mensajes hay en una cola como una enumeración de mensajes y simplemente iterar a través de esa colección, pero no se eliminarán de la cola en sí. Todavía estarán en la cola.

Message-Driven Bean (MDB)

MDB is an asynchronous Java EE message consumer.

- Defined with @MessageDriven annotation, or using ejb-jar.xml
- Implements MessageListener interface
- Container manages connectivity, subscription, and message delivery for the MDB based on configuration provided
- Optionally, injects a MessageDrivenContext object in order to:
 - Access Security Properties
 - Control Transactions



```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName="destinationLookup",
        propertyValue="jms/productQueue"),
    @ActivationConfigProperty(propertyName="destinationType",
        propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName="messageSelector",
        propertyValue="product='tea' AND price > 2.95")})
public class ProductMessageHandler implements MessageListener {
    @Inject
    private MessageDrivenContext context;
    public void onMessage(Message message) {...}
}
```



Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Message-Driven bean requirements:

- MBD class must be public but not final or abstract.
- Annotate the message-driven bean class by using the MessageDriven metadata annotation.
- Optionally, use resource injection to obtain a MessageDrivenContext instance.
- Include a public, no-argument, empty constructor.
- Provide the onMessage method to consume messages.
- Do not define a finalize method.

Message-driven beans do not:

- Have remote component or local component interfaces
- Provide Java EE components with a direct client interface. Java EE technology clients communicate with a bean by sending a message to the destination (queue or topic) for which the bean is the MessageListener object.
- Have a client-visible identity. They do not hold client conversational states. They are anonymous to clients.
- Participate in a client's transaction because they have no client. The message sender's transaction and security contexts are unavailable to a message-driven bean. Consequently, message-driven beans must start their own transaction and security context.

@ActivationConfigProperty

The following standard properties are recognized for JMS message-driven beans:

- `acknowledgeMode` `Auto_acknowledge` (default) OR `Dups_ok_acknowledge`: This property is only used with Bean Managed Transactions. In Container Managed Transaction Scenario (default), container is responsible for message acknowledgement.
- `messageSelector`: Specifies which messages to receive using SQL-like syntax
 - If a message contains properties that match the selection criteria, it is delivered to the message-driven bean.
 - If the message does not match the criteria, the message-driven bean is not notified.
 - You can declare the JMS message selector by using the activation configuration property `messageSelector`.
- `destinationType` `javax.jms.Queue` or `javax.jms.Topic`
- `destinationLookup`: Specifies the queue or topic JNDI name
- `connectionFactoryLookup`: Specifies the JMS connection factory JNDI name
- `subscriptionDurability` `Durable` or `NonDurable`: Is used if the message-driven bean is intended to be used with a Topic, and the bean provider has indicated that a durable subscription should be used
- `subscriptionName` Specifies the name of the durable subscription
- `clientId`: Specifies the JMS client of the provider

JMS message-driven beans:

- They are annotated with `@javax.ejb.MessageDriven`.
- They implement the `javax.jms.MessageListener` interface.
- The application server can use integrated JMS support or use a connector.

Non-JMS message-driven beans:

- They are annotated with `@javax.ejb.MessageDriven`.
- They implement a message listener interface specific to the messaging service.
- The application server requires a connector.

Y, por último, el bean largamente esperado impulsado por mensajes. Te prometí eso y aquí viene. Puede anotarse con una anotación basada en mensajes o puede describirse en el archivo `ejb-jar.xml`. Implementa escucha de mensajes. Por lo tanto, anulamos el método `onMessage`, que acepta mensaje como argumento. Opcionalmente, podemos inyectarle contexto basado en mensajes.

La idea es que te suscribas a una cola en particular o a un tema a través del conjunto de anotaciones conocido como Configuración de activación. Así que estás creando estas propiedades de configuración de activación y eso es todo. El mensaje entra en esa cola o en ese tema y este bean recogerá ese mensaje.

Recuerde que mencionamos que los mensajes tienen propiedades. Este es uno interesante. Usando las propiedades como filtro o condición, se llama Selector de mensajes, básicamente digo que este bean no está recibiendo todos los mensajes. Estoy diciendo que solo está recibiendo té que tienen un precio superior a \$ 2.99. Es una especie de [¿Porsche?] té. Esa es la naturaleza de este frijol en particular. Así que está buscando ciertos tipos de objetos, pero no todos. Entonces podría usar las propiedades del mensaje por razones de filtración. Puede haber algún otro bean que busque todos los productos en esa cola, no lo sé, pero ciertamente es un enfoque interesante.

MDB Life Cycle

Message-Driven Bean lifecycle container callback operations:

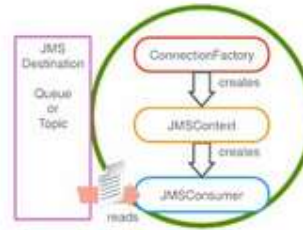
- PostConstruct is invoked when bean instance is created.
- PreDestroy is invoked when bean instance is disposed.

Message-Driven EJB Life Cycle



```

package mensaj;
// imports ...
@MessageDriven(...)
public class ProductMessageHandler implements MessageListener {
    @PostConstruct
    public void init() {...}
    @PreDestroy
    public void cleanup() {...}
    public void onMessage(Message message) {...}
}
  
```



Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Containers can create and pool multiple instances of message-driven beans to service the same message destination.

The operation of these instances is independent of each other, that is, Message-Driven Beans are stateless.

When a message arrives at a message destination, the container chooses the next available instance associated with that destination to service the message.

El ciclo de vida del bean controlado por mensajes es muy parecido a un bean de sesión sin estado. Dos estados-- Inexistente y Listo-- dos métodos que le permiten controlar su código cuando su bean pasa de un estado a otro-- postConstruct y preDestroy. Eso es todo.

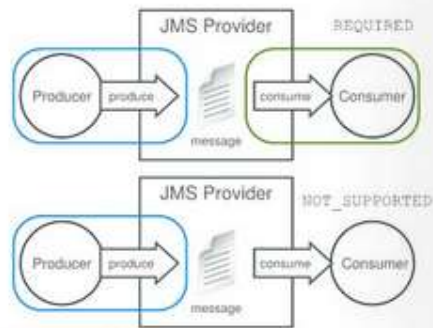
JMS and Transactions

The producer and consumer of messages do not share Security or Transaction Context.

- MDB can only use **REQUIRED** or **NOT_SUPPORTED** CMT Transactional semantics.
- For Transactional MDB, consider controlling number of delivery retries.

```

@MessageDriven(activationConfig = {
    ...
    @ActivationConfigProperty(
        propertyName="messageSelector",
        propertyValue="JMSXDeliveryCount < 3")
    @TransactionManagement(CONTAINER)
    public class ProductMessageHandler implements MessageListener {
        @TransactionalAttribute(REQUIRED)
        public void onMessage(Message msg) {
            ...
        }
    }
}
  
```



Copyright © 2015, Oracle and/or its affiliates. All rights reserved.

Because no client provides a transaction context for calls to a message-driven bean, beans that use container-managed transactions must be deployed using the **Required** or **NotSupported** transaction attribute.

When using the **REQUIRED** Transactional Attribute and if a container rolls back an MDB transaction, message consumption is rolled back and it can be retried. In this scenario, consider using **JMSRedelivered** and **JMSXDeliveryCount** properties to avoid getting into an indefinite loop of retrying same message delivery. To achieve this create another MDB that can be mapped to the same destination to handle errors that exceed the number of retries.

```
propertyValue="JMSXDeliveryCount > 3"
```

Y, por último, los beans controlados por mensajes y, en general, los componentes JMS (productores y consumidores) pueden participar en las transacciones. Sin embargo, la transacción de un productor no es lo mismo que la transacción de un consumidor. Siempre son transacciones diferentes. Puede elegir si el consumidor consume mensajes y contexto transaccional o no. Tal vez quieras una sesión negociada. Tal vez quiera hacerlo como reconocimientos o reconocimientos automáticos. No sé, eso depende de ti. Tú decides cómo quieres que funcione.

Pero ciertamente puede decir, aquí, administración de transacciones, contenedor. Bueno, en realidad, esto es predeterminado para un Enterprise Java Bean de todos modos, por lo que es un bean controlado por mensajes, por lo que es administrado por contenedor de manera predeterminada. Y luego puede decir, atributo de transacción requerido o tal vez no compatible. Sus otros atributos transaccionales no están disponibles para beans controlados por mensajes. Solo se requiere y no se admite. Entonces, o se une a la transacción del productor JMS o simplemente consume mensajes sin contexto de transacción. Eso es todo. Es tan simple como eso.

Eche un vistazo a esta propiedad. Contador de entrega. Propiedad muy importante. Son varias veces que el bean intentó manejar el mensaje. Aquí dice menos de tres, básicamente contando intentos. Así que hay un problema con un mensaje. Inténtalo de nuevo, inténtalo de nuevo, pero no más de tres veces. Eso es lo que dice la propiedad aquí.

Handle Errors with Transactions

Transactional JMS Consumer

- It can put messages into "error queue" after exiting a number of retry attempts.
- Error Handler consumer can now take care of these messages.

```
@MessageDriven(...)
@TransactionManagement(CONTAINER)
public class ProductMessageHandler
    implements MessageListener {
    @TransactionAttribute(REQUIRED)
    public void onMessage(Message msg) {
        if (msg.getBooleanProperty("JMSRedelivered")) {
            int count = msg.getIntProperty("JMSXDeliveryCount");
            if (count > 3) {
                // move message away to "error queue"
            }
        }
    }
}
```

An Error Handler is just another JMS consumer subscribed to the "Error Queue" where you placed those messages that exceeded a number of retry attempts.

Porque si no tienes cuidado con eso, lo que podrías hacer accidentalmente es que podrías crear una especie de escenario de patata caliente. Imagina esto. Tú publicas el mensaje, ¿verdad? ESTÁ BIEN. Intentas consumirlo. El consumidor que consume el mensaje genera una excepción.

Se revierte. ¿Se requiere transacción? Está bien. Se revierte. El mensaje lo intenta de nuevo. Pero, ¿y si el problema estuviera en el mensaje? Lo intentará de nuevo indefinidamente en un bucle a menos que limites ese contador de entrega y digas, ¿sabes? Tres intentos son suficientes. Y luego publica el mensaje en otra cola en otro lugar, donde tiene otro controlador que se ocupa de los mensajes de error.

Desviaste el mensaje. Estás diciendo que no quieres volver a consumirlo repetidamente en el mismo bean, en el mismo objeto de consumo, porque sabes a dónde te llevará. Dará lugar a un error de todos modos porque el error podría estar con el mensaje en sí.

Summary

In this lesson, you should have learned how to:

- Describe Java Message Service (JMS) API messaging models
- Implement Java SE and Java EE message producers and consumers
- Use durable and shared topic consumer subscriptions
- Create message-driven beans
- Use transactions with JMS



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

Y eso es todo por esta sesión en particular. Entonces, lo que hemos cubierto son los usos de la API JMS en un mundo Java SE y en un mundo Java EE. Examinamos todo tipo de consumidores y productores. Descubrimos que esta API JMS 2.0 está muy simplificada. Básicamente, todo se controla desde el objeto JMS Context, que le permite crear mensajes, productores, consumidores y el resto.

Una parte muy importante en la que estamos tratando de descubrir una diferencia entre los tipos de consumidores-- recuerden, compartidos y no compartidos, duraderos y no duraderos, buscadores de colas-- todo tipo de permutaciones de tipos de consumidores. Uso de beans controlados por mensajes como consumidores. Manejo de transacciones JMS.

Practice Overview

This practice covers the following tasks:

- Create WebLogic JMS Server configuration
- Modify ExpiringProduct Timer EJB to produce messages
 - Post messages into the Queue
- Create Message-Driven Bean mapped to a Queue
 - Consume messages from the Queue
 - Pass messages to the Stateless EJB Facade business method



Copyright © 2016, Oracle and/or its affiliates. All rights reserved.

En un ejercicio práctico, se le pedirá que cree colas JMS en los mensajes [INAUDIBLE] del servidor WebLogic como un producto que expira. Ese es el temporizador único que creó en un ejercicio anterior. Le agregará funcionalidad. Publicará mensajes en la cola. Entonces, un producto está a punto de caducar, envíe un mensaje a la cola y luego quitemos el mensaje de la cola en otro bean, un bean controlado por mensajes, y lo pasaremos a los beans Java empresariales sin estado [INAUDIBLE] para su posterior manejo. Y básicamente la lógica habrá, me gustaría descontar el producto que está a punto de caducar. Está bien.