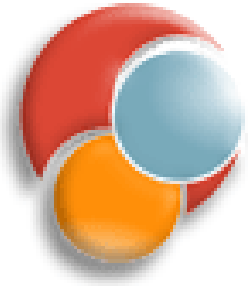


Servicios de Mensajes con JMS

Sesión 3: Transacciones JMS y JavaEE

Puntos a tratar

- Transacciones Locales
  - Ejemplos Síncrono y Asíncrono
- Transacciones Distribuidas
- Conexiones Perdidas
- JMS y JavaEE
  - Gestión de Recursos y Transacciones
  - Ejemplo de EJB y Servlet

Transacciones Locales

- Un cliente JMS puede usar transacciones locales para **agrupar bien envíos o bien recepciones en operaciones atómicas**.
- Para crear sesiones transaccionales, poner respectivamente a `true` el primer argumento del método `Connection.createSession()`.

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

- JMS no ofrece ningún método explicito de inicio de transacción.
  - Nada más crear la sesión transaccional, la transacción ha comenzado.

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

Transacciones Locales (II)

- Toda transacción forma parte de una sesión transaccional.
  - Tan pronto como se llama a commit o rollback, finaliza una transacción y comienza otra.
  - Cerrar una sesión transaccional implica un rollback automático de la transacción, incluyendo los envíos y recepciones pendientes.
- Los métodos anteriores **no pueden usarse en EJBs** ya que se usan transacciones distribuidas.

- 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*.**
- Podemos realizar:
  - varios *sends*
  - varios *receives*
  - o la recepción antes de enviar

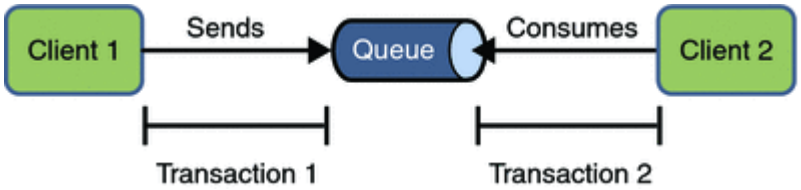
No Hacer Esto…

- No hacer transaccional *Request/Reply*
- Siempre que enviemos un mensaje y esperemos recibirlo dentro de la misma transacción el programa se colgará
- 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();
```

… Por que

- al enviar un mensaje dentro de una transacción, realmente no se envía hasta que no se realiza el *commit*.
- **La transacción no puede contener ninguna recepción que dependa de un mensaje enviado previamente**.
  - 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.
  - Debemos hacer una transacción desde el productor al recurso JMS y otra desde éste al consumidor.



- Producir y/o consumir mensajes dentro de **una sesión puede ser transaccional**
- Pero producir y consumir un mensaje específico entre **diferentes sesiones no puede ser transaccional**.

Ejemplo Síncrono

```
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);
        receiver = (QueueReceiver) session.createConsumer(queue);
        publisher = (TopicPublisher) session.createProducer(topic);

        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();
    }
}
```

Ejemplo Asíncrono (I)

```
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);
        receiver = (QueueReceiver) session.createConsumer(queue);

        listener = new TextListener(session);
        receiver.setMessageListener(listener);
        // Llamamos a start() para empezar a consumir
        connection.start();
        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();
    }
}
```

Ejemplo Asíncrono (II)

```
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) {
            }
        }
    }
}
```

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.
  - estos recursos suelen ser 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`).
- 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`.
- El gestor de transacciones de un servidor de aplicaciones utiliza los interfaces XA directamente, pero el cliente JMS solo ve las versiones no-transaccionales.

Conexiones Perdidas

- Si el proveedor JMS se cae debe intentar la reconexión. Si no lo consiguiese, debe notificar al cliente de la situación, mediante el lanzamiento de una excepción.
- ¿Problema? Un consumidor asíncrono no realiza ninguna llamada de envío o recepción **à** no llegar a detectar la pérdida de la conexión.
- JMS ofrece el interfaz `ExceptionListener` para capturar todas las conexiones perdidas y notificar a los clientes de dicha situación.

```
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.
  - El consumidor asíncrono podrá implementar este interfaz para poder actuar en esta situación, e intentar la reconexión de modo manual

Ejemplo de *ExceptionListener*

```
private 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();
            <strong>connection.setExceptionListener(this);</strong>
        } catch (JMSException ex) {
            ex.printStackTrace(System.err);
        }
    }

    @Override
    <strong>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 {
            …
            …
    }
}
```

JMS en Aplicaciones JavaEE

- Los componentes web y EJBs no deben crear más de una sesión activa (sin cerrar) por conexión.
- Cuando utilizamos la anotación `@Resource` en una aplicación cliente, la declaramos como un recurso estático:

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

- Estas declaraciones en un EJB de sesión, un MDB o un componente web **no** deben ser estáticas:

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

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

EJB Sesión para Producir/Recibir Mensajes Síncronos

- Un EJB de sesión puede producir mensajes o consumirlos de manera síncrona
  - La consumición dentro de un bloque síncrono reduce los recursos del servidor, por tanto, no es una buena práctica realizar un `receive` dentro de un EJB.
- 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, **excepto** en la **gestión de los recursos** y las **transacciones**.

Gestión de Recursos

- Es importante liberar los recursos (conexión y sesión) cuando dejan de utilizarse.
  - Si queremos mantener un recurso únicamente durante la vida de un método de negocio, lo cerramos en el bloque `finally` dentro del método.

- Si queremos mantener un recurso durante la vida de una **instancia EJB**, se recomienda 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`.

Transacciones

- En vez de usar transacciones locales, utilizamos transacciones CMT para métodos de los EJBs que realizan envíos o recepciones de mensajes
  - Permite que el contenedor EJB gestione la demarcación de las transacciones.
  - No hay 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,
  - solo si nuestros requisitos son muy complejos y dominamos muy bien los conceptos sobre transacciones.

- Normalmente, CMT produce el comportamiento más eficiente y correcto

Ejemplo de EJB

```
@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();
            }
        }
    }
}
```

Ejemplo de Aplicación Web

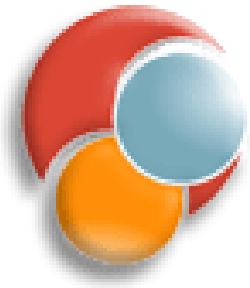
```
public class ProductorJMSServlet extends HttpServlet {
  @Resource(name = "jms/ConnectionFactory")
  private ConnectionFactory connectionFactory;
  @Resource(name = "jms/Queue")
  private Queue queue;

  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response) throws Exception {
      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 msj: " + mensaje);
      } catch (JMSException jmse) {
          jmse.printStackTrace(System.err);
      } finally {
          out.close();
      }
  }
```

```
private void enviaMensajeJMS(String msj) 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(msj);

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



¿Preguntas...?