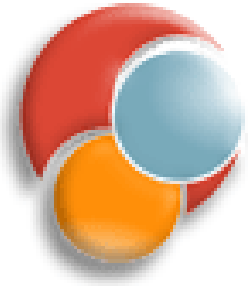


Servicios de Mensajes con JMS

Sesión 4: Message Driven Beans

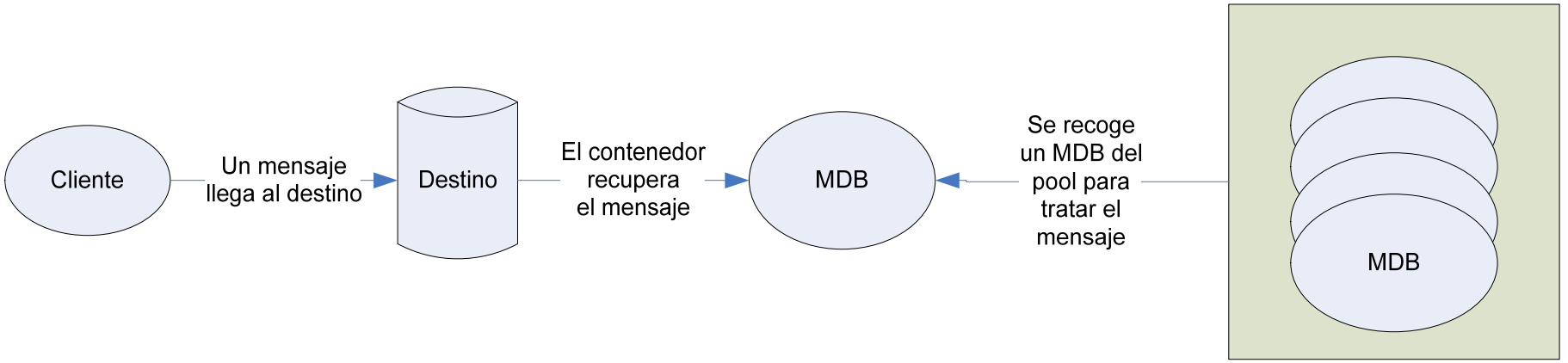
Puntos a tratar

- Introducción
- Reglas de Programación
- Anotaciones
- Callbacks
- Envío de Mensajes JMS desde el MDB
- Transacciones Distribuidas
- Mejores Prácticas
- Roadmap

Message Driven Bean

- Es un **oyente de mensajes** que puede consumir de modo **asíncrono** mensajes de una cola o de una *durable subscription*.
  - Dichos mensajes pueden ser enviados por cualquier componente JavaEE (cliente, otro EJB o una componente Web como un servlet).
  - Incluso desde una aplicación o sistema que no use tecnología JavaEE.
- Proporciona ***multi-threading*** al manejar los mensajes entrantes mediante múltiples instancias de beans alojados en el *pool* del servidor de aplicaciones.
- Todo MDB contiene el método `onMessage()`
- A diferencia de un MDB, un cliente JMS realiza las siguientes tareas:
  - Crear un consumidor asíncrono para recibir el mensaje.
    Con un MDB asociamos el destino y la factoría de conexiones durante el despliegue vía anotación/descriptor despliegue.
  - Registrar el *listener* de mensajes con `setMessageListener`
    El MDB registra el *listener* automáticamente.
  - Especificar el modo de acuse de recibo
    por defecto es `AUTO_ACKNOWLEDGE`

Proceso MDB


Cliente
Un mensaje llega al destino
Destino
El contenedor recupera el mensaje
MDB
Se recoge un MDB del pool para tratar el mensaje
MDB

Más Características…

- El MDB usa la **anotación `@MessageDriven`**
  - Permite especificar las propiedades del bean o de la factoría de conexión, tales como el tipo de destino, la subscripción duradera, el selector de mensajes, o el modo de acuse de recibo.
- El contenedor iniciará una **transacción** justo ANTES de llamar al método `onMessage()` y hará un *commit* de esta transacción cuando dicho método haga el return, a no ser que la transacción esté marcada como *rollback* en el contexto del MDB.
- Difiere de otros EJBs, en que el MDB no tiene interfaz local o remota.
  - Sólo la clase bean.
  - Se parece a un *Stateless Session Bean* (SSB); sus instancias son short-lived y no retienen estado para un cliente específico.
  - Pero sus variables pueden contener información de estado entre los diferentes mensajes de cliente: por ejemplo, un conexión a una base de datos, o una referencia a un EJB, etc...
- El contenedor tiene un **pool de objetos MDB** que permite que los mensajes se procesen concurrentemente
  - Puede afectar al orden en que se reciben los mensajes.

Reglas de Programación

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

Ejemplo de MDB

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

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

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

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

@MessageDriven

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

- Todos los argumentos son opcionales.
- Ejemplo mínimo:

```
@MessageDriven
public class GestorPeticionesCompraMDB
```

Implementando el *Listener*

- El contenedor utiliza el *listener* para registrar el MDB en el proveedor de mensajes y pasar los mensajes entrantes a los métodos implementados en el *listener*.
- Con anotaciones:

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

- Con código:

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

- Otra opción es mediante el descriptor de despliegue, y dejar los detalles fuera del código.
- La elección entre un modo u otro suele ser cuestión de gustos.

activationConfig y ActivationConfigProperty

- La propiedad `activationConfig` permite especificar la configuración específica mediante un array de instancia de `ActivationConfigProperty`
- Definición de `ActivationConfigProperty:`

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

- Cada propiedad de activación es un par (nombre, valor)
  - Nombres de propiedades: `destinationType, connectionFactoryJndiName, destinationName , acknowledgeMode, subscriptionDurability, messageSelector`

Ejemplo de ActivationConfigProperty

- Ejemplo con las propiedades más utilizadas:

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

Otras propiedades

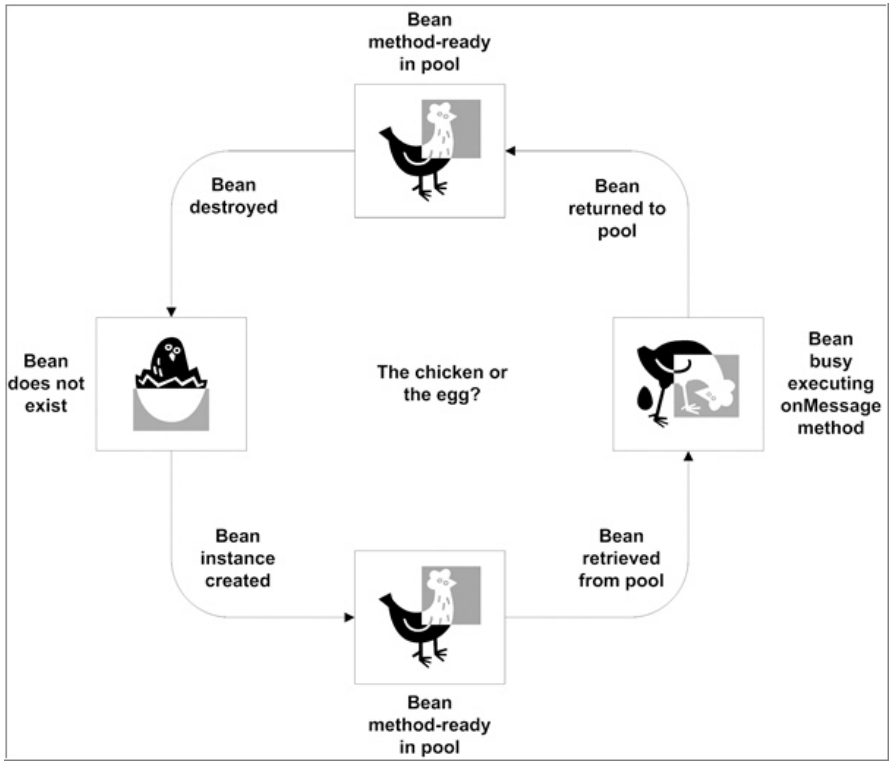
```
@ActivationConfigProperty(
    propertyName="acknowledgeMode",
    propertyValue="DUPS_OK_ACKNOWLEDGE")
```

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

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

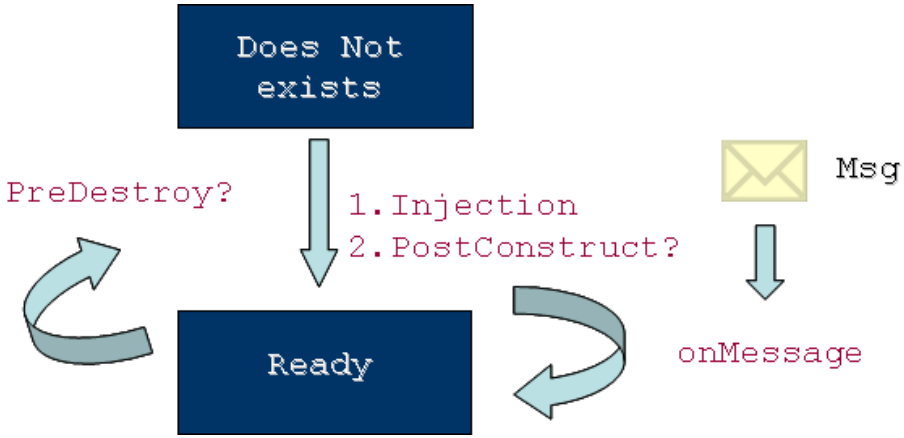
Responsabilidad del Contenedor

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



Callbacks

- PostContruct
  - Se llama inmediatamente un vez el MDB se ha creado, iniciado y se le han inyectado todo los recursos
- PreDestroy
  - Se llama antes de quitar y eliminar las instancias bean del pool.



- Estos *callbacks* se utilizan para reservar y liberar recursos inyectados que se usan dentro de `onMessage`

Ejemplo *Callbacks (I) – MDB + JDBC*

```
@MessageDriven(mappedName = "jms/Queue", activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Queue")
})
public class ConsumidorMDBJDBCBean implements MessageListener {
    private java.sql.Connection connection;
    private DataSource dataSource;
    @Resource
    private MessageDrivenContext context;

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

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

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

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

Ejemplo *Callbacks (II) – MDB + JDBC*

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

        try {
            if (message instanceof TextMessage) {
                msg = (TextMessage) message;
                System.out.println("Recibido MDB [" + msg.getText() + "]");
            } else {
                System.err.println("El mensaje no es de tipo texto");
            }
            // Accedemos a la base de datos;
            this.preguntaBBDD("Total de Libros");
        } catch (JMSException jmse) {
            jmse.printStackTrace();
            context.setRollbackOnly();
        } catch (SQLException sqle) {
            sqle.printStackTrace();
            context.setRollbackOnly();
        } catch (Throwable t) {
            System.err.println("Exception en onMessage():" + t.getMessage());
            context.setRollbackOnly();
        }
    }

    private int preguntaBBDD(String mensaje) throws SQLException {
        int result = -1;
        Statement stmt = connection.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT COUNT(*) FROM LIBRO");
        if (rs.next()) {
            result = rs.getInt(0);
            System.out.println(mensaje + " " + result);
        }
        return result;
    }
}
```

Envío de Mensajes JMS desde MDBs

- Además de los recursos de BBDD, los *callbacks* también se utilizan para gestionar los objetos administrados de JMS (los destinos y la factoría de conexiones).

La tarea que más se realiza dentro de un MDB es enviar mensajes JMS.

- Por ejemplo, cuando un MDB recibe una petición puede que algo funcione mal o que la petición sea incompleta, y por tanto, la mejor manera de notificar esto es vía JMS a una cola de error sobre la que estará escuchando el productor del mensaje.

Ejemplo *MDB* + *JMS* (I)

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

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

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

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

Ejemplo *MDB + JDBC (II)*

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

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

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

Transacciones Distribuidas

- Los clientes JMS indican en la sesión si ésta es o no transaccional.
- Con los MDBs le cedemos la decisión al contenedor
- Por defecto, el contenedor comenzará una transacción antes de iniciar el método `onMessage` y realizará el *commit* tras el *return* del método, a no ser que se marque como *rollback* mediante el contexto *message-driven*.
- Una aplicación JavaEE puede utilizar transacciones distribuidas para combinar el envío/recepción de mensajes JMS con modificaciones a una BBDD y cualquier otra operación con un gestor de recursos.
  - Las transacciones distribuidas permiten acceder a múltiples componentes de una aplicación dentro de una única transacción.
  - Por ejemplo, un Servlet puede empezar una transacción, acceder a varias BBDD, llamar a un EJB que envía un mensaje JMS y realizar un commit de la transacción.

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

Transacciones CMT y BMT dentro de un MDB

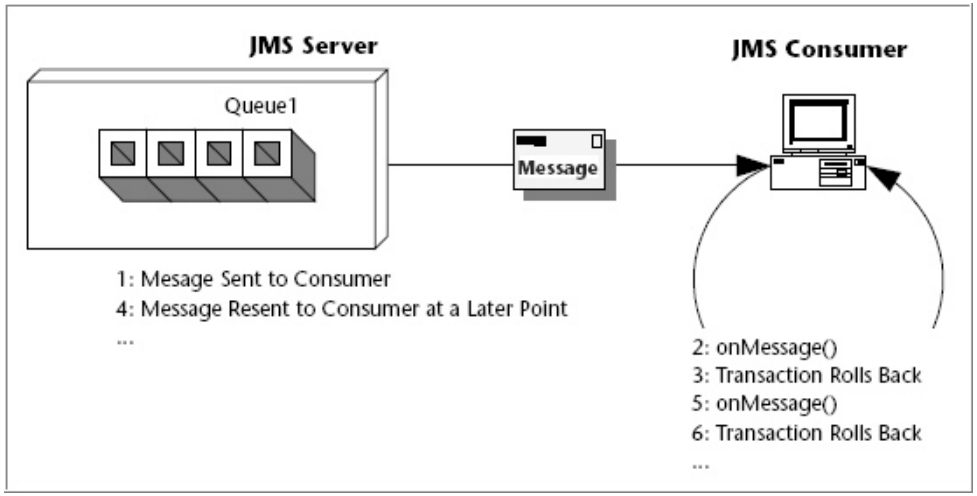
- Con los MDBs, podemos utilizar tanto transacciones CMT como BMT.
- Si un MDB no va anotado transaccionalmente, tu tipo es `Required`.
  - Esto significa que si no hay ninguna transacción en progreso, se comenzará una nueva antes de la llamada al método y se realizará el commit al finalizar el mismo.
- Con CMT podemos emplear los métodos del `MessageDrivenContext`:
  - `setRollbackOnly`: para el manejo de los errores. Marca la transacción actual para que la única salida de la transacción sea un *rollback*.
  - `getRollbackOnly`: utiliza este método para comprobar si la transacción actual está marcada para hacer *rollback*.

- Si utilizamos BMT, la entrega de un mensaje en el método `onMessage` tiene lugar fuera del contexto de la transacción distribuida.
  - La transacción comienza cuando se llama al método `UserTransaction.begin` dentro del método `onMessage`, y finaliza al llamar a `UserTransaction.commit` o `UserTransaction.rollback`.
  - Cualquier llamada al método `Connection.createSession` debe tener lugar dentro de la transacción.

Si en un BMT llamamos a `UserTransaction.rollback`, no se realiza la re-entrega del mensaje, mientras que si en una transacción CMT llamamos a `setRollbackOnly`, si que provoca la re-entrega del mensaje.

Mensajes Venenosos

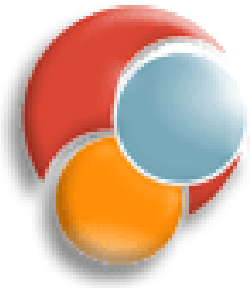
- Un mensaje venenoso es un mensaje que recibe el MDB para el cual no está preparado.
  - Si un MDBs no puede manejar uno de los mensajes recibidos (porque espera un ObjectMessage y recibe uno de texto).
  - La recepción, sin control, de este tipo de mensajes causa que la transacción en la que está inmerso el MDB haga un *rollback*.
  - Esto implica que como el MDB sigue escuchando, el mensaje venenoso se le enviará una y otra vez y entraremos en un bucle infinito.
- Solución:
  - Fijar un número de reenvíos máximo
  - Cola de *dead messages*



Mejores Prácticas

- Elige con Cuidado el Modelo de Mensajería
  - PTP resuelve el 90% de los casos
  - Intenta codificar de manera independiente al dominio
- Modulariza
  - Coloca la lógica de los MDBs fuera, en otra clase
- Bueno Uso de los Filtros de Mensajes
  - Es mejor crear dos colas que una cola con filtros
- Elige el Tipo de Mensajes con Cuidado
  - Cuidado con los mensajes XML
- Configura el Tamaño del Pool
  - Si el pool es muy pequeño, los mensajes se procesarán lentamente.
  - Si es grande, muchas instancias MDB concurrentes pueden ahogar la máquina.

- Más consejos: http://www.precisejava.com/javaperf/j2ee/JMS.htm



¿Preguntas...?