Transacciones

Índice

1. Introducción a las transacciones

En los negocios, una transacción incluye un intercambio entre dos partes. Cuando compras un periódico, intercambias dinero por un objeto; cuando trabajas para una compañía, intercambias conocimiento y tiempo por dinero. Si nos encontramos en uno de estos intercambios, siempre tenemos cuidado de asegurarnos que no sucede nada extraño. Si le damos al quiosquero un billete de 5€, esperamos que nos devuelva 4€ junto con el periódico, que vale 1€. Monitorizamos la seguridad de la transacción para asegurarnos de que cumple todas las restricciones que conlleva.

En software de negocios, una transacción incluye el concepto de un intercambio comercial. Una transacción de un sistema de negocios (transacción para abreviar) es la ejecución de una unidad-de-trabajo que accede uno o más recursos compartidos, normalmente bases de datos. Una unidad-de-trabajo es un conjunto de actividades que se relacionan mutuamente y que deben ser realizadas completamente. Por ejemplo, una operación de reserva de un billete de avión en un sistema informático puede estar formada por la selección del asiento a reservar, el cargo en una tarjeta de crédito y la generación de un billete. Todas estas acciones forman una unidad-de-trabajo que no puede romperse.

Las transacciones forman parte de distintos tipos de sistemas. En cada transacción el objetivo es el mismo: ejecutar una unidad-de-trabajo que resulte en un intercambio fiable. He aquí algunos ejemplos de sistemas de negocios que usan transacciones:

- **Cajeros automáticos**

  Los cajeros automáticos son un ejemplo típico de sistema en el que es fundamental el uso de transacciones. Cuando sacas dinero del cajero, por ejemplo, se debe chequear que tienes dinero suficiente en la cuenta corriente, después se debe entregar el dinero y por último se debe realizar un cargo en la cuenta.

- **Compra on-line**

  En una compra on-line también se debe hacer un uso intensivo de las transacciones. Cuando realizas una compra on-line debes de proporcionar el número de tarjeta de crédito, éste debe validarse y después debe cargarse el precio de la compra. Luego se debe emitir un pedido al almacén para que realice el envío de la compra. Todas estas acciones deben ser una unidad-de-trabajo, una transacción que se debe ejecutar de forma indivisible.

Los sistemas que necesitan usar transacciones normalmente son complejos y realizan operaciones que conllevan el uso grandes cantidades de datos. Las transacciones deben por ello preservar la integridad de los datos, lo que significa que todas las operaciones que

forman las transacciones deben funcionar perfectamente o la que la transacción no se debe ejecutar en absoluto.

1.1. Características de las transacciones

En el campo de la gestión de transacciones se han identificado cuatro características que las transacciones deben cumplir para que el sistema sea considerado seguro. Las transacciones deben ser atómicas, consistentes, aisladas y duraderas (ACID, en inglés). A continuación describimos estos términos:

- **Atómica**

  Para ser atómica, una transacción debe ejecutarse totalmente o no ejecutarse en absoluto. Esto significa que todas las tareas dentro de una unidad-de-trabajo deben ejecutarse sin error. Si alguna de estas tareas falla, la transacción completa se debe abortar y todos los cambios que se han realizado en los datos deben deshacerse. Si todas las tareas se ejecutan correctamente, la transacción se comete (commit), lo que significa que los cambios relizados en los datos se hacen permanentes o duraderos.

- **Consistente**

  La consistencia es una característica transaccional que debe ser impuesta tanto por el sistema transaccional como por el desarrollador de la aplicación. La consistencia se refiere a la integridad del almacén de datos. El sistema transaccional cumple su obligación de consistencia asegurando que una transacción es atómica, aislada y duradera. El desarrollador de la aplicación debe asegurarse que la base de datos tiene las restricciones apropiadas (claves primarias, integridad referencial, y otras) y que la unidad-de-trabajo, la lógica de negocio, no resulta en datos inconsistentes (esto es, los datos no se corresponden con lo que representan del mundo real). En una transferencias entre cuentas, por ejemplo, un cargo en una cuenta debe ser igual a un ingreso en otra.

- **Aislada**

  Una transacción debe poder ejecutarse sin interferencia de otros procesos o transacciones. En otras palabras, los datos a los que accede una transacción no pueden ser modificados por ninguna otra parte del sistema hasta que la transacción se completa.

- **Duradera**

  Durabilidad significa que todos los cambios en los datos realizados durante el curso de una transacción deben escribirse en algún tipo de almacenamiento físico antes de que la transacción concluya con éxito. Esto asegura que los cambios no se pierden si el sistema se cae.

1.2. Demarcación de la transacción

Las aplicaciones transaccionales deben poder comenzar y concluir transacciones, y deben poder indicar qué cambios deben hacerse permanentes o cuáles deben descartarse. Se denomina demarcación de la transacción a la indicación de los límites de la transacción dentro de un programa.

La arquitectura Java EE especifica dos tipos de demarcación de transacciones: programática y declarativa. La demarcación programática se realiza utilizando las funciones de JTA (Java Transaction Api) que marcan el comienzo -*begin*- y el final (con éxito -*commit*- o fracaso -*rollback*-) de una transacción. En la demaración declarativa, sin embargo, no se definen explícitamente el comienzo y la finalización de la transacción, sino que las transacciones están asociadas a métodos (de los componentes EJB). El comienzo de ejecución de un método define el comienzo de una transacción y su finalización normal el final con éxito. El final con fracaso de una transacción se produce cuando se sale del método por una excepción. En la demarcación declarativa es posible además definir el tipo de transacción asociada a cada método (lo veremos más adelante).

Un ejemplo de demarcación programativa de transacción usando JTA es el siguiente código:

```
try {

   userTran.begin(); //comienza la transaccion

   cliente.grabaPedido(pedido);                // operacion 1
   tarjetaCredito = cliente.tarjetaCredito();  // operacion 2
   tarjetaCredito.carga(pedido.precio);        // operacion 3
   almacen.descuenta(pedido);                  // operacion 4

   userTran.commit(); //fin de la transaccion
}
catch (Exception e} {
   userTran.rollback(); // se ha producido un error
}
```

Si no sucede ningún error entre el comienzo y el final de la transacción, la misma se realiza completamente haciendo una llamada al método commit del gestor de la transacción. Si sucede algún error y alguna de las operaciones no concluye correctamente, la transacción se aborta y se realiza la llamada al método rollback del gestor de la transacción.

2. Gestión de transacciones con JDBC

Es posible gestionar las transacciones utilizando JDBC. Para ello, el objeto

`java.sql.Connection` proporciona los siguientes métodos:

- `public void setAutoCommit(boolean)`
- `public boolean getAutoCommit()`
- `public void commit()`
- `public void rollback()`

Para gestionar manualmente transacciones hay que obtener una conexión, llamar al método `setAutoCommit(false)` para abrir la transacción y marcar el final de la misma con `commit()` o `rollback()`. El funcionamiento típico es el siguiente:

```
import java.sql.*;
import javax.sql.*;

DataSource ds = obtainDataSource();
Connection conn = ds.getConnection();
try {
   conn.setAutoCommit(false);
   // ...
   st = conn.prepareStatement("update
         operation set ffinReal=now() where idOperacion=?");
   st.setString(1, idOperacion);
   st.executeUpdate();
   // .. otras modificaciones en otras tablas
   conn.commit();
} catch (SQLException sqle) {
   try {
      conn.rollback();
   } catch (SQLException e) {
      throw new RuntimeException("Error haciendo el rollback", e);
   }
   throw new DAOException("Error al realizar la devolución", sqle);
} finally {
   try {
      if (st != null) st.close();
      if (conn != null) conn.close();
   } catch (SQLException sqlError) {
      throw new RuntimeException("Error cerrando las conexiones",
sqlError);
   }
}
```

Un problema de este enfoque es que obliga a incluir el manejo de las transacciones en los métodos de los Data Access Object (ya que fuera de ellos no podemos acceder a la conexión de la base de datos), con lo que debemos escribir código de lógica de negocio en la capa de persistencia. Esto hace que nuestra capa de datos pierda flexibilidad.

Sería interesante hacer que los DAO únicamente gestionen actualizaciones elementales y que puedan participar en transacciones declaradas fuera de ellos. La arquitectura Java EE proporciona dos formas de hacer esto: utilizando JTA (Java Transaction Api) o utilizando

componentes EJB.

3. Gestión de transacciones con JTA

Una alternativa al uso de las transacciones JDBC es JTA. Para utilizarla, es necesario un driver JDBC que implemente las interfaces `javax.sql.XADataSource`, `javax.sql.XAConnection` y `javax.sql.XAResource`. Es necesario también configurar una fuente de datos XA con el servidor de aplicaciones (ver la sesión de ejercicios) y darle un nombre JNDI.

Una vez definido el objeto `XADataSource` ya es posible trabajar con él sin cambiar nada de código en el programa, ya que funciona como una fuente de datos JDBC normal. La única diferencia es que el método `getConnection()` de la fuente de datos devuelve un objeto `Connection` que implementa la interfaz `XAConnection` y que puede participar en las transacciones JTA.

El servidor de aplicaciones también define un objeto que implementa la interfaz `javax.transacion.UserTransaction` y que debemos obtener para controlar las transacciones. Esta interfaz proporciona los siguientes métodos:

- `public void begin()`
- `public void commit()`
- `public void rollback()`
- `public int getStatus()`
- `public void setRollbackOnly()`
- `public void setTransactionTimeout(int)`

Para comenzar una transacción la aplicación se debe llamar a `begin()`. Para finalizarla, la transacción debe llamar o bien a `commit()` o bien a `rollback()`.

Un ejemplo de uso para el servidor de aplicaciones WebLogic 9.2 es el siguiente:

```
import javax.transaction.*;
import javax.naming.*;

Context ctx = getInitialContext();

UserTransaction tx = (UserTransaction)
  ctx.lookup("javax.transaction.UserTransaction");

utx.begin();

try {
   // llamadas a los DAO
   operacionDao.devuelvePrestamo(idOperacion);
```

```
   if (usuarioTO.getEstado != EstadoUsuario.reserva) {
      listOp = operacionDao.
         .selectOperacionesActivas(TipoOperacion.prestamo,
                                   usuarioTO.login);
      if (listOp != null)
         usuarioDao.cambiaEstadoUsuario(usuarioTO.login,
                                        EstadoUsuario.prestamo);
      else usuarioDao.cambiaEstadoUsuario(usuarioTO.login,
                                          EstadoUsuario.activo);
   utx.commit();
   }
} catch (Exception e) {
   try {
      utx.rollback();
   } catch (Exception e) {
      throw new RuntimeException("Error haciendo el rollback", e);
   }
   throw new BusinessException("Error al hacer la devolución", e);
}
```

La ventaja de este enfoque con respecto a la gestión de las transacciones dentro de los DAO es que libera a éstos de la gestión de transacciones y permite simplificar su código. Las transacciones son ahora algo externo a los DAO y podemos manejarlas desde los objetos de negocio.

La utilización de JTA, sin embargo, sólo es recomendable cuando se quiere tener un control programativo de las transacciones. Lo más común es definir los métodos de negocio como métodos de EJBs de sesión y dejar que sea el contenedor EJB el que se encargue de gestionar las transacciones de forma automática. Lo vemos en la siguiente sección.

4. Gestión de transacciones con componentes EJB

Los componentes EJB gestionan de forma automática las transacciones. Toda llamada a un método de un componente EJB crea un contexto transaccional que es gestionado por el servidor de aplicaciones y que dura hasta que el método ha terminado (con éxito o con fracaso, al lanzar una excepción).

Al igual que con JTA, el servidor de aplicaciones utiliza para gestionar las transacciones un sistema de *two phase commit* que permite que éstas sean distribuidas. Los recursos que participan en estas transacciones deben ser fuentes de datos XA. Los propios EJB (de sesión con estado, de entidad o de mensajes) implementan el protocolo XA y pueden participar en las transacciones distribuidas.

Un uso muy común de los EJB de sesión es la implementación de objetos de negocio que definen transacciones que abarcan a más de un DAO. De esta forma, se libera a los DAO de la gestión de transacciones y su implementación se hace más sencilla y flexible. Veremos con

detalle este uso de los EJB en la sesión de ejercicios.

En los siguientes apartados vamos a usar un ejemplo de un método de un EJB que a su vez llama a otros EJB para enfatizar el carácter distribuido de las transacciones en EJB. Sin embargo, todas las características transaccionales que comentaremos se aplican también a los EJB que manejan DAOs.

4.1. Un ejemplo con EJBs

Vamos a ver un ejemplo de una aplicación que gestiona reservas de crucero.

Suponemos un EJB de sesión con estado llamado `AgenteViaje` que contiene un método llamado `reservaPasaje` con el que se realiza la reserva de un pasaje. Suponemos que las llamadas previas a otros métodos del EJB han definido los campos `cliente`, `crucero` y `cabina`, y que lo único que tiene que hacer el método es realizar la reserva y el cargo en la tarjeta de crédito. Ambas operaciones son gestionadas por componentes EJB con interfaces remotas `Reservas` y `PagosTarjeta`.

```
public class AgenteViaje implements SessionBean {
  private Cliente cliente;
  private Crucero crucero;
  private Cabina cabina;
  ...

  public TicketTO reservaPasaje(CreditCardTO tarjeta, double precio)
    throws EstadoConversacionalIncompleto {

    if (cliente == null || crucero == null || cabina == null) {
      throw new EstadoConversacionalIncompleto();
    }
    try {
      // Se obtienen los home
      ReservasHome resHome = getEJB(jndiContext,
"java:comp/env/ejb/ReservasHome",
                                    ReservasHome.class);
      PagosTarjetaHome ppHome = getEJB(jndiContext,
"java:comp/env/ejb/PagosTarjetaHome",
                                     PagosTarjetaHome.class);

      // Se crean los ejb
      Reservas reservas =  resHome.create();
      PagosTarjeta pagos = ppHome.create();

      // Se llaman a los métodos de la interfaz de negocio

      reservas.hazReserva(customer,cruise, cabin, price);
      pagos.procesaPago(customer, card, price);
      TicketTO ticket = new TicketTO(customer,cruise,cabin,price);
      return ticket;
```

```
    } catch(Exception e) {
        throw new EJBException(e);
    }
  }
...
}
```

Si se genera alguna excepción en alguna de las llamadas del método `reservaPasaje`, se captura y se lanza a su vez una excepción `EJBException`. Esta es una excepción de tipo `RuntimeException` y veremos que es la responsable de que la transacción haga un *rollback*.

Una primera medida de la fiabilidad del EJB `AgenteViaje` es su atomicidad: ¿asegura que la transacción se ejecuta completamente o no se ejecuta en absoluto? Para responder debemos concentrarnos en las tareas críticas que modifican o crea información nueva. En el método `reservaPasaje()`, se crea un EJB `Reservas`, el EJB `PagosTarjeta` realiza un cargo en una tarjeta de crédito, y se crea un objeto `TicketTO`. Todas estas tareas deben tener éxito para que la transacción completa lo tenga a su vez.

Para entender la importancia de la característica de atomicidad podríamos imaginar qué sucedería si cualquiera de estas tareas fallara. Si, por ejemplo, la creación de un EJB `Reservas` fallara pero todas las otras tareas tuvieran éxito, el cliente terminaría probablemente expulsado del crucero o compartiendo el camarote con un extraño. En lo que concierne a la agencia de viajes, el método `reservaPasaje()` habría sido ejecutado con éxito porque se habría generado un `TicketTO`. Si se genera un billete sin la creación de una reserva, el estado del sistema de negocio se convierte en inconsistente con la realidad, porque el cliente ha pagado por un billete pero la reserva no se ha registrado. De la misma forma, si el EJB `PagosTarjeta` falla al cargar la tarjeta de crédito del cliente, el cliente obtiene un crucero gratis. Seguro que él se quedará contento, pero gerencia no. Por último, si el `TicketTO` no se crea nunca, el cliente no tendrá ningún registro de la transacción y probablemente no podrá subir al crucero.

Por ello, la única forma de que pueda completarse la operación `reservaPasaje()` es si todas sus partes críticas se ejecutan con éxito. Si algo va mal, el proceso completo debe abortarse. Abortar una transacción requiere más que simplemente no finalizar las tareas; además, todas las tareas dentro de la transacción deben deshacerse. Si, por ejemplo, la creación de los EJB `Reservas` y `PagosTarjeta` se realiza con éxito, pero la creación del `TicketTO` falla, los registros `reservation` y `payment` no deben añadirse a la base de datos.

Para que la operación `reservaPasaje()` sea completamente segura debe cumplir los otros requisitos de una transacción: debe ser consistente, aislada y duradera.

Para mantener la consistencia de las operaciones desde el punto de vista de la lógica del negocio, es necesario que se cumplan las otras tres propiedades y además que el desarrollador de la aplicación sea estricto a la hora de aplicar las restricciones de integridad en toda la implementación de la aplicación. Por ejemplo, de nada serviría que el sistema transaccional asegurase la atomicidad de la operación `reservaPasaje()` si el desarrollador no incluyera dentro del método una llamada a la consulta de la tarjeta de crédito y devolviera directamente el TicketTO. Desde el punto de vista del negocio, la transacción habría fallado, ya que se ha emitido un ticket sin realizar un cobro.

El aislamiento de la operación tiene que asegurar que otros procesos no van a modificar los datos de los EJBs mientras que la transacción está desarrollándose.

Por último, la durabilidad de la transacción obliga a que todas las operaciones hayan sido hechas persistentes antes de dar la transacción por terminada.

Asegurarnos que las transacciones se adhieren a los principios ACID requiere un diseño cuidadoso. El sistema tiene que monitorizar el progreso de una transacción, para asegurarse de que todo funciona correctamente, de que los datos se modifican de forma correcta, de que las transacciones no interfieren entre ellas y de que los cambios pueden sobrevivir a una caída del sistema. Comprobar todas estas condiciones conlleva un montón de trabajo. Afortunadamente, la arquitectura EJB soporta de forma automática el manejo de transacciones.

4.2. Alcance de la transacción

El alcance de una transacción es un concepto crucial para comprender las transacciones. En este contexto, el alcance una transacción consiste en aquellos EJBs de entidad y de sesión que están participando en una transacción particular.

En el método `reservaPasaje()` del EJB `AgenteViaje`, todos los EJBs que participan son parte del mismo alcance de transacción. El alcance de la transacción comienza cuando el cliente invoca el método `reservaPasaje()` del EJB `AgenteViaje`. Una vez que el alcance de la transacción ha comenzado, éste se propaga a los dos EJB que se crean: `Reservas` EJB y `PagosTarjeta` EJB.

Como ya hemos comentado, una transacción es una unidad-de-trabajo constituida por una o más tareas. En una transacción, todas las tareas que forman la unidad-de-trabajo deben ser un éxito para que la transacción en su totalidad tenga éxito; la transacción debe ser atómica. Si alguna tarea falla, las actualizaciones realizadas por las otras tareas deben desacerse. En EJB, las tareas se expresan como métodos de los enterprise bean, y una unidad de trabajo consiste en un conjunto de invócaciones a métodos los enterprise bean. El alcance de la transacción incluye todos los EJB que participan en al unidad de trabajo.

Es fácil trazar el alcance de una transacción siguiendo el hilo de ejecución. Si la invocación del método `reservaPasaje()` comienza una transacción, entoces de forma lógica, la transacción termina cuando el método se completa. El alcance de la transacción `reservaPasaje()` incluiría los EJB `AgenteViaje`, `Reservas` y `PagosTarjeta`.

Una transacción puede terminar si se genera una excepción mientras que el método `reservaPasaje()` está en ejecución. La excepción puede lanzarse desde uno de los otros EJBs o desde el mismo método `reservaPasaje()`. Para que el contenedor de EJB active la gestión de transacciones, se debe lanzar una excepción de tipo `RuntimeException`. En el caso del ejemplo anterior, se lanza la excepción `EJBException`, que es de tipo *runtime*.

La gestión de la transacción puede causar o no causar rollback (vuelta a los valores iniciales de los datos) dependiendo del tipo de transacción declarado. Lo veremos más adelante.

4.3. Gestión declarativa de las transacciones

Una de las ventajas principales de la arquitectura Enterprise JavaBeans es que permite la gestión declarativa de transacciones. Sin esta característica las transacciones deberían controlarse usando una demarcación explícita de la transacción. Esto conlleva el uso de APIs bastantes complejos como el Objecto Transaction Service (OTS) de OMG, o su implementación Java, el Java Transaction Service (JTS) que soporta la ya vista JTA. La demarcación explícita es difícil de usar correctamente para los desarrolladores, sobre todo si no están habituados a la programación de sistemas transaccionales.

Además, la demarcación explícita requiere que el código transaccional se escriba junto con la lógica de negocio, lo que reduce la claridad del código y, más importante, crea objetos distribuidos inflexibles. Una vez que la demarcación de la transacción está grabada en el objeto de negocio, los cambios en la conducta de transacción obligan a cambios en la misma lógica de negocio.

Con la gestión declarativa de transacciones, la conducta transaccional de los EJBs puede controlarse usando el descriptor del despliegue, que establece atributos de transacción para los métodos individuales del enterprise bean. Esto significa que la conducta transaccional de un EJB puede modificarse sin cambiar la lógica de negocio del EJB. Además, un EJB desplegado en una apliación puede definirse con una conducta transaccional distinta que la del mismo bean desplegado en otra aplicación. La gestión declarativa de las transacciones reduce la complejidad del manejo de las transacciones para los desarrolladores de EJB y de aplicaciones y hace más sencilla la creación de aplicaciones transaccionales robustas.

Cuando un EJB se despliega, podemos establecer su atributo de transacción a uno de los siguientes valores:

- `NotSupported`
- `Supports`
- `Required`
- `RequiresNew`
- `Mandatory`
- `Never`

La especificación EJB 2.0 aconseja que los beans de entidad con persistencia gestionada por el contenedor usen sólo los atributos `Required`, `RequiresNew` y `Mandatory`. Esta restricción asegura que todos los accesos a bases de datos suceden en el contexto de una transacción, lo cual es importante cuando el contenedor está gestionando automáticamente la persistencia.

Podemos establecer un atributo de transacción para el EJB completo (en cuyo caso se aplica a todos los métodos) o establecer distintos atributos de transacción para los métodos individuales. Lo primero es mucho más sencillo y conlleva menos riesgo de errores, pero lo segundo ofrece mayor flexibilidad. Los fragmentos de código de los siguientes apartados muestran cómo establecer los atributos de transacciones de un EJB en el descriptor del despliegue del EJB.

4.3.1. Descriptor del despliegue

En el descriptor XML del despliegue, un elemento `container-transaction` especifica los atributos de transacción para los EJBs descritos en el descriptor de despliegue:

```
<ejb-jar>
    ...
    <assembly-descriptor>
        ...
        <container-transaction>
            <method>
                <ejb-name>AgenteViajeEJB</ejb-name>
                <method-name> * </method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        </container-transaction>
        <container-transaction>
            <method>
                <ejb-name>AgenteViajeEJB</ejb-name>
                <method-name>listaCamarotesDisponibles</method-name>
            </method>
            <trans-attribute>Supports</trans-attribute>
        </container-transaction>
        ...
    </assembly-descriptor>
    ...
```

```
</ejb-jar>
```

Este descriptor de despliegue especifica los atributos de transacción para el EJB `AgenteViaje`. Cada elemento `container-transaction` especifica un método y el atributo de transacción que debería aplicarse a ese método. El primer elemento `container-transaction` especifica que todos los métodos tengan por defecto un atributo de transacción de `Required`; el carácter `*` es un comodín que indica todos los métodos del EJB `AgenteViaje`. El segundo elemento `container-transaction`hace caso omiso del atributo por defecto para especificar que el método `listaCamarotesDisponibles()` tendrá un atributo de transacción `Supports`. Tenemos que especificar a qué EJB nos estamos refiriendo con el elemento `ejb-name`, ya que un descriptor XML de despliegue puede contener muchos EJBs.

4.3.2. Definición de los atributos de transacción

Vamos a definir ahora los atributos de transacción que hemos declarado previamente. Tal y como comentamos arriba, los atributos recomendados por la especificación EJB 2.0 son `Required`, `RequiresNew` y `Mandatory`

- **`Required`**

  Este atributo significa que el método del enterprise bean debe invocarse dentro del alcance de una transacción. Si el cliente o el EJB que realiza la llamada es parte de una transacción el EJB con el atributo `Required` automáticamente se incluye en el alcance de esa transacción. Si, sin embargo, el cliente o EJB que realiza la llamada no está incluído en una transacción, el EJB con el atributo `Required` comienza su propia transacción. Esta nueva transacción concluye cuando el EJB termina.

- **`RequiresNew`**

  Este atributo significa que se debe comenzar siempre una nueva transacción. Independiéntemente de si el cliente o EJB que realiza la llamada es parte de una transacción, un método con el atributo RequiresNew siempre comienza una nueva transacción. Si el cliente que realiza la llamada ya está incluido en una transacción, esa transacción se suspende hasta que la llamada al método con el atributo RequiresNew finaliza. En ese momento la transacción original vuelve a estar activa.

- **`Mandatory`**

  Este atributo significa que el método del enterprise bean debe siempre ser parte del alcance de la transacción del cliente que realiza la llamada. Si este cliente o EJB no es parte de una transacción, la invocación fallará, generándose una excepción `javax.transaction.TransactionRequiredException` a los clientes remotos o una excepción

`javax.ejb.TransactionRequiredLocalException` a los clientes locales.

4.4. Propagación de la transacción

Para ilustrar el impacto de los atributos de transacción sobre los métodos del enterprise bean, miraremos una vez más al método `reservaPasaje()` del EJB `AgenteViaje`.

Para que `reservaPasaje()` se ejecute como una transacción con éxito, tanto la creación del EJB `Reservas` como el cargo a la tarjeta de crédito deben también terminar con éxito. Esto significa que ambas operaciones deben incluirse en la misma transacción. Si alguna operación falla, la transacción completa falla. Podríamos haber especificado un atributo de transacción `Required` como el atributo por defecto para todos los EJBs incluidos, porque ese atributo refuerza la política deseada de que todos los EJBs deben ejecutarse dentro de una transacción y por ello asegura la consistencia de los datos.

Como un monitor de transacciones, el servidor EJB vigila cada llamada a un método en la transacción. Si cualquiera de las actualizaciones falla, todas las actualizaciones a todos los EJBs serán recuperadas (*rolled back*). Si las actualizaciones se realizan, la transacción se confirma (*commit*).

En los casos en los que el contenedor gestiona implicitamente la transacción, las decisiones de commit y rollback se manejan de forma automática. Supongamos que el EJB `AgenteViaje` se crea y se usa en el cliente como sigue:

```
AgenteViaje agente = agenteHome.create(cliente);
agente.setCamaroteID(camarote_id);
agente.setCruceroID(crucero_id);
try {
    agent.reservaPasaje(card,price);
} catch(Exception e) {
    System.out.println("Transaction failed!");
}
```

Más aún, asumamos que el método `reservaPasaje()` tienen un atributo de transacción `RequiresNew`. En este caso, el cliente que invocal el método `reservaPasaje()` no es parte de la transacción. Cuando se invoca a `reservaPasaje()` en el EJB `AgenteViaje`, se crea una nueva transacción, tal y como dicta el atributo `RequiresNew`. Esto significa que el EJB `AgenteViaje` se registra en el gestor de transacciones del servidor de EJB, el cual gestionará la transacción automáticamente. El gestor de transacciones coordina transacciones, propagando el alcance de la transacción desde un EJB al siguiente para asegurarse de que todos los EJBs tocados por una transacción se incluyen en la unidad de trabajo de la transacción. De esta forma, el gestor de transacciones puede monitorizar las actualizaciones realizadas por cada enterprise bean y decidir, basándose en el

éxito de estas actualizaciones, si hacer permanentes los cambios hechos por todos los enterprise beans en las bases de datos o si echarlos atrás y deshacerlos. Si se lanza una excepción del sistema en el método `reservaPasaje()`, la transacción se deshace automáticamente.

Cuando el método `procesaPago()` se invoca dentro del método `reservaPasaje()`, el EJB `PagosTarjeta` se registra en el gestor de transacciones bajo el contexto transaccional que se creó para el EJB `AgenteViaje`. Cuando se crea el nuevo EJB `Reservas`, también se registra en el gestor de transacciones bajo la misma transacción. Cuando se registran todos los EJBs y se realizan todas las actualizaciones, el gestor de transacciones chequea todo para asegurarse de que sus actualizaciones funcionarán. Si uno de los EJB devuelve un error o falla, los cambios realizados por los EJB `PagosTarjeta` o `Reservas` se deshacen por el gestionador de transacciones.

Además de gestionar las transacciones en su mismo entorno, un servidor EJB puede cooordinarse con otros sistemas transaccionales. Si, por ejemplo, el EJB `PagosTarjeta` viniera de un servidor EJB distinto, los dos servidores EJB cooperarían para gestionar la transacción como una unidad-de-trabajo. Esto se llama una transacción distribuída.

Una transacción distribuida es mucho más complicada, y requiere lo que se llama two-phase commit (2PC). 2PC es un mecanismo que permite que una transacción sea gestionada a través de distintos servidores y recursos (por ejemplo, bases de datos y proporcionadores JMS). Los detalles de un sistema 2PC están más allá del alcance de este tema. No habrá que hacer ninguna modificación en el código para que nuestros beans apliquen transacciones distribuidas. Es un trabajo trasparente al programador de EJB.

5. Anotaciones EJBGen Weblogic para transacciones

En Weblogic 9.2 se utiliza el parámetro `transactionAtribute` en las anotaciones de los métodos `LocalMethod` y `RemoteMethod` para indicar que el método es transaccional.

Como valor del atributo se puede indicar:

- `Constants.TransactionAttribute.UNSPECIFIED`
- `Constants.TransactionAttribute.NOT_SUPPORTED`
- `Constants.TransactionAttribute.SUPPORTS`
- `Constants.TransactionAttribute.REQUIRED`
- `Constants.TransactionAttribute.REQUIRES_NEW`
- `Constants.TransactionAttribute.MANDATORY`
- `Constants.TransactionAttribute.NEVER`

Estas constantes corresponden con las distintas posibilidades de gestionar las transacciones vistas anteriormente.

También es posible declarar un tipo de gestión de transacciones por defecto para todos los métodos de un EJB, declarando uno de los valores anteriores en el atributo `defaultTransaction` de la anotación `Session` del EJB.