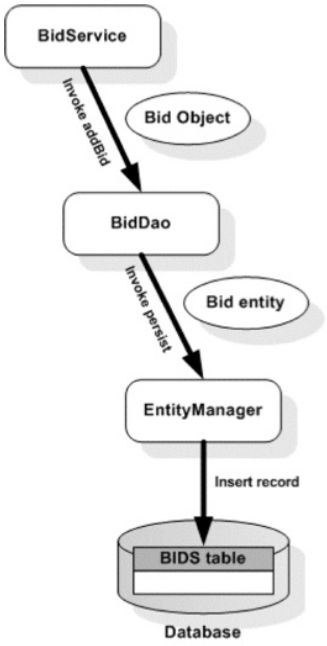
JPA, transacciones, temporizadores e interceptores

Índice

1. JPA y componentes EJB

Ya conocemos JPA, el estándar de Java EE para gestionar la persistencia de una aplicación. JPA define un ORM (*Object-Relational Mapping*) que permite modelar las clases persistentes de la aplicación utilizando clases Java (*entidades*) que son mapeadas con un esquema SQL del servidor de base de datos relacional.

La forma habitual de trabjar con JPA y EJB es que los beans de sesión definan una interfaz de negocio transaccional, que puede ser remota y/o local. Es posible agrupar distintas llamadas a métodos de negocio de distintos beans porque el contenedor gestiona la transaccionalidad mediante JTA.



Para implementar la capa de negocio los beans utilizan unos objetos DAO que encapsulan el acceso a las entidades y se obtienen por inyección de dependencias. Las clases DAO obtienen el *entity manager* también por inyección de dependencias. El contenedor se encarga de gestionar esa inyección y se asegura de que todos los objetos DAO que están una misma transacción van a trabajar con el mismo *entity manager* y, por tanto, van a compartir también el *contexto de persistencia*.

Repasemos los conceptos fundamentales.

1.1. Entidades

Las entidades son clases Java con la anotación `@Entity` que definen el mapeo con las tablas del esquema de base de datos.

Como ejemplo proporcionamos la definición de las entidades `Autor` y `Mensaje` que definen las clases de dominio de una posible aplicación que mantiene un foro de mensajes.

`Autor.java`

```
@Entity
@NamedQueries({
           @NamedQuery(name="Autor.findAll",
                       query="SELECT a FROM Autor a")
        })
public class Autor implements Serializable {
   @Id
   @GeneratedValue
   Long id;
   private String correo;
   private String nombre;
   @OneToMany(mappedBy = "autor", cascade = CascadeType.ALL)
   private List<Mensaje> mensajes = new ArrayList<Mensaje>();

   public Autor() {
   }


   //...
}
```

`Mensaje.java`:

```
@Entity
@NamedQueries({
    @NamedQuery(name = "Mensaje.numMensajesAutor",
    query = "SELECT count(m) "
    + "FROM Mensaje m "
    + "WHERE m.autor.id = :id"),
    @NamedQuery(name = "Mensaje.mensajesAutor",
    query = "SELECT m "
    + "FROM Mensaje m "
    + "WHERE m.autor.id = :id")
})
public class Mensaje implements Serializable {
    @Id
    @GeneratedValue
    @Column(name = "mensaje_id")
    private Long id;
    private String texto;
    @Temporal(javax.persistence.TemporalType.DATE)
    private Date fecha;
    @ManyToOne
    private Autor autor;

    public Mensaje() {
    }

   //...
}
```

Algunas cosas que hacer notar en el código:

- La clase debe implementar `Serializable` y definir un método constructor público sin argumentos.
- La anotación `@Table` define explícitamente el nombre de la tabla con la que se mapea la entidad. Si no existe, se toma el nombre de la clase.
- La clave primaria se define con la anotación `@Id`.
- Los atributos de la entidad se mapean con columnas de la tabla del esquema.
- Las relaciones uno-a-uno y uno-a-muchos se mapean con las anotaciones `@OneToMany` y `@ManyToOne`. El atributo `mappedBy` define la columna de la entidad *propietaria* de la relación, la que contiene la clave ajena en el modelo relacional.
- Las entidades contienen también las consultas JPQL definidas con un nombre (*named query*).

1.2. Unidad de persistencia

La *unidad de persistencia* se define en el fichero de configuración `persistence.xml` y especifica la fuente de datos con la que JPA debe conectarse para realizar el mapeo de las entidades en tablas.

El fichero `persistence.xml` puede formar parte de un fichero WAR o un fichero EJB-JAR, o puede ser empaquetado junto con las clases en un fichero JAR que puede luego incluirse en un fichero WAR o EAR. En todos esos casos debe estar dentro de un directorio `META-INF`.

- Si se empaqueta la unidad de persistencia en un fichero EJB-JAR, el `persistence.xml` debería colocarse en el directorio `META-INF` del EJB-JAR.
- Si se empaqueta la unidad de persistencia en un fichero WAR, el `persistence.xml` debería colocarse en el directorio `WEB-INF/classes/META-INF` del WAR.
- Si se empaqueta la unidad de persistencia en un fichero JAR que se incluye en un fichero WAR o EAR, el fichero JAR debería estar situado en el directorio `WEB-INF/lib` del WAR o en el directorio de bibliotecas del EAR.

Por ejemplo, este el fichero `persistence.xml` definido para el proveedor de persistencia `EclipseLink`

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="
 http://java.sun.com/xml/ns/persistence
 http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="mensajes-ejbPU" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>jdbc/ejb</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
      <property name="eclipselink.ddl-generation" value="create-tables"/>
    </properties>
```

```
  </persistence-unit>
</persistence>
```

Resaltamos algunos puntos:

- Se define como nombre de la unidad de persistencia `mensajes-ejbPU`.
- Se debe utilizar JTA como forma de gestionar las transacciones cuando JPA va a estar gestionado por el contenedor.
- Se utiliza la fuente de datos `jdbc/ejb` que debe estar definida en el servidor de aplicaciones.
- Con la propiedad `exclude-unlisted-classes` puesta a `false` no hace falta listar explícitamente las clases de entidad que se van a gestionar, se incluyen automáticamente todas las anotadas con `entity`.
- El valor `create-tables` crea las tablas mapeadas si no existen.

1.3. Entity manager y contexto de persistencia

Todas las acciones efectivas sobre las base de datos se realizan a través del *entity manager*. Es el objeto que se encarga de realizar las consultas y actualizaciones en las tablas y de mantener sincronizados con la base de datos los objetos entidad que residen en su contexto de persistencia.

Recordemos que el contexto de persistencia de un *entity manager* está formado por todos los objetos entidad gestionadas por el *entity manager* y que representan una caché de primer nivel de los datos.

El *entity manager* se encarga de propagar a la base de datos los cambios realizados en los objetos entidad haciendo un *flush* de forma automática cuando la transacción termina o antes de ejecutar una query.

En JPA gestionado por el contenedor la forma normal de obtener el *entity manager* es usando la inyección de dependencias:

```
@PersistenceContext(unitName = "mensajes-ejbPU")
EntityManager em;
```

El atributo `unitName` indica la unidad de persistencia con la que se conecta el *entity manager*.

Esta inyección de dependencias hay que hacerla en un objeto gestionado al que el contenedor tenga acceso. Lo habitual es hacerla en el bean de sesión que define la interfaz de negocio o, lo que vamos a hacer nosotros, en los propios objetos DAO inyectados en los beans de sesión:

`TablonService.java`:

```
@Stateless
public class TablonService {

    @Inject
```

```
    MensajeDao mensajeDao;
    @Inject
    AutorDao autorDao;

    @Override
    public String getMensaje(Long id) {
        Mensaje mensaje = mensajeDao.find(id);
        return mensaje.getTexto();
    }

    @Override
    public String getAutor(Long id) {
        Autor autor = autorDao.find(id);
        return autor.getNombre();
    }

    //...
```

En los DAO se declara la inyección del *entity manager*:

`Dao.java`:

```
abstract class Dao<T, K> {
   @PersistenceContext(unitName = "mensajes-ejbPU")
   EntityManager em;

   public abstract T find(K id);

   public T create(T t) {
      em.persist(t);
      em.flush();
      em.refresh(t);
      return t;
   }

   public T update(T t) {
      return (T) em.merge(t);
   }

   public void delete(T t) {
      t = em.merge(t);
      em.remove(t);
   }
}
```

`AutorDao.java`:

```
public class AutorDao extends Dao<Autor, Long> {

    @Override
    public Autor find(Long id) {
        return em.find(Autor.class, id);
    }

    @SuppressWarnings("unchecked")
    public List<Autor> findAll() {
        return em.createNamedQuery("Autor.findAll").getResultList();
    }
}
```

`MensajeDao.java`:

```
public class MensajeDao extends Dao<Mensaje, Long> {

    @Override
    public Mensaje find(Long id) {
        return em.find(Mensaje.class, id);
    }

    @SuppressWarnings("unchecked")
    public List<Mensaje> findMensajesAutor(Long id) {
        return em.createNamedQuery("Mensaje.mensajesAutor")
                .setParameter("id", id).getResultList();
    }
}
```

El contenedor es el encargado de inyectar el *entity manager* en los DAO. El contenedor decidirá si debe crear un entity manager nuevo o si debe utilizar uno ya existente dependiendo del atributo `type` de la anotación `@PersistenceContext`.

El valor por defecto del atributo `type` es `PersistenceContextType.TRANSACTION`:

```
@PersistenceContext(type=PersistenceContextType.TRANSACTION)
EntityManager em;
```

En los entity managers de tipo `TRANSACTION` se inyecta el mismo *entity manager* en todos los DAO que comparten la transacción. Esto significa que los distintos DAOs estarán compartiendo también el mismo contexto de persistencia y aprovechando la caché de entidades gestionadas.

Por ejemplo, todas las llamadas a DAOs en siguiente método de negocio `intercambiaMensajes` utilizarán el mismo *entity manager*:

```
@Stateless
public class TablonService {

   @Inject
   AutorDao autorDao;
   @Inject
   MensajeDao mensajeDao;

   ...

   public void intercambiaMensajes(Long idAutorOrigen,
                                   Long idAutorDestino) {
      Autor autorOrigen = autorDao.find(idAutorOrigen);
      Autor autorDestino = autorDao.find(idAutorDestino);
      List<Mensaje> mensajesAutorOrigen = autorOrigen.getMensajes();
      List<Mensaje> mensajesAutorDestino = autorDestino.getMensajes();

      // cambiamos los autores de los mensajes

      for (Mensaje mensaje : mensajesAutorOrigen) {
         mensaje.setAutor(autorDestino);
         mensajeDao.update(mensaje);
      }
      for (Mensaje mensaje : mensajesAutorDestino) {
         mensaje.setAutor(autorOrigen);
         mensajeDao.update(mensaje);
      }

      // intercambiamos las colecciones en memoria
```

```
        autorDestino.setMensajes(mensajesAutorOrigen);
        autorOrigen.setMensajes(mensajesAutorDestino);
    }
}
```

El otro tipo de gestión de la vida del entity manager se utiliza con los beans de sesión con estado. El contexto de persistencia se mantiene abierto mientras que existe el bean con estado y las entidades permanecen gestionadas a lo largo de múltiples transacciones. Es lo que se denomina contexto de persistencia extendido:

```
@PersistenceContext(type=PersistenceContextType.EXTENDED)
EntityManager em;
```

2. Transacciones

2.1. Gestión de transacciones con JTA

JTA (*Java Transaction API*) es el estándar propuesto por Java EE para gestionar transacciones distribuidas. Es posible utilizar JTA en componentes gestionados por el servidor de aplicaciones (servlets, enterprise beans y clientes enterprise).

JTA se incluye en el perfil Web de Java EE 6, por lo que podemos utilizarlo para gestionar las transacciones de componentes EJB lite.

A mediados de los 80 comenzaron los esfuerzos de estandarización de APIs y protocolos relacionados con transacciones. La estandarización de APIs permitiría que la portabilidad de los gestores de recursos (bases de datos y otros) entre distintos monitores de transacciones. Como resultado de estos esfuerzos se desarrollaron, entre otros, el modelo de procesamiento distribuido de transacciones (modelo DTP) de X/Open y la especificación Object Transaction Service (OTS) que, basándose en el primero, define el manejo de transacciones en CORBA. Ambos modelos han tenido una importante influencia en los modelos de procesamiento de transacciones de Java JTA.

En el modelo DTP de X/Open se definen tres entidades:

- Gestores de recursos (resource managers): Controlan los recursos (bases de datos, colas de mensajes, etc.) e implementan la interfaz XA. Esta interfaz es usada por el gestor de transacciones alistar gestores de recursos y para realizar el protocolo two-phase commit. Tiene operaciones del estilo de start, end, commit o rollback.
- Gestor de transacciones (transaction manager): Se encarga de alistar y eliminar gestores de recursos y ejecutar entre todos ellos el protocolo 2PC cuando se realiza una transacción. Los gestores de transacciones deben implementar la interfaz TX para que el programa de aplicación y otros gestores de transacciones se comuniquen con ellos. El programa de aplicación le solicita operaciones como begin, commit o rollback.
- Programa de aplicación: se comunica con el gestor de transacción usando la interfaz

TX para delimitar el comienzo y final de la transacción. Se comunica con el gestor de recursos para realizar cambios en los datos usando JDBC o cualquier otra API que implemente el gestor de recursos.

La especificación JTA define una arquitectura para la construcción de servidores de aplicaciones transaccionales y define un conjunto de interfaces para los componentes de esta arquitectura. Los componentes son los mismos que se define en el modelo DTP de X/Open: la aplicación, los gestores de recursos, el gestor de transacciones y el gestor de comunicaciones. De esta forma se define una arquitectura que posibilita la definición de transacciones distribuidas utilizando el algoritmo *two-phase-commit*.

Un requisito fundamental para que un recurso pueda incorporarse a las transacciones JTA es que las conexiones con el recurso sean de tipo `XAConnection`. Para ello, en el caso de una base de datos JDBC, es necesario un driver JDBC que implemente las interfaces `javax.sql.XADataSource`, `javax.sql.XAConnection` y `javax.sql.XAResource`.

Es necesario también configurar en el servidor de aplicaciones la fuente de datos indicando que el recurso va a utilizar JTA. De esta forma el método `getConnection()` de la fuente de datos devuelve un objeto `Connection` que implementa la interfaz `XAConnection` que puede participar en las transacciones JTA.

2.1.1. Gestión de la transacción con la interfaz UserTransaction

Para gestionar una transacción JTA de forma programativa deberemos en primer lugar obtener del servidor de aplicaciones el objeto `javax.transacion.UserTransaction`. La forma más sencilla de hacerlo es mediante inyección de dependencias, declarándolo con la anotación:

```
@Resource
UserTransaction utx;
```

Una vez obtenido el `UserTransaction` podemos llamar a los métodos de la interfaz para demarcar la transacción:

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

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()`.

El método `setRollbackOnly()` modifica la transacción actual de forma que el único resultado posible de la misma sea de roll back (falllo).

El método `setTransactionTimeout(int segundos)` permite modificar el timeout

asociado con la transacción actual. El parámetro determina la duración del timeout en segundos. Si se le pasa como valor cero, el timeout de la transacción es el definido por defecto.

El siguiente ejemplo simplificado muestra un fragmento de código que utiliza transacciones JTA. En el ejemplo mostramos un fragmento de código en el que un usuario de una biblioteca devuelve tarde un préstamo y se le anota una multa

```
UserTransaction tx;

//...
// Obtenemos el UserTransaction
// y lo guardamos en tx
//...

tx.begin();
try {
   operacionService.devolverEjemplar(ejemplar);
   usuarioService.multar(login, fechaDevolucionPrevista,
                         fechaDevolucionReal);
   tx.commit();
   }
} catch (Exception e) {
   tx.rollback();
   throw new BusinessException("Error al hacer la devolución", e);
}
```

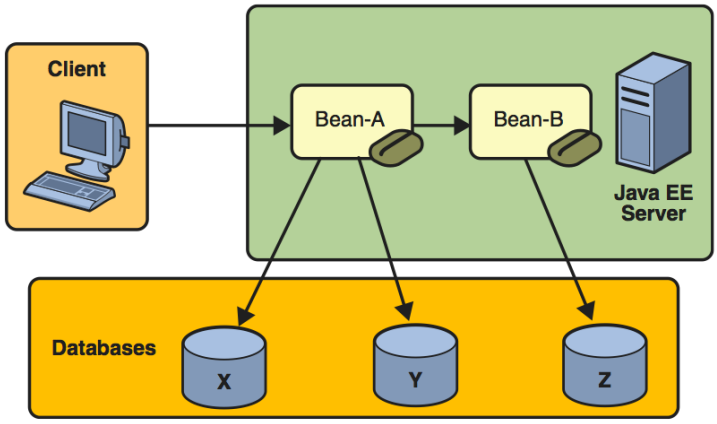
Los objetos `operacionService` y `usuarioService` son beans de sesión sin estado y los métodos `devolverEjemplar` y `multar` son métodos atómicos. Cada uno de ellos crea su propia transacción. JTA nos permite incorporar ambas llamadas en una única transacción, de forma que si uno de los métodos devuelve algún error se deshace toda la operación.

El código anterior puede ejecutarse en cualquier componente gestionado por el contenedor, como por ejemplo un servlet. También puede incluirse en un bean que define un servicio componiendo llamadas a otros servicios.

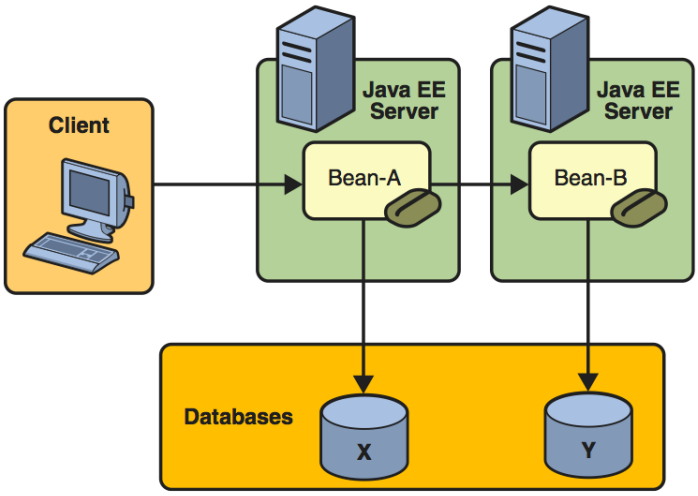
La otra forma de gestionar transacciones es gestionándolas de forma declarativa con anotaciones en los EJBs. Lo veremos más adelante.

2.1.2. Transacciones a través de múltiples bases de datos

Una de las características interesantes de JTA es que permite gestionar transacciones distribuidas que afectan a distintas bases de datos. Podemos utilizar distintas unidades de persistencia en los beans y agruparlas en una misma transacción. Las siguientes figuran muestran dos posibles escenarios:



En la figura anterior, el cliente invoca un método de negocio en el Bean-A. El método de negocio inicia una transacción, actualiza la base de datos X, actualiza la base de datos Y e invoca un método de negocio en el Bean-B. El segundo método de negocio actualiza la base de datos Z y devuelve el control al método de negocio en el Bean-A, que finaliza con éxito la transacción. Las tres actualizaciones de la base de datos ocurren en la misma transacción.



En la figura anterior, el cliente llama a un método de negocio en el Bean-A, que comienza una transacción y actualiza la base de datos X. Luego el Bean-A invoca un segundo método de negocio en el Bean-B, que reside en un servidor de aplicaciones remoto. El método en el Bean-B actualiza la base de datos Y. La gestión de transacciones de ambos servidores de aplicaciones se asegura de que ambas actualizaciones se realizan en la misma transacción. Es un ejemplo de una transacción distribuida a nivel de servidores de aplicaciones.

Para este segundo ejemplo necesitamos un servidor de aplicaciones compatible con la especificación completa de Java EE 6. No basta con un servidor compatible con el perfil Web.

2.2. Gestión de transacciones con beans de sesión

Los beans de sesión permiten dos tipos de gestión de transacciones, denominados CMP (*Container Managed Persistence*) y BMP (*Bean Managed Persistence*).

Cuando se utiliza CMP, los beans gestionan de forma automática las transacciones. Toda llamada a un método del bean es interceptada por el contenedor, que crea un contexto transaccional que dura hasta que el método ha terminado (con éxito o con fracaso, al lanzar una excepción). Si el método termina con éxito el contenedor hace un commit de la transacción. Si el método lanza una excepción el contenedor hace un rollback.

Cuando se utiliza BMP es el programador el que debe abrir y cerrar las transacciones utilizando JTA de forma explícita en el código de los métodos del bean.

Ambos enfoques se basan en JTA, que es utilizado por el servidor de aplicaciones para gestionar las transacciones distribuidas utilizando el algoritmo *two phase commit*. Los recursos que participan en estas transacciones deben ser fuentes de datos XA.

Un uso muy común de los beans de sesión es la implementación de la capa de negocio de la aplicación, definiendo métodos de servicio transaccionales. En cada método de servicio se realizan llamadas a la capa de persistencia definida por los DAO que gestionan las entidades del modelo. Los DAO utilizan entity managers inyectados por el contenedor. El contenedor se encarga automáticamente de inyectar el mismo entity manager en todos los DAO que participan en la misma transacción, de forma que todos los DAO van a compartir el mismo contexto de persistencia. Se libera a los DAO de la gestión de transacciones y su implementación se hace más sencilla y flexible.

Al utilizar JTA es posible llamar desde un método de servicio a otros servicios de la capa de negocio y englobar todo ello en una única transacción.

2.2.1. Gestión de transacciones BMT

En la gestión de transacciones denominada BMT (*Bean Managed Transactions*) el programador se hace cargo de la gestión de transacciones dentro de los métodos de los beans utilizando JTA. Ya hemos visto anteriormente un ejemplo cuando describimos JTA:

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class PedidoService {
   @Resource
   private UserTransaction userTransaction;

   public void hacerPedido(Item item, Cliente cliente) {
```

```
      try {
         userTransaction.begin();
         if (itemService.disponible(item)) {
            pedidoService.add(cliente,item);
            clienteService.anotaCargo(cliente,item);
            itemService.empaqueta(cliente,item);
         }
         userTransaction.commit();
      } catch (Exception e) {
         userTransaction.rollback();
         e.printStackTrace();
      }
   }
}
```

En la línea 2 se declara el tipo de gestión de transacción como BMT, para lo que se define el valor `TransactionManagementType.BEAN` como tipo de la anotación `@TransactionManagement`. En la línea 4 se declara el recurso `UserTransaction` que vamos a utilizar para controlar la transacción mediante JTA. Se define utilizando inyección de dependencias. En la línea 9 se inicia la transacción con la sentencia `userTransaction.begin()`. Las siguientes líneas de código definen las llamadas a los métodos transaccionales de otros beans de sesión. En estos métodos se utilizan conexiones XA que participan en la transacción definida en el bean. Si alguna de estas llamadas genera una excepción, ésta se recoge en la línea 16 y se llama a `userTransaction.rollback()`, con lo que automáticamente se deshace la transacción y todos los recursos implicados en la transacción vuelven a su estado original. En el caso en que todas las llamadas terminen correctamente, se realiza la llamada `userTransaction.commit();` que la transacción y este final se propaga a todos los recursos que intervienen en la transacción.

Vemos que el funcionamiento es idéntico al que hemos definido cuando hemos hablado de JTA. Todas las llamadas a los beans de sesión se incorporan a la misma transacción, *sin modificar el código de los métodos*. De esta forma no perdemos flexibilidad en los métodos de negocio, siguen siendo atómicos, se pueden utilizar de forma independiente y, además, pueden incorporarse distintas llamadas a la misma transacción.

Un problema de BMT es que no se puede unir una transacción definida en un método a una transacción ya abierta por el llamador. De esta forma, si el método anterior es llamado desde un cliente que tiene una transacción JTA abierta, esta transacción del llamador se suspende. JTA no permite transacciones anidadas. De la misma forma, los métodos dentro de la transacción definida en el bean (`pedidoBean.add()` y demás) no pueden a su vez abrir una nueva transacción con BMT, ya que estaríamos en el mismo caso que el que hemos mencionado, abriendo una transacción anidad. En el siguiente apartado veremos que la gestión de transacciones por el contenedor (CMT, *Containter Managed Transaction*) proporciona más flexibilidad, ya que, por ejemplo, permite que los métodos de los beans se incorporen a transacciones ya abiertas.

2.2.2. Gestión de transacciones CMT

Cuando verdaderamente aprovechamos todas las funcionalidades del contenedor EJB es cuando utilizamos el otro enfoque de la gestión de las transacciones en los enterprise beans, el llamado CMT (*Container Managed Transactions*). Aquí, el programador no tiene que escribir explícitamente ninguna instrucción para definir el alcance de la transacción, sino que es el contenedor EJB el que la maneja automáticamente. En este enfoque, los métodos de negocio de los beans se anotan y podemos configurar con anotaciones cómo se van a comportar. Veamos cómo definiríamos con CMT el mismo ejemplo anterior.

```
@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class PedidoService {

   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public void hacerPedido(Item item, Cliente cliente) {
      if (itemService.disponible(item)) {
         pedidoService.add(cliente,item);
         clienteService.anotaCargo(cliente,item);
         itemService.empaqueta(cliente,item);
      }
   }
}
```

Definimos en la clase el tipo de gestión de transacciones como `TransactionManagementType.CONTAINER`. El contenedor se encarga de la gestión de las transacciones del bean. Nosotros sólo debemos indicar en cada método qué tipo de gestión de transacción se debe hacer. En el método `hacerPedido` lo hemos definido como `REQUIRED` usando la anotación `@TransactionAttribute`.

La anotación `@TransactionAttribute` se puede realizar a nivel de método o a nivel de clase. En este último caso todos los métodos de la clase se comportan de la forma especificada.

El tipo de atributo de transacción REQUIRED obliga a que el método se ejecute dentro de una transacción. En el caso en que el llamador del método haya abierto ya una transacción, el contenedor ejecuta este método en el ámbito de esa transacción. En el caso en que el llamador no haya definido ninguna transacción, el contenedor inicia automáticamente una transacción nueva.

El tipo de atributo de transacción REQUIRED es el que tienen por defecto todos los métodos de los componentes EJB. Por eso se dice que son componentes transaccionales.

Todas las llamadas a otros métodos desde el bean se ejecutan en una transacción. Estas llamadas pueden ser a la capa de persistencia o a otros métodos de negocio.

Si el método no termina normalmente, sino que lanza una excepción de tipo `RuntimeException` (porque la lanza él mismo o alguna de las llamadas realizadas) se realiza automáticamente un rollback de la transacción. El contenedor captura la excepción

provocada por el método y llama a JTA para realizar un rollback.

Si se lanza una excepción chequeada que no es de tipo `RuntimeException` el rollback no se realiza automáticamente. En este caso hay que llamar al método `setRollbackOnly` del `EJBContext` que se puede obtener por inyección de dependencias:

```
@Stateless
public class PeliculaService {
   @Resource
   private EJBContext context;

   public void metodoTransaccional() throws AlgunaExcepcionChequeada {

      // Llamadas a otros métodos

      if (seHaProducidoError) {
         context.setRollbackOnly();
         throw new AlgunaExcepcionChequeada("Se ha producido un error");
      }
   }
}
```

Si el método del bean termina normalmente, el contenedor hace commit de la transacción.

El código resultante es muy sencillo ya toda la gestión de la transacción se realiza de forma implícita. En el código sólo aparece la lógica de negocio.

La utilización de las transacciones CMT tiene la ventaja añadida de que los métodos pueden participar en transacciones abiertas en otros métodos.

2.2.2.1. Propagación de transacciones

Existen seis posibles formas de propagar una transacción, definidas por los posibles tipos de atributos de transacción:

- `TransactionAttributeType.REQUIRED`
- `TransactionAttributeType.REQUIRES_NEW`
- `TransactionAttributeType.SUPPPORTS`
- `TransactionAttributeType.MANDATORY`
- `TransactionAttributeType.NOT_SUPPORTED`
- `TransactionAttributeType.NEVER`

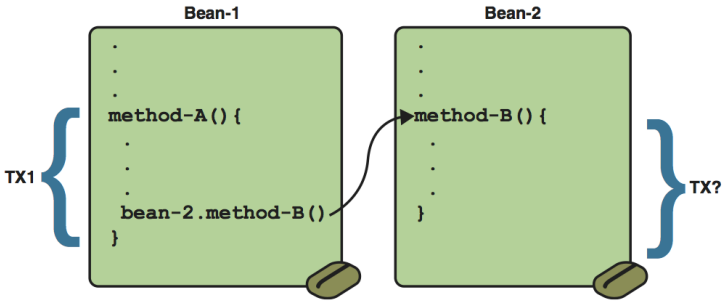
El funcionamiento de cada tipo de gestión depende de si el llamador del método hace la llamada con una transacción ya abierta o no. Vamos a ver un ejemplo. Supongamos que los métodos a los que se llama en `hacerPedido()` no son métodos de un DAO (un POJO), sino que se tratan de métodos de negocio de otros EJBs:

```
@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class PedidoEJB implements PedidoLocal {
   @Resource
   private SessionContext context;

   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public void hacerPedido(Item item, Cliente cliente) {
```

```
      try {
         if (itemEJB.disponible(item)) {
            pedidoEJB.add(cliente,item);
            clienteEJB.anotaCargo(cliente,item);
            itemEJB.empaqueta(cliente,item);
         } catch (Exception e) {
            context.setRollbackOnly();
         }
      }
   }
}
```

En este caso tenemos una situación como la que se muestra en la siguiente figura, en la que un método de negocio de un EJB llama a otro método de negocio. ¿Cómo van a gestionar la transacción los métodos llamados? Va a depender de qué tipo de `@TransactionAttribute` se hayan definido en los métodos.



- **`REQUIRED`**: se trata del tipo de gestión de transacciones por defecto. Si no declaramos el tipo en el método (ni en el bean) éste es el que se aplica. El código del método siempre debe estar en una transacción. Si el llamador realiza la llamada al método dentro de una transacción, el método participa en ella y se une a la transacción. En el caso de que el llamador no haya definido ninguna transacción, se crea una nueva que finaliza al terminar el método.

  Si la transacción falla en el método y éste participa en una transacción definida en el cliente, el contenedor deshará la transacción en el método y devolverá al cliente la excepción `javax.transaction.RollbackException`, para que el cliente obre en consecuencia.

  Si el método termina sin que se genere ninguna excepción y la transacción es nueva (el llamador no había definido ninguna), el contenedor hace un commit de la transacción. En el caso en que el método participe en una transacción abierta en el cliente, será éste el que deberá hacer el commit.
- **`REQUIRES_NEW`**: indica que el contenedor debe crear siempre una transacción nueva al comienzo del método, independientemente de si el cliente ha llamado al método con una transacción creada o no. La transacción en el llamador (si existe) queda suspendida hasta que la llamada termina. Además, el método llamado no participa en la transacción del llamador. El método llamado tiene su propia transacción y realiza un commit cuando termina. Si la transacción en el cliente falla después, el rollback ya

no afecta al método terminado.

Un ejemplo de uso de este atributo es una llamada a un método para escribir logs. Si hay un error en la escritura del log, no queremos que se propague al llamador. Por otro lado, si el llamador falla queremos que quede constancia de ello.

- **`SUPPORTS`**: el método hereda el estado transaccional del llamador. Si el llamador define una transacción, el método participa en ella. Si no, el método no se ejecuta en ninguna transacción.

  Suele utilizarse cuando el bean realiza operaciones de lectura que no modifican el estado de sus recursos transaccionales.
- **`MANDATORY`**: el cliente debe obligatoriamente crear una transacción antes de llamar a este método. Si se llama al método desde un entorno en el que no se ha abierto una transacción, se genera una excepción.

  Se usa muy raramente.
- **`NOT_SUPPORTED`**: el método se ejecuta sin crear una transacción. En el caso en que el llamador haya creado una, ésta se suspende, se ejecuta el método y después continua. Para el cliente el comportamiento es igual que `REQUIRES_NEW`. La diferencia es que en el bean no se ha creado ninguna transacción.

  Se suele usar sólo por MDB soportando un proveedor JMS no transaccional
- **`NEVER`**: se genera una excepción si el cliente invoca el método habiendo creado una transacción. No se usa casi nunca.

3. Temporizadores

Las aplicaciones que implementan flujos de trabajo de negocio tienen que tratar frecuentemente con notificaciones periódicas. El servicio temporizador del contenedor EJB nos permite planificar notificaciones para todos los tipos de enterprise beans excepto para los beans de sesión con estado. Podemos planificar una notificación para una hora específica, para después de un lapso de tiempo o a intervalos temporales. Por ejemplo, podríamos planificar los temporizadores para que lancen una notificación a las 10:30 AM del 23 de Mayo, en 30 días o cada 12 horas.

Los temporizadores van descontando el tiempo definido. Cuando llegan a cero (saltan), el contenedor llama al método anotado con la anotación `@Timeout` en la clase de implementación del bean. El método `@Timeout` contiene la lógica de negocio que maneja el evento temporizado.

3.1. El método Timeout

Los métodos anotados con `@Timeout` en la clase de implementación del bean deben devolver `void` y tomar un objeto `javax.ejb.Timer` como su único parámetro. No pueden

lanzar excepciones capturadas por la aplicación.

```
@Timeout
public void timeout(Timer timer) {
    System.out.println("TimerBean: timeout occurred");
}
```

3.2. Creando temporizadores

Para crear un temporizador, el bean debe invocar uno de los métodos `createTimer()` de la interfaz `TimerService`. Cuando el bean invoca el método `createTimer()`, el servicio temporizador comienza a contar hacia atrás la duración del timer.

Como ejemplo veamos el siguiente bean de sesión:

```
@Stateless
public class TimerSessionBean implements TimerSessionLocal {
    @Resource
    TimerService timerService;

    public void setTimer(long intervalDuration) {
        Timer timer = timerService.createTimer(
                    intervalDuration,
                    "Creado un nuevo timer");
    }

    @Timeout
    public void timeout(Timer timer) {
        System.out.println("Se ha lanzado el timeout");
    }
}
```

Vemos que el bean define un método `setTimer()` en el que se usa el objeto *timerService* obtenido en línea 3 con la anotación `@Resource`. En este método definimos un temporizador inicializándolo a los milisegundos determinados por `intervalDuration`.

El temporizador puede ser un temporizador de parada única, en el que podemos indicar el momento de la parada o el tiempo que debe transcurrir antes de la misma, o un temporizador periódico, que se lanza a intervalos regulares. Básicamente son posibles cuatro tipos de temporizadores:

- Parada única con tiempo: `createTimer(long timeoutDuration, Serializable info)`
- Parada única con fecha: `createTimer(Date firstDate, Serializable info)`
- Periódico con intervalos regulares y tiempo inicial: `createTimer(long timeoutDuration, long timeoutInterval, Serializable info)`
- Periódico con intervalos regulares y fecha inicial: `createTimer(Date firstDate, long timeoutInterval, Serializable info)`

Veamos un ejemplo de utilización de temporizadores periódico con fecha inicial. Supongamos que queremos un temporizador que expire el 1 de Mayo de 2008 a las 12h. y que, a partir de esa fecha, expire cada tres días:

```
Calendar unoDeMayo = Calendar.getInstance();
```

```
unoDeMayo.set(2008, Calendar.MAY, 1, 12, 0);
long tresDiasEnMilisecs = 1000 * 60 * 60 * 24 * 3;
timerService.createTimer(unoDeMayo.getTime(),
                         tresDiasEnMilisecs,
                         "Mi temporizador");
```

Los parámetros `Date` y `long` representan el tiempo del temporizador con una resolución de milisegundos. Sin embargo, debido a que el servicio de temporización no está orientado hacia aplicaciones de tiempo real, el callback al método `@Timeout` podría no ocurrir con precisión de milisegundos. El servicio de temporización es para aplicaciones de negocios, que miden el tiempo normalmente en horas, días o incluso duraciones más largas.

Los temporizadores son persistentes. Si el servidor se apaga (o incluso se cae), los temporizadores se graban y se activarán de nuevo en el momento en que el servidor vuelve a poner en marcha. Si un temporizador expira en el momento en que el servidor está caído, el contenedor llamará al método `@Timeout` cuando el servidor se vuelva a comenzar.

3.3. Cancelando y grabando temporizadores

Los temporizadores pueden cancelarse con los siguientes eventos:

- Cuando un temporizador de tiempo expira, el contenedor EJB llama al método `@Timeout` y cancela el temporizador.
- Cuando el bean invoca el método `cancel` de la interfaz `Timer`, el contenedor cancela el timer.

Para grabar un objeto `Timer` para rerferencias futuras, podemos invocar su método `getHandle()` y almacenar el objeto `TimerHandle` en una base de datos (un objeto `TimerHandle` es serializable). Para reinstanciar el objeto `Timer`, podemos recuperar el `TimerHandle` de la base de datos e invocar en él el método `getTimer()`. Un objeto `TimerHandle` no puede pasarse como argumento de un método definido en un interfaz remoto. Esto es, los clientes remotos no pueden acceder al `TimerHandle` del bean. Los clientes locales, sin embargo, no sufren esta restricción.

3.4. Obteniendo información del temporizador

La interfaz `Timer` define también métodos para obtener información sobre los temporizadores:

```
public long getTimeRemaining();
   public java.util.Date getNextTimeout();
   public java.io.Serializable getInfo();
```

El método `getInfo()` devuelve el objeto que se pasó como último parámetro de la invocación `createTimer`. Por ejemplo, en el código anterior esta información es la cadena "`Creado un nuevo timer`".

También podemos recuperar todos los temporizadores activos de un bean con el método `getTimers()`, que devuelve una colección de objetos `Timer`.

3.5. Temporizadores y transacciones

Los enterprise bean normalmente crean temporizadores dentro de transacciones. Si la transacción es anulada, también se anula automáticamente la creación del temporizador. De forma similar, si un bean cancela un temporizador dentro de una transacción que termina siendo anulada, la cancelación del temporizador también se anula.

En los beans que usan transacciones gestionadas por el contenedor, el método `@Timeout` tiene normalmente el atributo de transacción `REQUIRES` o `REQUIRES_NEW` para preservar la integridad de la transacción. Con estos atributos, el contenedor EJB comienza una nueva transacción antes de llamar al método `@Timeout`. De esta forma, si la transacción se anula, el contenedor volverá a llamar al método `@Timeout` de nuevo con los valores iniciales del temporizador.

3.6. Ventajas y limitaciones

Entre las ventajas de los temporizadores frente a la utilización de algún otro tipo de planficadores podemos destacar:

- Los temporizadores son parte del estándar Java EE con lo que la aplicación será portable y no dependerá de APIs propietarias del servidor de aplicaciones.
- La utilización del servicio de temporización de EJB viene incluido en Java EE y no tiene ningún coste adicional. No hay que realizar ninguna configuración de un planificador externo y el desarrollador no tiene que preocuparse de buscar uno.
- El temporizador es un servicio gestionado por el contenedor, y no se requiere un thread separado como en un planificador externo.
- Las transacciones se soportan completamente.
- Por defecto, los temporizadores son objetos persistentes que sobreviven a caídas del contenedor.

Limitaciones

- La mayoría de planificadores proporcionan la posibilidad de utilizar clases Java estándar; sin embargo los temporizadores requieren el uso de enterprise beans.
- Los temporizadores EJB adolecen del soporte de temporizadores al estilo cron, fechas de bloqueo, etc. disponibles en muchos planificadores de tareas.

4. Interceptores

Los *interceptores* (*interceptors*) son objetos que son capaces de interponerse en las llamadas a los métodos en los eventos de ciclo de vida de los beans de sesión y de

mensaje. Nos permiten encapsular conductas comunes a distintas partes de la aplicación que normalmente no tienen que ver con la lógica de negocio. Los interceptores son una característica avanzada de la especificación EJB que nos permite modularizar la aplicación o incluso extender el funcionamiento del contenedor EJB. En esta sesión veremos una introducción a la definición y al funcionamiento de los interceptores para interponerse en las llamadas a los métodos de negocio.

4.1. La clase Interceptor

Comencemos con un ejemplo. Supongamos que queremos analizar el tiempo que tarda en ejecutarse un determinado método de negocio de un bean.

```
public void addMensajeAutor(String nombre, String texto) {
   long startTime = System.currentTimeMillis();
   Autor autor = findAutor(nombre);
   if (autor == null) {
      autor = new Autor();
      autor.setNombre(nombre);
      em.persist(autor);
   }
   mensajeService.addMensaje(texto, nombre);
   long endTime = System.currentTimeMillis() - startTime;
   System.out.println("addMensajeAutor() ha tardado: " +
            endTime + " (ms)");
}
```

Aunque el método compilará y se ejecutará correctamente, el enfoque tiene serios problemas de diseño:

- Se ha añadido en el método `addMensajeAutor()` código que no tiene nada que ver con la lógica de negocio de la aplicación. El código se ha hecho más complicado de leer y de mantener.
- El código de análisis no se puede activar y desactivar a conveniencia. Hay que comentarlo y volver a recompilar la aplicación.
- El código de análisis es una plantilla que podría reutilizarse en muchos métodos de la aplicación. Pero escrito de esta forma habría que escribirlo en todos los métodos en los que queramos aplicarlo.

Los interceptores proporcionan un mecanismo para encapsular este tipo de código de una forma sencilla y aplicarlo a nuestros métodos sin interferir directamente. Los interceptores proporcionan una estructura para este tipo de conducta de forma que puedan ser ampliados y extendidos fácilmente en una clase. Por último, proporcionan un mecanismo simple y configurable para aplicar la conducta en el lugar que deseemos.

Este tipo de código que *envuelve* los métodos de la aplicación se suele denominar también *aspecto* y es la base de una técnica de programación denominada AOP (*Aspect Oriented Programming*). Un *framework* muy popular de programación dirigida por aspectos en Java es AspectJ.

Vamos ya a detallar cómo implementar los interceptores. Es muy sencillo. Basta con

crear una clase Java con un método precedido por la anotación `@javax.interceptor.AroundInvoke` y con la siguiente signatura:

```
@AroundInvoke
Object <nombre-metodo>(javax.interceptor.InvocationContext invocation)
   throws Exception;
```

El método `@AroundInvoke` en una clase de intercepción envuelve la llamada al método de negocio y es invocado en la misma pila de llamada, en la misma transacción y en el mismo contexto de seguridad que el método que se está interceptando. El parámetro `javax.interceptor.InvocationContext` es una representación genérica del método de negocio que el cliente está invocando. A través de él podemos obtener información como el bean al que se está llamando, los parámetros que se están pasando en forma de un array de objetos, y una referencia al objeto `java.lang.reflect.Method` que contiene la representación del método invocado. `InvocationContext` también se usa para dirigir realmente la invocación. Veamos cómo utilizar este enfoque para hacer el análisis del tiempo anterior:

```
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

public class Profiler {

   @AroundInvoke
   public Object profile(InvocationContext invocation)
            throws Exception {
      long startTime = System.currentTimeMillis();
      try {
         return invocation.proceed();
      } finally {
         long endTime = System.currentTimeMillis() - startTime;
         System.out.println("El método " + invocation.getMethod() +
                  " ha tardado " + endTime + " (ms)");
      }
   }
}
```

El método anotado con `@AroundInvoke` en nuestra clase interceptora es el método `profile`. Tiene un aspecto muy parecido al código que escribimos en el método `addMensajeAutor`, con la excepción de que no incluimos la lógica de negocio. En la línea 11 se llama al método `InvocationContext.proceed()`. Si se debe llamar a otro interceptor el método `proceed()` realiza esa llamada. Si no hay ningún otro interceptor en espera, el contenedor EJB llama entonces al método del bean que está siendo invocado por el cliente.

En las líneas 13 y 15 se calcula el tiempo de ejecución y se imprime en la salida estándar. El método `InvocationContext.getMethod()` proporciona acceso al objeto `java.lang.reflect.Method` que representa el método real que se está invocando. Se usa en la línea 14 para imprimir el nombre del método al que se está llamando.

La estructura del código permite capturar las posibles excepciones que pudiera generar la invocación del método del bean. Al poner el cálculo del tiempo final en la parte `finally` nos aseguramos de que siempre se ejecuta.

Además del método `getMethod()`, la interface `InvocationContext` tiene otros métodos interesantes:

```
package javax.interceptor;

public interface InvocationContext {
   public Object getTarget();
   public Method getMethod();
   public Object[] getParameters();
   public void setParameters(Object[] newArgs);
   public java.util.Map<String, Object> getContextData();
   public Object proceed() throws Exception;
}
```

El método `getTarget()` devuelve una referencia a la instancia del bean objetivo. Podríamos cambiar nuestro método `profile` para que imprimiera los parámetros del método al que se está invocando utilizando el método `getParameters()`. El método `setParameters()` permite modificar los parámetros de la invocación al método del bean, de forma que podemos hacer que el bean ejecute el código con los parámetros proporcionados por el interceptador. El método `getContextData()` devuelve un objeto `Map` que está activo durante toda la invocación al método. Los interceptores pueden utilizar este objeto para pasarse datos de contexto entre ellos durante la misma invocación.

Una vez que hemos escrito la clase de intercepción, es hora de aplicarlo a la llamada al bean. Podemos hacerlo utilizando anotaciones o utilizando descriptores de despliegue XML. Veremos ambas opciones en los siguientes apartados.

4.2. Aplicando interceptores con anotaciones

La anotación `@javax.interceptor.Interceptors` se puede usar para aplicar interceptores a un método particular de un bean o todos los métodos de un bean. Para aplicar el método anterior de profiling basta con aplicar la anotación en su definición:

```
  @Interceptors(Profiler.class)
   public void addMensajeAutor(String nombre, String texto) {
      // ...
   }
```

La anotación `@Interceptors` también puede aplicarse a todos los métodos de negocio de un bean realizando la anotación en la clase:

```
@Stateless
@Interceptors(Profiler.class)
public class AutorServiceBean implements AutorServiceLocal {

   @PersistenceContext(unitName = "ejb-jpa-ejbPU")
   EntityManager em;
   @EJB
   MensajeServiceDetachedLocal mensajeService;

   public Autor findAutor(String nombre) {
      return em.find(Autor.class, nombre);
   }
```

```
   // ...
```

4.3. Aplicando interceptores con XML

Aunque la anotación `@Interceptors` nos permite aplicar fácilmente los interceptores, tiene el inconveniente de que nos obliga a modificar y recompilar el código cada vez que queremos añadirlos o deshabilitarlos. Es más interesante definir los interceptores mediante XML, en el fichero descriptor de despliegue `ejb-jar.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns = "http://java.sun.com/xml/ns/javaee"
         version = "3.0"
         xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation = "http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
<assembly-descriptor>
   <interceptor-binding>
      <ejb-name>AutorServiceBean</ejb-name>
      <interceptor-class>
             org.especialistajee.ejb.interceptor.Profiler
          </interceptor-class>
      <method>
         <method-name>addMensajeAutor</method-name>
      </method>
   </interceptor-binding>
</assembly-descriptor>
</ejb-jar>
```

Una de las ventajas de utilizar el descriptor de despliegue XML es que es posible aplicar el interceptor a todos los métodos de todos los beans desplegados en el módulo:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns = "http://java.sun.com/xml/ns/javaee"
         version = "3.0"
         xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation = "http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
<assembly-descriptor>
   <interceptor-binding>
      <ejb-name>*</ejb-name>
<interceptor-class>es.ua.jtech.ejb.interceptor.Profiler</interceptor-class>
   </interceptor-binding>
</assembly-descriptor>
</ejb-jar>
```