



## ***Frameworks de persistencia - JPA***

Sesión 6 - Transacciones y JPA en Java EE



## Índice

- Introducción
- Transacciones *entity manager*: Interfaz EntityTransaction
- Transacciones JTA
- Concurrencia
- Gestión optimista en JPA
- Bloqueos en JPA



## Características generales

- ACID
  - A = atómicas
  - I = aisladas
- En JDBC
  - autocommit = true por defecto
  - setAutocommit(false) para gestionar explícitamente las transacciones
  - Aislamiento dependiente de la BD: bloqueos y MMVC
- JPA: capa sobre JDBC
  - bloqueos optimistas



## Interfaz EntityTransaction

- Las transacciones JPA en aplicaciones standalone Java SE se basan directamente en la gestión de transacciones de la BD
- Se gestionan a partir del entity manager, gestionando el objeto `EntityTransaction` devuelto por el método `getTransaction` del entity manager
- Interfaz `EntityTransaction`:

```
public interface EntityTransaction {  
    public void begin();  
    public void commit();  
    public void rollback();  
    public void setRollbackOnly();  
    public boolean getRollbackOnly();  
    public boolean isActive();}
```



## Excepciones

- Si se intenta comenzar una transacción cuando otra ya está activa se lanza una `IllegalStateException`
- Si se llama a `commit()` o `rollback()` con una transacción no activa se lanza una `IllegalStateException`
- Si ocurre un error durante el rollback se lanza una `PersistenceException`
- Si ocurre un error durante el commit se lanza una `RollbackException` (hija de `PersistenceException`)
- Todas las excepciones son de tipo `RuntimeException`



## Código con transacciones y excepciones

- Cuando se deshace una transacción en la base de datos todos los cambios realizados durante la transacción se deshacen también.
- Cuidado: el modelo de memoria de Java no es transaccional. No hay forma de obtener una instantánea de un objeto y revertir su estado a ese momento si algo va mal.

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
try {
    tx.begin();
    // Operacion sobre entidad 1
    // Operacion sobre entidad 2
    tx.commit();
} catch (RuntimeException ex) {
    try {
        tx.rollback();
    } catch (RuntimeException rbEx) {
        log.error("No se ha podido deshacer la transacción", rbEx);
    }
    throw ex;
} finally {
    em.close();
}
```



## Transacciones JTA

- JTA (Java Transaction API) permite gestionar transacciones distribuidas en las que intervienen más de una base de datos
- La transacción se gestiona por el servidor de aplicaciones a través de un objeto `UserTransaction`
- El objeto se obtiene por inyección de dependencias con la anotación `@Resource`:

```
@Resource  
UserTransaction utx;
```

- Se puede obtener en cualquier objeto gestionado por el servidor de aplicaciones: un servlet, un EJB, ...



## API JTA

- Muy similar a la de EntityTransaction

```
public interface UserTransaction {  
    public void begin();  
    public void commit();  
    public void rollback();  
    public int getStatus();  
    public void setRollbackOnly();  
    public void setTransactionTimeout(int segundos);  
}
```

- Veremos un ejemplo al final de la clase, cuando veamos el código de JPA en un servlet





## Concurrencia

- Problema complejo que impacta directamente en el rendimiento de la aplicación
- Ejemplos variados: reserva de asiento, lectura/escritura de valores

```
public void reservaAsiento(Pasajero pasajero, int numAsiento, long numVuelo) {
    EntityManager em = emf.createEntityManager();
    AsientoKey key = new AsientoKey(numAsiento, numVuelo);
    em.getTransaction().begin();
    Asiento asiento = em.find(Asiento.class, key);
    if (!asiento.getOcupado()) {
        asiento.setOcupado(true);
        asiento.setPasajero(pasajero);
        pasajero.setAsiento(asiento);
    }
    em.getTransaction().commit();
    em.close();
}
```



## Ejemplo 2

- La concurrencia añade un elemento de complejidad adicional a la gestión de transacciones cuando se trabaja con caches y copias en memoria
- Supongamos las siguientes operaciones. Si se ejecutan concurrentemente 2 clientes es posible que X quede en un estado inconsistente.

*Cliente 1*

Leer el dato X  
Sumar 10 a X  
Escribir el dato X

*Cliente 2*

Leer el dato X  
Sumar 10 a X  
Escribir el dato X



## Solución en JPA

- En JPA los bloqueos explícitos sólo se soportan en la especificación 2.0
- Bloqueos optimistas, utilizando versiones
- Hay que definir un atributo con la anotación `@Version`, aunque en Hibernate no es necesario

```
@Entity
public class Autor {
    @Id
    private String nombre;
    @Version
    private int version;
    private String correo;
    ...
}
```

```
@Entity
@org.hibernate.annotations.Entity (
    optimisticLock = OptimisticLockType.ALL,
    dynamicUpdate = true
)
public class Autor {
    @Id
    private String nombre;
    private String correo;
    ...
}
```



## Bloqueos optimistas

- Una transacción T1 realiza una lectura sobre un objeto. Se obtiene automáticamente su número de versión.
- Otra transacción T2 modifica el objeto y realiza un commit. Automáticamente se incrementa su número de versión.
- Si ahora la transacción T1 intenta modificar el objeto se comprobará que su número de versión es mayor que el que tiene y se generará una excepción.
- En este caso el usuario de la aplicación que esté ejecutando la transacción T1 obtendrá un mensaje de error indicando que alguien ha modificado los datos y que no es posible confirmar la operación. Lo deberá intentar de nuevo.



## Control optimista

*Cliente 1*

Copiar X, guardando  
el número de versión

Sumar 10 a X

Si el número de versión  
de X es el mismo que la  
copia: escribirlo y aumentar  
el número de versión

Sino: abortar la transacción

*Cliente 2*

Copiar X, guardando  
el número de versión

Sumar 10 a X

Si el número de versión  
de X es el mismo que la  
copia: escribirlo y aumentar  
el número de versión

Sino: abortar la transacción



## Bloqueos explícitos

- Estándar a partir de la especificación 2.0 de JPA

```
public void reservaAsiento(Pasajero pasajero, int numAsiento, long numVuelo) {
    EntityManager em = emf.createEntityManager();
    AsientoKey key = new AsientoKey(numAsiento, numVuelo);
    em.getTransaction().begin();
    Asiento asiento = em.find(Asiento.class, key);
    em.lock(asiento, LockType.READ);
    if (!asiento.getOcupado()) {
        asiento.setOcupado(true);
        asiento.setPasajero(pasajero);
        pasajero.setAsiento(asiento);
    }
    em.getTransaction().commit();
    em.close();
}
```



## Tipos de bloqueos

Tipo de bloqueo	Descripción
OPTIMISTIC	Obtiene un bloqueo de lectura optimista en la entidad si tiene un atributo de versión
OPTIMISTIC_FORCE_INCREMENT	Obtiene un bloqueo de lectura optimista en la entidad si tiene un atributo de versión e incrementa el valor de versión del atributo
PESSIMISTIC_READ	Obtiene un bloqueo de lectura de larga duración sobre un dato para prevenir que sea borrado o modificado. Otras transacciones pueden leer los datos mientras que se mantiene el bloque, pero no pueden modificarlo o borrarlo.
PESSIMISTIC_WRITE	Obtiene un bloque de escritura de larga duración sobre un dato para prevenir que sea leído, modificado o borrado.
PESSIMISTIC_FORCE_INCREMENT	Obtiene un bloqueo e incrementa el atributo de versión
READ	Sinónimo de OPTIMISTIC. Es preferible utilizar este último.
WRITE	Sinónimo de OPTIMISTIC_FORCE_INCREMENT. Es preferible usar este último.
NONE	No se realiza ningún bloqueo sobre los datos.



## JPA en aplicaciones Java EE

- Cambios con respecto a aplicaciones standalone Java SE:
  - Conexión con la BD en `persistence.xml`: no se accede directamente a la BD a través de un driver, sino a través de una fuente de datos configurada en el servidor de aplicaciones
  - Tipo de transacción en `persistence.xml`: se usa JTA
  - Los entity managers y las unidades de persistencia se obtienen por inyección de dependencias, usando las anotaciones `@PersistenceContext` y `@PersistenceUnit`
  - Se define la anotación `@transactional` para hacer transaccionales métodos o clases
  - Podemos definir los DAOs y las clases de servicio como beans gestionados e inyectarlos usando CDIs





## Creación de fuente de datos MySQL en WildFly

- Seguir los apuntes para configurar una fuente de datos `MysqlXADataSource` en la consola de WildFly
- Importante el nombre JNDI:  
`java:/datasources/MensajesDS`

The screenshot shows the WildFly 8.2.1.Final administration console. The 'Configuration' tab is active, and the 'DATASOURCES' sub-system is selected. The 'JDBC XA Datasources' configuration page is displayed, showing a table of available datasources. The 'MensajesDS' datasource is highlighted, and its configuration details are shown below.

Name	JNDI	Enabled?
MensajesDS	java:/datasources/MensajesDS	<input checked="" type="checkbox"/>

Attributes: [Edit](#)

Name:	MensajesDS
JNDI:	java:/datasources/MensajesDS
XA Data Source Class:	com.mysql.jdbc.jdbc2.optional.MysqlXADataSource
Is enabled?:	true
Statistics enabled?:	false
Driver:	mysql-connector-java-5.1.33.jar_com.mysql.jdbc.Driver_5_1



## Fichero persistence.xml

```
<?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">
    <jta-data-source>java:/datasources/MensajesDS</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="update" />
      <property name="hibernate.show_sql" value="true" />
    </properties>
  </persistence-unit>
</persistence>
```



## Inyección del entity manager en un servlet (sin DAOs)

```
@WebServlet(name="holamundo", urlPatterns="/holamundo")
public class HolaMundo extends HttpServlet {

    @PersistenceContext(type = PersistenceContextType.TRANSACTION,
                        unitName = "mensajes")
    EntityManager em;
    @Resource
    UserTransaction tx;

    protected void doGet(HttpServletRequest request,
                        HttpServletResponse response)
        throws ServletException, IOException {
        try {
            tx.begin();
            AutorDao autorDao = new AutorDao(em);
            Autor autor = autorDao.find(1L);
            autor.setNombre("Nuevo Nombre");
            autorDao.update(autor);
            tx.commit();
        } catch (Exception e) {
            try {
                tx.rollback();
            } catch (javax.transaction.SystemException e1) {
                throw new ServletException(e);
            }
        }
    }
}
```



## Modificación del DAO

```
abstract class Dao<T, K> {  
  
    @PersistenceContext(unitName = "mensajes")  
    EntityManager em;  
  
    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);  
    }  
  
    public abstract T find(K id);  
}
```

Se inyecta el entity manager usando la anotación `@PersistenceContext` con el nombre de la unidad de persistencia



## Modificación del servicio

```
@Transactional
public class AutorServicio {

    @Inject
    AutorDao autorDao;
    @Inject
    MensajeDao mensajeDao;

    public Autor createAutorMensaje(String nombre,
                                    String correo,
                                    String texto) {
        Autor autor = new Autor(nombre, correo);
        autor = autorDao.create(autor);
        Mensaje mensaje = new Mensaje(texto, autor);
        mensaje = mensajeDao.create(mensaje);
        return autor;
    }

    public List<Autor> listAllAutores() {
        List<Autor> autores = autorDao.listAllAutores();
        return autores;
    }
}
```

La anotación `@Transactional` define transacciones en todos los métodos.

Se inyectan los DAOs



## Inyección de la clase servicio en un servlet

```
@WebServlet(name="nuevoAutorMensaje", urlPatterns="/nuevoAutorMensaje")
public class NuevoAutorMensaje extends HttpServlet {

    @Inject
    AutorServicio autorServicio;

    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response) throws
        ServletException,
        IOException {

    }

    protected void doGet(HttpServletRequest request,
                          HttpServletResponse response)
        throws ServletException, IOException {
        String login = request.getParameter("login");
        String mensaje = request.getParameter("mensaje");
        System.out.println(login);
        System.out.println(mensaje);

        autorServicio.createAutorMensaje(login, login, mensaje);

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<!DOCTYPE HTML PUBLIC \"\" +
            \"-//W3C//DTD HTML 4.0 \" +
            \"Transitional//EN\">");
        out.println("<HTML>");
        out.println("<BODY>");
        out.println("<h3>Autor y mensaje correctos</h3>");
        out.println("</BODY>");
        out.println("</HTML>");
    }
}
```

Se inyecta la clase servicio



**¿Preguntas?**