



# **CURSO DE HIBERNATE 5**

  
**OpenWebinars**



# HIBERNATE

## (2) HIBERNATE



Más que un ORM.  
Comparativa con  
otros productos. JPA.  
Maven. Módulos



## (4) ENTIDADES

Definición del modelo  
del dominio. Entidades  
y ciclo de vida. XML y  
anotaciones. Tipos de  
datos.



## (1) INTRODUCCION

Persistencia, desfase  
objeto-relacional,  
ORM. Productos y  
estándares



## (3) PRIMER PROYECTO

Hibernate.cfg.xml,  
EntityManager y  
persistence.xml



## (5) ASOCIACIONES

ManyToOne, OneToMany,  
OneToOne, ManyToMany



# HIBERNATE



## (6) ELEMENTOS AVANZADOS

Campos calculados,  
herencia.

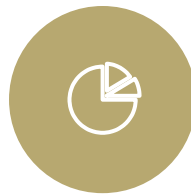
## (7) COLECCIONES

Mapeo de colecciones.  
Tipos (list, set, map).  
Colecciones  
ordenadas (sorted vs.  
ordered).



## (9) CONTEXTO DE PERSISTENCIA

Almacenamiento,  
recuperación y borrado  
de entidades.



## (10) TRANSACCIONES

Control de concurrencia.  
Patrones y antipatrones.

## (8) GENERACION DEL ESQUEMA

Customización del  
proceso de  
generación del  
esquema.





# HIBERNATE

## (12) ENVERS



Introducción a la  
auditoria de entidades.



## (11) CONSULTAS HPQL VS JPQL

Consultas con  
parámetros,  
Anotaciones. SQL nativo






1.

**TRANSACCIONES**

# TRANSACCIONES

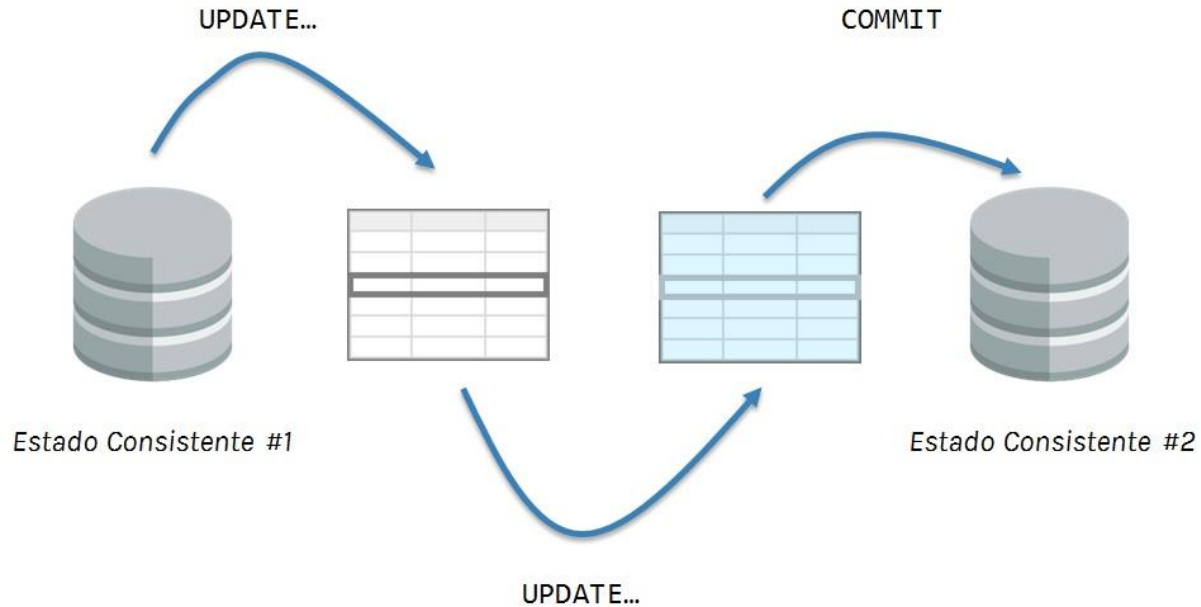
TITULAR	SALDO
Pepe	500
Juan	300

TITULAR	SALDO
Pepe	400
Juan	400

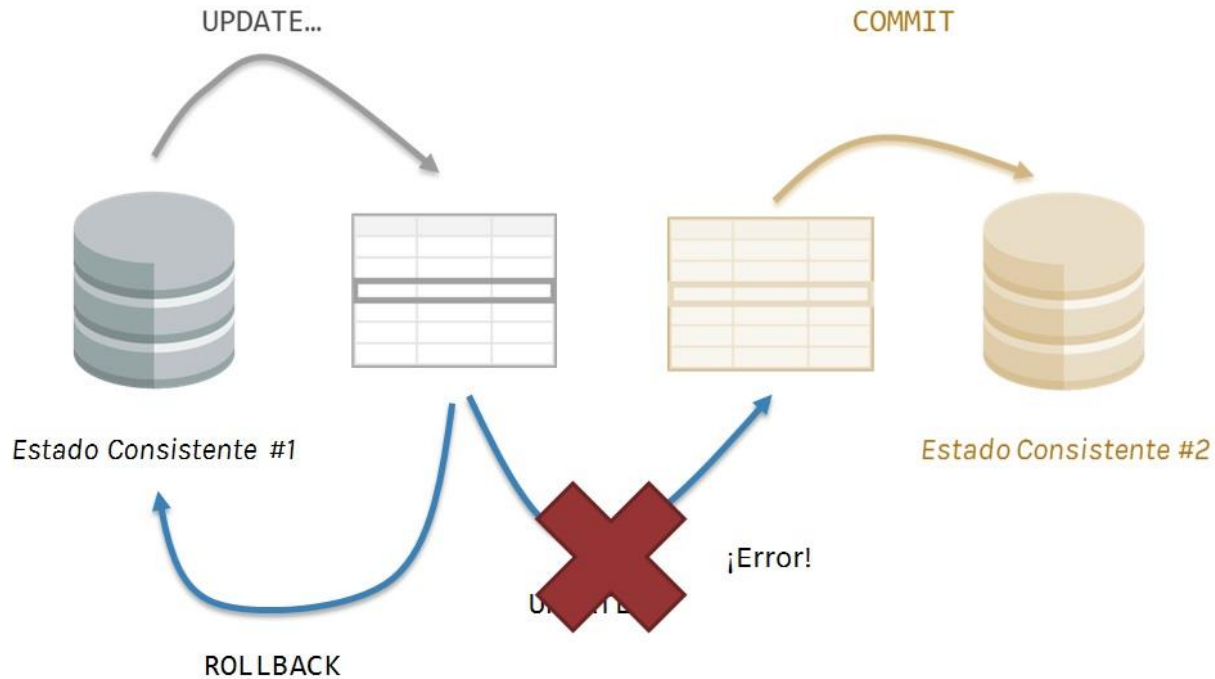


Transferencia  
De Pepe a Juan  
de 100€

# OPERACIONES ATÓMICAS



# ROLLBACK







2.

**RESOURCE LOCAL vs  
JTA**

## RESOURCE LOCAL

- ▶ Se delega la responsabilidad de las transacciones en el programador.
- ▶ Obtenidas desde el ***entityManager***.
- ▶ De tipo *javax.persistence.EntityTransaction*
- ▶ Manejo a través de los métodos *begin()*, *commit()* y *rollback()*.

# RESOURCE LOCAL

```
try {  
    //Iniciamos una transacción  
    em.getTransaction().begin();  
    //Persistimos los datos  
    em.persist(userAccount);  
    //Comitemos la transacción  
    em.getTransaction().commit();  
    System.out.println("El objeto ha sido dado de alta correctamente. Muchas gracias.");  
} catch (Exception e) {  
    System.out.println("El objeto no ha sido dado de alta correctamente. Disculpe las molestias");  
    System.err.println(e.getMessage());  
    if (em.getTransaction().isActive()) {  
        em.getTransaction().rollback();  
    }  
}
```

# JTA (Java Transaction Api)

- ▶ Asociado a Java EE
- ▶ El servidor de aplicaciones nos tiene que dar soporte para usarlo.
- ▶ Spring también nos lo ofrece.

```
@Bean
public JpaTransactionManager transactionManager() {
    JpaTransactionManager transactionManager = new JpaTransactionManager();
    transactionManager.setEntityManagerFactory(entityManagerFactory.getObject());
    return transactionManager;
}
```

```
@Repository
@Transactional
public class UserDao {
```



3.

## PATRONES Y ANTIPATRONES

# SESIÓN POR OPERACIÓN

## (ANTIPATRÓN)

- ▶ Un *Session* (o *EntityManager*) por cada consulta u operación.
- ▶ Se deben agrupar operaciones en transacciones.
- ▶ No es positivo el uso de ***autocommit***.

# SESIÓN POR PETICIÓN (PATRÓN)

- ▶ Un *Session* (o *EntityManager*) por cada petición (*request*) al sistema.
- ▶ Aplicaciones web.
- ▶ Se recibe, se hacen todas las operaciones con la base de datos, se cierra y se envía respuesta al cliente.

# CONVERSACIONES LARGAS

## (PATRÓN)

- ▶ En determinados contextos, no se puede usar el patrón sesión-por-petición.
- ▶ Orientado a sesiones de trabajo largas.
- ▶ Problemas de conflictos entre usuarios.
  - Versionado automático: control optimista de la concurrencia.
  - Objetos separados: sesión-por-petición-con-objetos-separados para cada usuario. Posterior fusión. Versionado automático.
  - Sesión extendida: desconexión y reconexión a la base de datos.



# SESIÓN POR APLICACIÓN

## (ANTIPATRÓN)

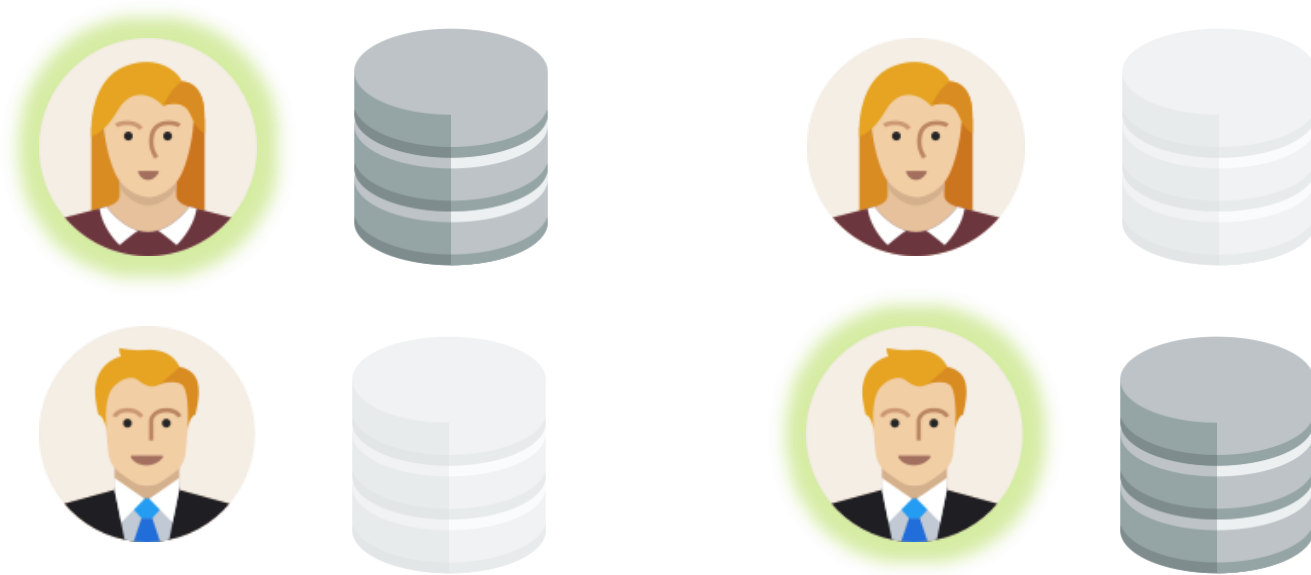
- ▶ Un *Session* (o *EntityManager*) para toda la aplicación.
- ▶ Problemas de concurrencia (no ThreadSafe).
- ▶ Cierre de sesión cuando sucede una excepción (¿qué sucede con el resto de usuarios?).



4.

## CONTROL DE CONCURRENCIA

# SENSACIÓN DE AISLAMIENTO

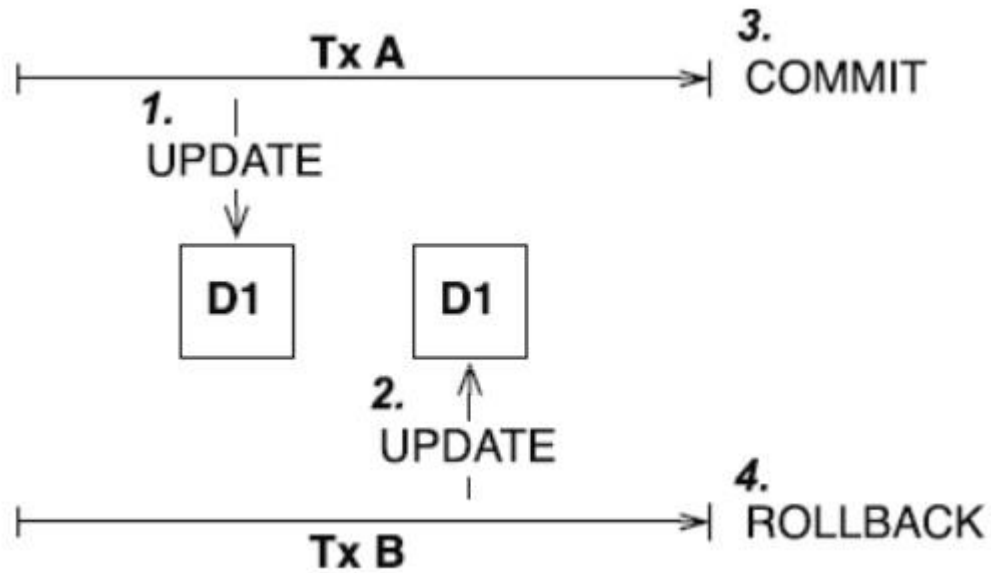


# PROBLEMAS

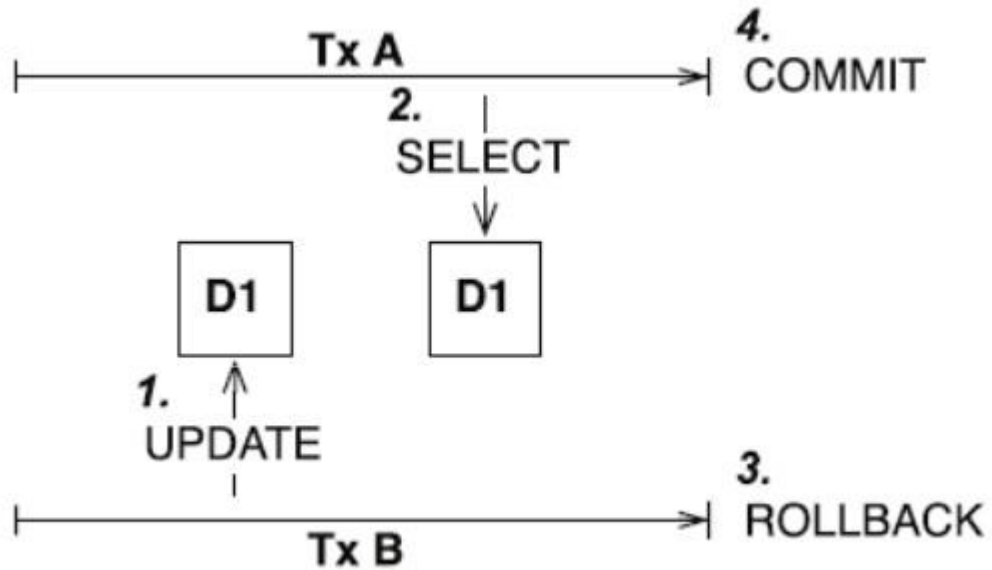
ASOCIADOS A LA

# CONCURRENCIA

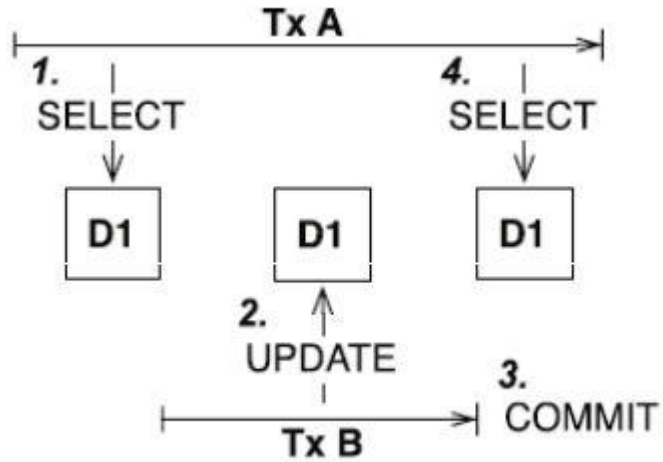
# LOST UPDATE



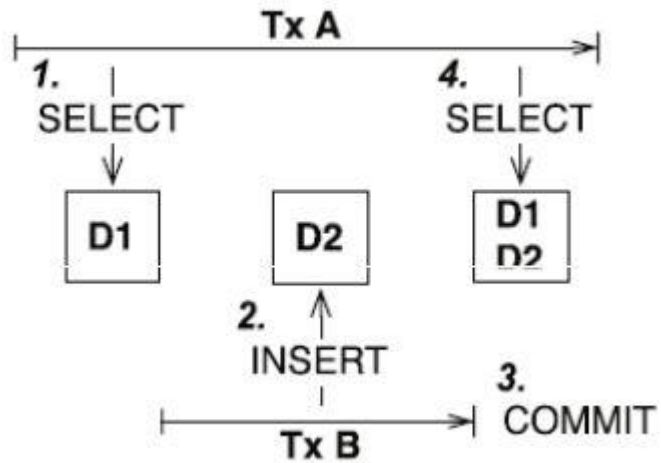
## DIRTY READ



# NON REPETEABLE READ



# PHANTOM READ





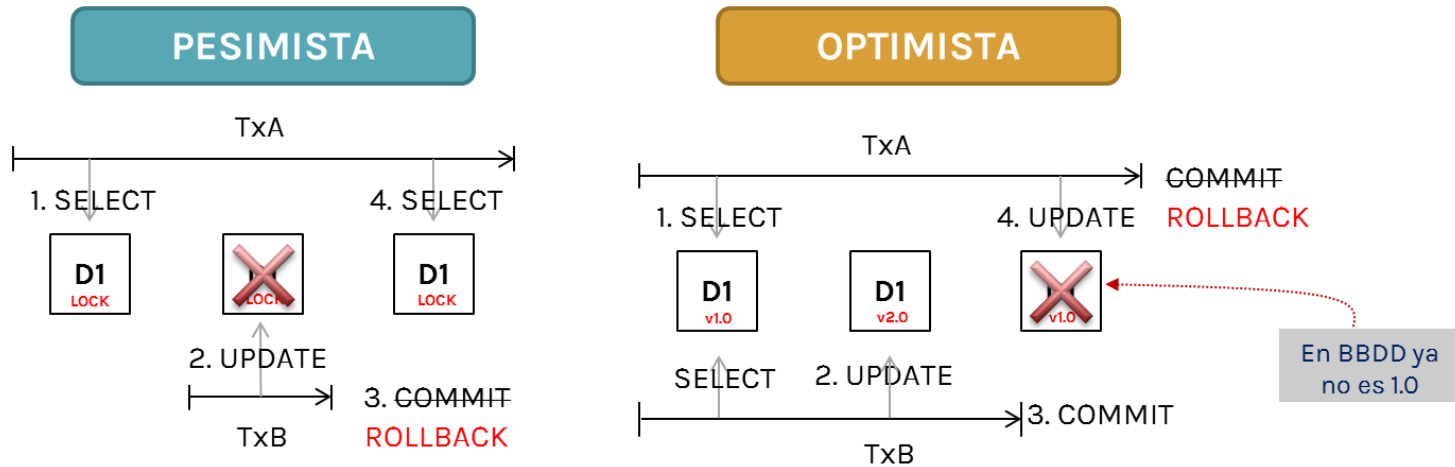


# MECANISMOS

DE CONTROL DE LA

# CONCURRENCIA

# OPTIMISTAS vs. PESIMISTAS

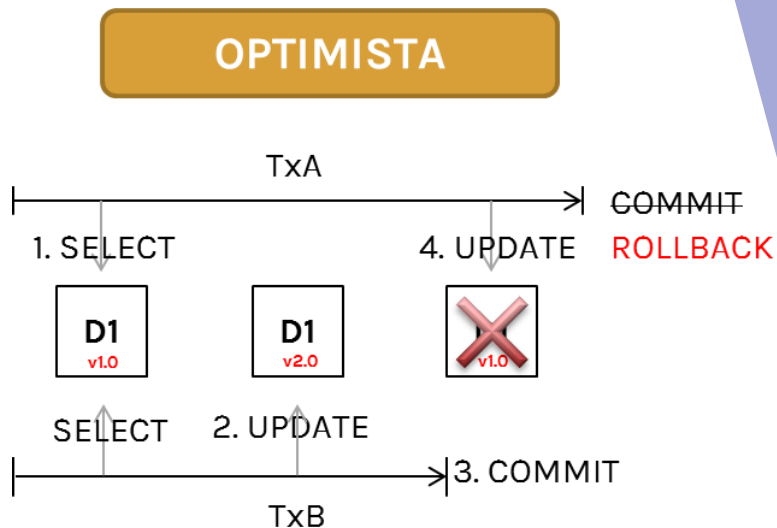


# BLOQUEO OPTIMISTA

JPA implementa el bloqueo optimista a partir de un campo de versión.

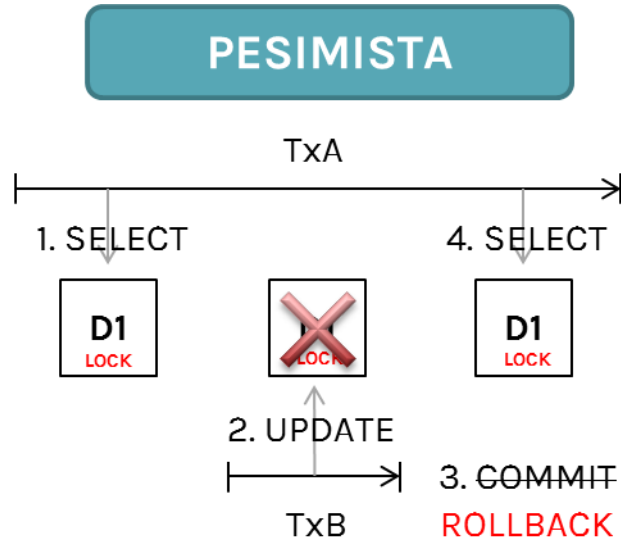
```
@Version  
private long version;
```

```
@Version  
private Date version;
```



# BLOQUEO PESIMISTA

Podemos delegar en la base de datos, o usar uno explícito.



# BLOQUEO PESIMISTA

- ▶ READ, OPTIMISTIC: se comprueba al final de la transacción.
- ▶ WRITE, OPTIMISTIC\_FORCE\_INCREMENT: la versión de la entidad se incrementa aunque la entidad no cambie.
- ▶ PESSIMISTIC\_FORCE\_INCREMENT: la entidad se bloquea y su versión incrementada aunque no cambie.
- ▶ PESSIMISTIC\_READ: la entidad es bloqueada de forma pesimista usando un bloqueo compartido.
- ▶ PESSIMISTIC\_WRITE: se utiliza un bloqueo explícito.