

Gestion des Transactions

Transaction et concurrence

Gestion optimiste / pessimiste

Transaction utilisateur

Configuration de coordinateur de transaction

Hibernate supporte les 2 mécanismes Java pour gérer les transactions :

- JDBC
- JTA

Par défaut :

- si JPA le coordinateur de transaction utilisé est celui indiqué avec le type de transaction pour l'unité de persistance.
- Si non-JPA: JDBC

Ce comportement par défaut peut être remplacé par la propriété :
`hibernate.transaction.coordinator_class`

Dans le cas de JTA et en particulier les serveurs *JavaEE*, l'implémentation de JTA est récupérée

- Soit automatiquement, en utilisant JNDI :
lookup sur *TransactionManager* et *UserTransaction*
- Soit via la propriété *hibernate.transaction.jta.platform*

Transaction API

Hibernate fournit une API permettant de manipuler les transactions :

- ***begin, commit, rollback*** : Délimitation, validation, annulation
- ***markRollbackOnly*** : Marque une transaction pour le rollback
- ***getTimeout, setTimeout*** : Timeout de la transaction
- ***registerSynchronization*** : Enregistrer une méthode de callback pour la transaction
- ***getStatus*** : Récupérer le statut

Dans un contexte JTA, JavaEE, c'est généralement le serveur applicatif qui s'occupe de délimiter les transactions

Notion de Transaction

Objectifs

- Garantir l'intégrité des données au fil des modifications de ces dernières.
- Exemple : Transfert de fond du Compte A vers le Compte B
 - Modifications :
 - 1 : Debiter(CpteA, somme)
 - 2 : Crediter(CpteB, somme)
 - Intégrité
 - $\text{Solde}(\text{CpteA}) + \text{Solde}(\text{CpteB})$ est invariant

Notion de Transaction

Définitions

Une Transaction est une unité logique de travail sur les données qui respecte les propriétés suivantes :

- Atomicité
 - L'ensemble des modifications de la transaction aboutit ou aucune n'aboutit.
- Cohérence
 - La transaction amène les données d'un état cohérent à un autre état cohérent.
- Isolation
 - Les résultats d'une transaction ne sont pas visibles des autres transactions avant la fin de celle-ci
- Durabilité
 - Tous les résultats d'une transaction aboutie sont persistants (survivent à n'importe quel crash).

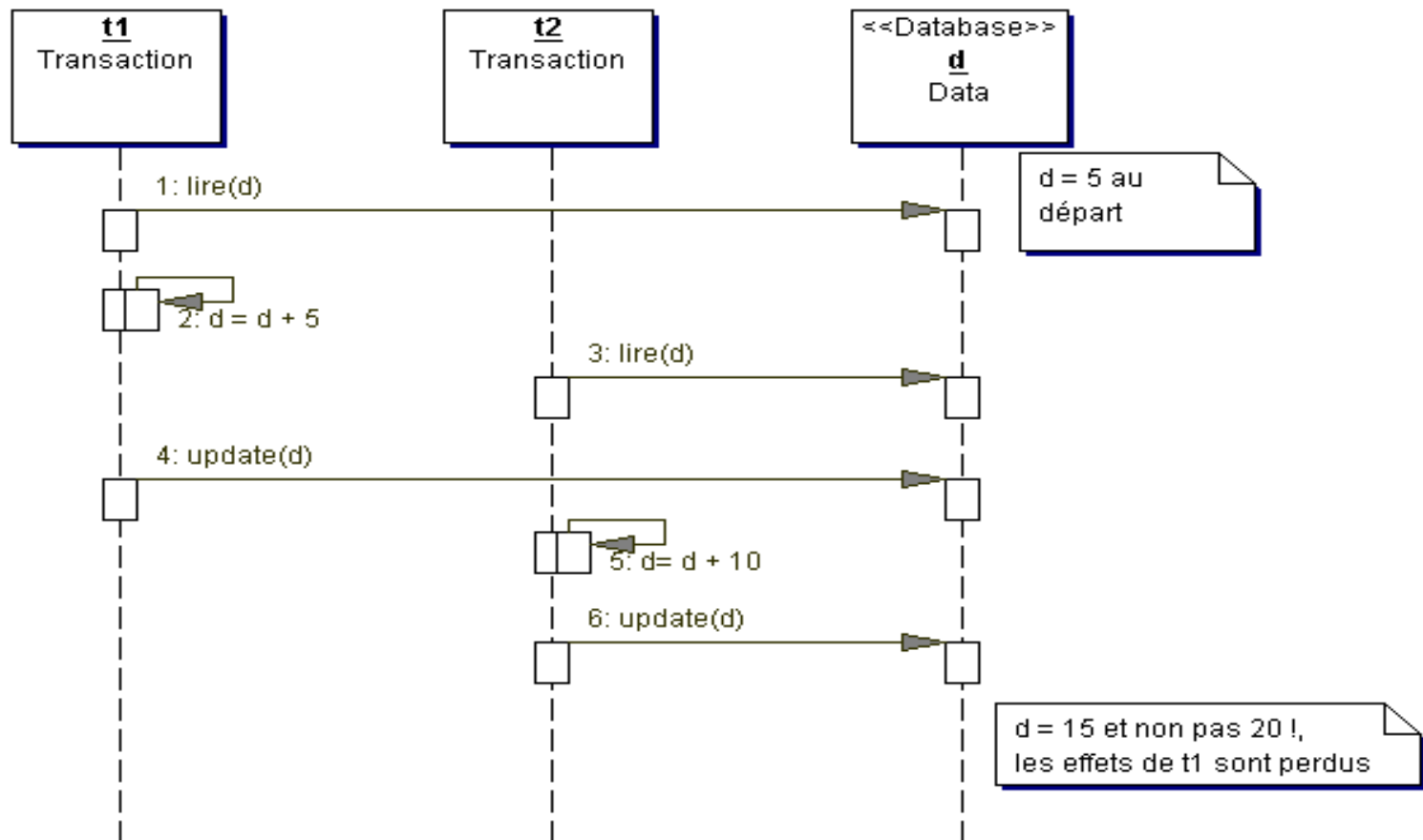
Transaction et Concurrency

Problèmes

- Problèmes de concurrence
 - Surviennent quand plusieurs processus ou threads veulent accéder aux mêmes données.
 - 3 problèmes de concurrence
 - Perte de mise à jour (dépendance écriture – écriture)
 - Dépendance non-validées
 - Analyse incohérente (Lecture-Ecriture)

Transaction et Concurrency

Perte de mise à jour



Niveaux d'isolation

Définitions

- Les BD proposent des verrous (R/W) pour contrer les problèmes liés à la concurrence
- Ils proposent des niveaux d'isolation qui permettent de faire un compromis entre la réactivité du système et les risques encourus
- 4 Niveaux sont définis
 - SERIALIZABLE
 - REPEATABLE-READ
 - READ-COMMITTED
 - READ-UNCOMMITTED

Niveaux d'isolation

Risques encourus

| Niveau Isolation / erreurs de lecture | Dirty Read | Unrepeatable Read | Phantom |
|---------------------------------------|------------|-------------------|---------|
| Read Uncommitted | O | O | O |
| Read Committed | N | O | O |
| Repeatable Read | N | N | O |
| Serializable | N | N | N |

Niveaux d'isolation

Lequel choisir ?

- Read-uncommitted
 - NON car trop dangereux
- Serializable
 - NON car trop lent et ne supporte pas la montée en charge
- Repeatable-Read
 - La session hibernate procure le même service.
- Read-committed
 - Le plus adapté (et le défaut)

```
<property name=« hibernate.connection.isolation »>2</property>
```

Hibernate et le contrôle de concurrence

Hibernate utilise directement les connexions JDBC et les ressources JTA sans ajouter de mécanisme de verrouillage particulier

A travers la session, qui est un cache dont la durée de vie est la transaction, Hibernate fournit un service de « repeatable reads »

Hibernate fournit alors 2 mécanismes pour le contrôle de concurrence :

- Versionning et stratégie optimiste

- Verrouillage stricte et stratégie pessimiste

Gestion des Transactions

Transaction et concurrence

Gestion optimiste / pessimiste

Transaction utilisateur

Stratégie de contrôle de la concurrence

Définitions

- Optimistic-locking
 - Pose des verrous déportée lors de la validation.
 - Utilisation de time-stamp ou versionning.
 - On résout les conflits lors de la validation des changements (commit)
- Pessimistic-locking
 - Pose des verrous immédiate.
 - On empêche les conflits

Stratégie de contrôle de la concurrence

Comparaison des stratégies

- Pessimistic-locking
 - Conflits fréquents et dangereux pour le système.
 - Réactivité peu importante
 - Transactions sur des domaines de données bien séparés
 - Résolution de conflits facile à automatiser
- Optimistic-locking
 - Conflits rares et peu dangereux pour le système
 - Réactivité importante
 - Résolution de conflits à la charge de l'utilisateur car difficile à automatiser.

Gestion Optimiste

Principe de versionnement

- Ajout d'une colonne technique
 - Version : Entier incrémenté à chaque modification de l'enregistrement
 - Timestamp : date de la dernière modification.
- Chaque mise à jour contient la restriction
WHERE version = X
- Si un conflit est détecté, l'utilisateur est mis au courant.

Gestion Optimiste

Mise en place dans Hibernate

- Mise en place pour la classe Theme
 - Modifier la table en ajoutant une colonne
 - Ajouter un attribut à la classe
 - Déclarer le versionnement avec le tag version

```
<class name="Theme" table="TTheme">
  <id name="id" column="id" access="field">
    <generator class="native"/>
  </id>
  <version name="version" column="version"/>
  <property name="label" column="label"/>
  ...
</class>
```

```
public class Theme {
    private Long id;
    private String label;
    private int version;
    ....
}
```


Gestion Optimiste

JPA

- Mise en place pour la classe Theme
 - Modifier la table en ajoutant une colonne
 - Ajouter un attribut à la classe
 - Annoter l'attribut avec *@Version*

```
@Entity
public class Theme {
    @Id @GeneratedValue
    private Long id;
    private String label ;
    @Version
    private int version;
    ....
}
```

Gestion Optimiste Usage

```
Session s1 = DBHelper.getFactory().openSession();
Session s2 = DBHelper.getFactory().openSession();
Transaction tx1 = s1.beginTransaction();
Transaction tx2 = s2.beginTransaction();
Theme info1 = (Theme)s1.get(Theme.class, new Long(3));
Theme info2 = (Theme)s2.get(Theme.class, new Long(3));
System.out.println("égalité en base ? :"+(info1.getId().equals(info2.getId())));
info1.setLabel("INFORMATIQUE");
info2.setLabel("INFO");
tx2.commit();
tx1.commit();
```



égalité en base ? :true

Hibernate: update TTheme set version=?, label=? where id=? and version=?

Hibernate: update TTheme set version=?, label=? where id=? and version=?

SEVERE: Could not synchronize database state with session

org.hibernate.StaleObjectStateException: Row was updated or deleted by another transaction

(or unsaved-value mapping was incorrect): [com.tsystems.etechno.j12.exemple.metier.Theme#3]

Gestion pessimiste

Pose de verrous explicites

- session
 - get()
 - load()
 - refresh()

```
Session s = DBHelper.getFactory().openSession();  
Transaction tx = s.beginTransaction();  
Theme info = (Theme)s.get(Theme.class, new Long(3),LockMode.UPGRADE);  
info.setLabel("INFORMATIQUE");  
tx1.commit();
```

Gestion pessimiste

types de verrous

- LockMode.NONE
 - Ne pas chercher en DB sauf si pas en cache.
 - C'est le défaut.
- LockMode.READ
 - Effectue une vérification de version.
- LockMode.UPGRADE
 - Effectue une vérification de version et obtient un verrou upgrade de la BD.
- LockMode.UPGRADE_NOWAIT
 - Idem mais propre à ORACLE qui supprime les attentes pour les transactions concurrentes.

Gestion des Transactions

Transaction et concurrence
Gestion optimiste / pessimiste
Transaction utilisateur

Transaction utilisateur

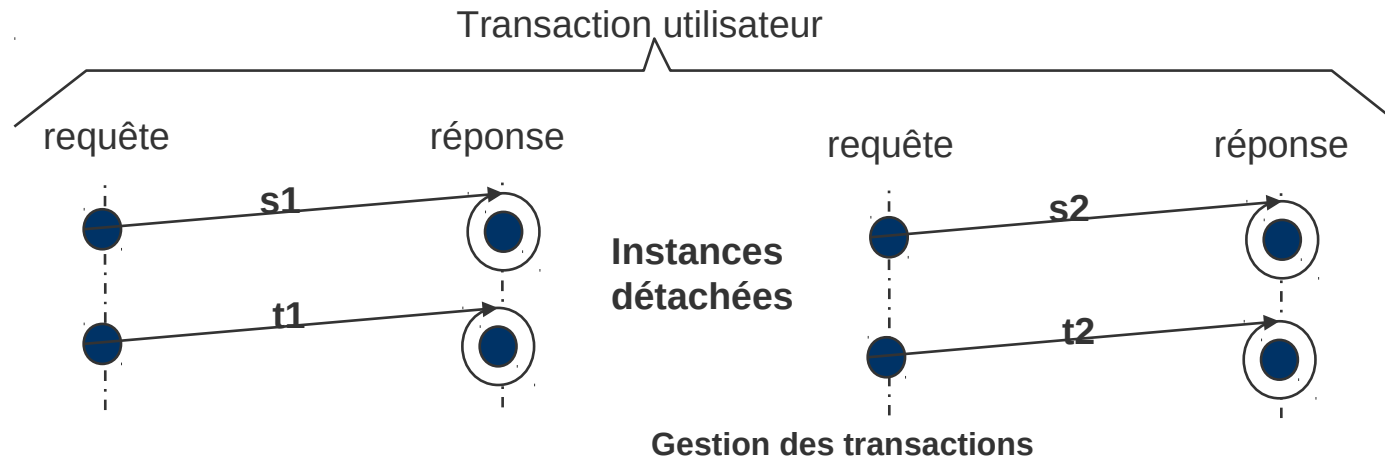
Définitions

- Transaction base de données
 - Transaction gérée par la base de données.
- Transaction utilisateur
 - Interaction homme-machine.
 - Proche de la notion de cas d'utilisation.
 - Ne peut bloquer des ressources.
 - C'est une transaction du point de vue de l'utilisateur.
 - On appelle également cela une conversation

Transaction utilisateur

une session par transaction BD

- Instances détachées
 - 1 : on obtient les objets persistants par une session.
 - 2 : les objets sont détachés pendant la manipulation utilisateur
 - 3 : on réattache les objets dans une autre session pour mise à jour



Exemple

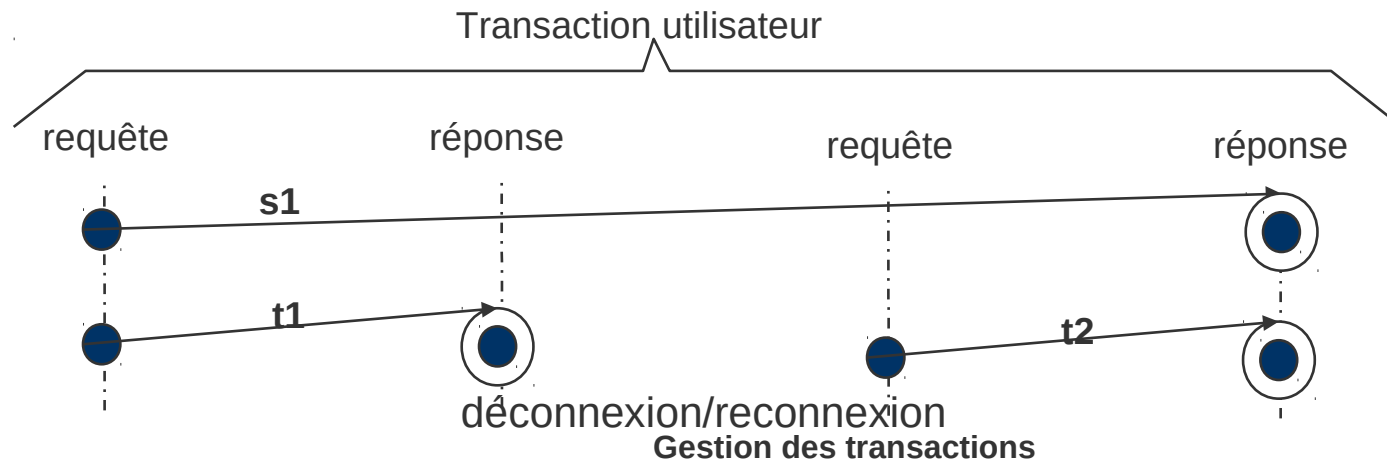
```
public class UseCaseManager {
    MyEntity myEntity;
    public void request1(long id) {
        Session sess = factory.openSession() ; Transaction tx = null;
        try {
            tx = sess.beginTransaction();
            this.myEntity = sess.load(MyEntity.class,id)
            tx.commit(); // Flush de la session et Mise à jour BD
        } catch (RuntimeException e) {      if (tx != null) tx.rollback() ;  throw e;
        } finally { sess.close() ; }
    }

    public void request2(String newValue) {
        Session sess = factory.openSession() ; Transaction tx = null;
        try {
            tx = sess.beginTransaction();
            myEntity = sess.load(MyEntity.class,myEntity.getId()); // Réattachement
            myEntity.setValue(newValue);
            tx.commit(); // Flush de la session et Mise à jour BD
        } catch (RuntimeException e) {
            if (tx != null) tx.rollback() ; throw e; // or display error message
        } finally { sess.close() ; }
    }
}
```


Transaction utilisateur

une session par transaction utilisateur

- La session est ouverte avec un FlushMode.MANUAL
- La session n'est fermée qu'à la dernière transaction BD
- `org.hibernate.Transaction.commit()` : Retourne la connexion JDBC dans le pool
- Les objets restent attachés à la session
- La dernière requête appelle explicitement `session.flush()`
- Une stratégie optimiste de contrôle de concurrence peut être appliquée



Exemple Session Hibernate

```
public class UseCaseManager {
    MyEntity myEntity ; Session sess;

    public void request1(long id, String newValue1) {
        Session sess = factory.openSession() ; Transaction tx = null ;
        sess.setFlushMode(FlushMode.MANUAL);
        try {
            tx = sess.getTransaction() ; tx.begin();
            this.myEntity = sess.load(MyEntity.class,id)
            myEntity.setValue1(newValue1) ;
            tx.commit(); // Relâche la connexion JDBC
        } catch (RuntimeException e) { if (tx != null) tx.rollback() ; throw e;}
        // On ne ferme pas la session
    }

    // Peut lancer une OptimisticLockException si versionning
    public void request2(String newValue2) {
        try {
            tx = sess.beginTransaction() ; // Obtient une nouvelle connexion JDBC
            myEntity.setValue2(newValue2) ; // Pas besoin de réattacher myEntity
            sess.flush();
            tx.commit();
        } catch (RuntimeException e) { if (tx != null) tx.rollback() ; throw e;
        } finally { sess.close() ; } } }
```

JPA, contexte de persistance étendu

JPA définit deux types de contexte de persistance:

- ✓ **transaction** (*transaction-scoped persistence context*)
 - les entités sont attachées à l'Entity Manager le temps d'exécution d'une transaction (en pratique celui d'une méthode)
 - ils deviennent détachés à la fin de la transaction
 - Mode par défaut
- ✓ **étendu** (*extended persistence context*)
 - les entités attachées à l'Entity Manager le restent même après la fin d'une transaction
 - Attention, le flush est quand même effectué lors de la fin de la transaction
 - Typiquement fait pour les EJB session stateful de JavaEE

Exemple JPA

```
public class UseCaseManager {
    @PersistenceContext(type = PersistenceContextType.EXTENDED)
    MyEntity myEntity ; EntityManager em;

    public void request1(long id, String newValue1) {
        try {
            // Récupération d'un contexte de persistance étendu (Java SE)
            em = DBHelper.getFactory().createEntityManager();
            // Pas d'ouverture de transaction !
            this.myEntity = em.find(MyEntity.class,id);
            myEntity.setValue1(newValue1) ;
            // Pas de flush en base car pas de transaction
        } catch (RuntimeException e) { throw e;}
    }

    // Peut lancer une OptimisticLockException si versionning
    public void request2(String newValue2) {
        try {
            tx = em.getTransaction().begin() ; // Obtient une nouvelle connexion JDBC
            // myEntity est déjà attaché
            myEntity.setValue2(newValue2) ;
            tx.commit(); // Flush vers la base car fermeture de transaction
        } catch (RuntimeException e) { if (tx != null) tx.rollback() ; throw e;
        } finally { em.close() ; }
    }
}
```