Spring et les transactions

GLG 203/Architectures Logicielles Java

Serge Rosmorduc
serge.rosmorduc@lecnam.net
Conservatoire National des Arts et Métiers

2019–2020

Démonstrations

https://gitlab.cnam.fr/gitlab/glg203_204_demos/11_spring_transactions.git

Les Transactions

Définition

Une **Transaction** est un traitement *atomique* (non sécable).
Quand une transaction est composée de plusieurs instructions qui ont des effets de bord :

- soit elles sont toutes exécutées ;
- soit aucune d'entre elle n'est prise en compte.

- on rencontre le plus souvent les transactions dans le monde des bases de données ;
- mais la notion peut s'étendre ailleurs : systèmes de messagerie par exemple.
- les transactions peuvent être **locales** : concerner une seule ressource
- ou **distribuées** : concerner **simultanément** plusieurs ressources : plusieurs BD, une BD et un système de messagerie, etc.

Pourquoi les transactions

Pseudo-code pour un virement :

```
x ← select solde from compte where id=1
y ← select solde from compte where id=2
update compte set solde= y + 100 where id= 2
update compte set solde= x - 100 where id= 1
```

En cas d'échec (déconnexion, plantage...) base de donnée dans un état incohérent

ACID

On souhaite disposer des propriétés suivantes :

Atomicity : les modifications sont toutes effectuées, ou aucune ne l'est

Consistence : si les invariants de la base sont vérifiés avant une transaction, que le code dans cette transaction (pris seul) garantit qu'ils sont vérifiés, alors ils sont effectivement vérifiés après transaction

Isolation : la transaction « voit » la base indépendamment de l'exécution des autres transactions

Durability : si le commit est fait, on garantit que la transaction est bien enregistrée même en cas de panne (journaux de log)

En réalité, **Atomicity** et **Durability** sont généralement vérifiée. Pour les autres, c'est plus compliqué.

Implémentation

Multiples mécanismes :

- utilisations de journaux : les opérations sont enregistrées avant d'être exécutées
  - garantit la reprise en cas de crash
  - permet le rollback (l'opération n'est pas encore enregistrée dans la base)
- utilisation de verrous

Programmation

Transactions en SQL (simplifié)

- Mode autocommit : par défaut chaque requête SQL constitue en elle-même une transaction.
- `SET AUTOCOMMIT FALSE;`
  - sort de ce mode (non transactionnel)
  - démarre la première transaction

Transactions en SQL (2)

COMMIT valide la transaction en cours (et commence une nouvelle transaction) les modifications effectuées sur la base deviennent permanentes

ROLLBACK annule la transaction en cours ; toutes les modifications depuis le dernier commit sont annulées ;

JDBC

Rappel

Il faut toujours fermer les connexions (et statements, etc...) ouverts.

- utiliser `try ... finally ...` (même sans catch !)
- ou mieux, utiliser « try with ressources » (JDK 1.7+)

```
String url= "jdbc:postgresql://localhost/guest";
Connexion db= DriverManager.getConnection(url, "guest", "toto");
Statement st= null;
try {
  try {
    st= db.createStatement();
    st.executeUpdate(...);
  } finally {
    if (st!= null) st.close();
  }
} finally {
  db.close();
}
```

Try-with-resources (java 7)

```
String url= "jdbc:postgresql://localhost/guest";
try (
  Connexion db= DriverManager.getConnection(
                    url, "guest", "toto");
  Statement st= db.createStatement();
) {
  db.executeUpdate(...);
}
```

- les ressources créées entre les « () » du try doivent être `Closeable` ;
- pour *toutes* ces ressources, la méthode `close()` est appelée **quoiqu'il arrive** à la fin du bloc try ...

Transactions jdbc

- Par défaut (specs) : mode auto-commit
- Pour passer en mode transactionnel :

```
connexion.setAutoCommit(false)
```

- Puis, selon les cas :

```
connexion.commit();
```

ou

```
connexion.rollback();
```

```
void virer(Connexion db, long idCompte1, long idCompte2, int somme) {
    db.setAutoCommit(false);
    try (
        PreparedStatement requete= db.prepareStatement(
           "select SOLDE from COMPTE where ID= ?");
        PreparedStatement pst= db.prepareStatement(
           "update COMPTE set SOLDE=? where ID= ?");
    ) {
        int solde1=0, solde2=0;
        requete.setLong(1, idCompte1);
        try (ResultSet r= requete.executeQuery()) {
            r.next(); solde1= r.getInt(1);
        }
        requete.setLong(1, idCompte2);
        try (ResultSet r= requete.executeQuery()) {
            r.next(); solde2= r.getInt(1);
        }
        pst.setLong(2,idCompte1); pst.setInt(1, solde1 - somme);
        pst.executeUpdate();
        pst.setLong(2,idCompte2); pst.setInt(1, solde2 + somme);
        pst.executeUpdate();
        db.commit();
    } catch (Exception e) {
        db.rollback();
        throw new RuntimeException(e);
    }
}
```

Transactions en Spring

Transactions et architecture

- Les transactions sont un mécanisme associé à la couche de persistance ;
- mais leur délimitation relève de la logique métier...
- comment résoudre ce problème ?

Transactions et architecture

- Les transactions sont un mécanisme associé à la couche de persistance ;
- mais leur délimitation relève de la logique métier...
- comment résoudre ce problème ?

- notation déclarative des transactions ;
- à n'importe quel niveau — mais souvent pour les services ;
- déclaration par
  - XML ;
  - annotations
- gestion explicite possible.

Mise en place (Spring et JPA)

- En springboot : la gestion des transaction est fournie « gratuitement » ;
- en Spring avec JPA
  - annoter une des classes de configuration avec `@EnableTransactionManagement`
  - définir :
    - une datasource dont les données seront tirées ;
    - un `entityManagerFactory` pour JPA ;
    - un **TransactionManager** pour la gestion des transactions.
  - suffit pour le cas classique JPA (ou JDBC) + **une** Datasource ;
  - ou pour des datasources indépendantes
- possibilité d'avoir des transactions portant sur **plusieurs** sources : transactions distribuées ;

Spring et import

Attention à un problème fréquent en Spring : plusieurs classes ont le même nom $\rightarrow$ faire attention aux imports.
Préférer les classes importées depuis Spring.

Configuration sans Springboot

00_spring-sans-springboot

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(basePackages = { "glg203.jpa01.dao" })
public class DatabaseConfig {
  @Bean
  public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder();
               .setType(EmbeddedDatabaseType.H2).build();
  }

  @Bean
  public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    HibernateJpaVendorAdapter vendorAdapter =
                                          new HibernateJpaVendorAdapter();
    vendorAdapter.setGenerateDdl(true);
    LocalContainerEntityManagerFactoryBean factory =
                             new LocalContainerEntityManagerFactoryBean();
    factory.setJpaVendorAdapter(vendorAdapter);
    factory.setPackagesToScan("glg203.jpa01.model");
    factory.setDataSource(dataSource());
    return factory;
  }

  // Suite au transparent suivant
```

Configuration sans Springboot (suite)

```
  /**
   * Trèèès important : le transaction manager, l'objet qui va gérer les t
   */
  @Bean
  public PlatformTransactionManager transactionManager() {
    JpaTransactionManager txManager = new JpaTransactionManager();
    txManager.setEntityManagerFactory(entityManagerFactory()
                                                    .getObject());
    return txManager;
  }

}
```

Classes et transaction

- Par défaut, les méthodes des repositories (`CrudRepository`, `PagingAndSortingRepository`, `JpaRepository` sont transactionnelles ;
- on peut rendre transactionnelle toutes les méthodes d'une classe en l'annotant avec `@Transactional`
- typiquement, on le fait pour les **services** ;
- on peut aussi annoter individuellement les méthodes.

Importer **org.springframework.transaction.annotation.Transactional** et non `javax.transaction.Transactional`

Exemple

```
@Service
@Transactional
public class PersonneService {
    @Autowired PersonneRepository repository;

    @Transactional(readOnly = true)
    public List<Personne> findAll() {
        return repository.findAll();
    }

    public Long creerPersonne(String nom, String prenom) {
        Personne p = new Personne(nom, prenom);
        repository.save(p);
        return p.getId();
    }

    public void importerPersonneFile(Reader r) {
        PersonneReader pReader = new PersonneReader(r);
        Personne p;
        while ((p = pReader.readPersonne())!= null) {
            repository.save(p);
        }
    }
}
```

Transactions et exceptions

- Par défaut, une méthode transactionnelle qui se termine normalement est validée par un `COMMIT` ;
- une méthode qui échoue suite à une `RuntimeException` est invalidée par un `ROLLBACK` ;
- ça n'est pas le cas par défaut pour les exceptions déclarées : si elles surviennent, les requêtes exécutées sont quand même validées...
- sauf si on le demande explicitement.

Exemple

La méthode :

```
public void importerPersonneFile(Reader r) {
  PersonneReader pReader = new PersonneReader(r);
  Personne p;
  while ((p = pReader.readPersonne())!= null) {
    repository.save(p);
   }
}
```

lit des descriptions de personnes dans un flux texte.

- elle peut échouer à moment donné si le flux contient des erreurs ;
- comme la classe `PersonneService` est transactionnelle et que l'échec sera visiblement marqué par une `RuntimeException`, les personnes déjà lues dans le fichier ne seront pas validées (`ROLLBACK`) ;
- ainsi, on corrigera le fichier, et on le relira ;
- sinon, on aurait eu une importation partielle du fichier.

L'annotation @Transactional

a les attributs :

- timeout : durée maximale de la transaction ;
- transactionManager : *qualifier* permettant de choisir un transactionManager spécifique quand plusieurs beans transactionManager sont définis ;
- readOnly : booléen ; dit si la transaction modifie ou non les données (optimisations possibles si true) ;
- rollbackFor, rollbackForClassName : liste de classes d'exceptions dont la levée doit déclencher un ROLLBACK ;
- noRollbackFor, noRollbackForClassName : liste de classes d'exceptions dont la levée ne doit pas déclencher un ROLLBACK ;
- isolation : isolement de la transaction (voir plus loin) ;
- propagation : propagation de la transaction (voir plus loin) ;

rollbackForClassName et **noRollbackForClassName** a comme valeur une String s qui est une *sous-chaîne* du nom des exceptions concernées.

- en cas d'ambiguïté, préférer utiliser comme valeur la notation longue (package + nom) ;
- ou **rollbackFor** et **noRollbackFor** ;
- pour les noms d'exceptions assez précis et non ambigus, pas de problème ;
- prend en compte l'héritage :

```
@Transactional(rollbackForClassName="IOException")
```

déclenchera un ROLLBACK pour *tous* les types de `IOException` (`FileNotFoundException` par exemple).

Exemple

```
@Service
@Transactional(
    rollbackFor = {CompteException.class})
public class BanqueService {
    @Autowired BanqueRepository repository;

    @Transactional(readOnly = true)
    public List<Compte> getComptes() {
        return repository.findAll();
    }

    public void virer(Long idSource, Long idDest, somme)
                                        throws CompteException {
        Compte c1 = repository1.findById(idSource).get();
        Compte c2 = repository2.findById(idDest).get();
        c1.virerVers(c2, somme); // exemple typique de transaction !
    }
}
```

Gestion manuelle des transactions

On peut gérer « à la main » l'exécution des transactions grâce à `TransactionTemplate`

- typiquement sous forme de bean injecté ;
- variante du pattern « template method » ;
- le code à exécuter est passé (typiquement sous forme de lambda) à l'objet TransactionTemplate, qui :
  - l'exécute dans un contexte transactionnel ;
  - permet de manipuler explicitement la transaction.

Exemple

```
public class Main {
 @Autowired TransactionTemplate transactionTemplate;
 @Autowired private CompteRepository repository;

 public void retirer(Long id, int somme) {
    transactionTemplate.executeWithoutResult(e -> {
      Compte c = repository.findById(id).get();
      int v = c.getSolde();
      c.setSolde(v - 50);
    });
 }
```

- `executeWithoutResult` prend comme argument un `Consumer<TransactionStatus>` :
- en gros, une lambda qui retourne void et prend comme argument un objet `TransactionStatus` ;
- le code de cette lambda est exécuté dans une transaction.

La classe TransactionTemplate

Méthodes :

- **executeWithoutResult** : à appeler quand la transaction ne retourne pas de résultat ;
- attend une lambda qui retourne void et prend comme argument un `TransactionStatus` ;
- **execute** : attend une fonction qui prend comme argument un `TransactionStatus` et retourne une valeur d'un type quelconque.
- **setIsolationLevel** : permet de fixer le niveau d'isolement à utiliser ; les valeurs sont définies dans la classe `TransactionDefinition` ;
- **setPropagationBehavior** règle la propagation de la transaction ;
- **setReadOnly** : permet de déclarer que la transaction ne modifiera pas les données.

Note : comme avec `@Transactional`, les `RuntimeException` déclenchent automatiquement des Rollback.

TransactionStatus

Cet objet, passé aux lambdas utilisées par les transactions, sert à consulter et à modifier l'état de la transaction en cours.
Notamment :

- setRollbackOnly() : cette méthode déclenche un rollback !
- flush() : force en gros un flush() sur l'entitymanager ;

Exemple

(On suppose que CompteException est une exception déclarée, pas une RuntimeException)

```
public class Main {
 @Autowired TransactionTemplate transactionTemplate;
 @Autowired private CompteRepository repository;

 public void retirer(Long id, int somme) {
    transactionTemplate.executeWithoutResult(status -> {
      try {
        Compte c = repository.findById(id).get();
        int v = c.getSolde();
        c.setSolde(v - 50);
      } catch (CompteException exception) {
        status.setRollbackOnly(); // Force le ROLLBACK
      }
    });
 }
```

Isolement des transaction

- Le mécanisme transactionnel décrit permet l'annulation/la validation d'une série d'opération
- théoriquement, chaque transaction doit « voir » la base comme si elle était la seule à s'exécuter.

Mais que se passe-t-il si des modifications sont effectuées en parallèle dans d'autres transactions ?

- dirty read ;
- non-repeatable read ;
- phantom read.

Dirty read

Définition

la transaction t1 lit des données fixées par une autre transaction t2 *qui ne sont pas encore validées par commit* En cas de rollback de t2, t1 a lu des données incorrectes.

Exemple

s1 <- solde compte1	
set solde compte1 = s1 + 1000	
	s2 <- solde compte1
ROLLBACK	
	set solde compte1 = s2 +2000
	COMMIT

- solde initial = 0
- Solde attendu : 2000
- Résultat : 3000

Non-repeatable read

dans une transaction, on fait :

```
v1 = lire entrée 1
.../* code qui ne modifie pas entrée 1 */
v2 = lire entrée 1
```

- une **autre** transaction a modifié l'entrée 1 ;
- et on obtient $v1 \neq v2$; comportement non prévisible

Exemple

le compte 1 a 100 euros

s1 <- solde compte1	s2 <- solde compte1
si s1 > 50 alors	set solde compte1 = s2 -75
	COMMIT
s1 <- solde compte1	
set solde compte 1 = s1 - 50	
fin si	
COMMIT	

la lecture de s1 donne deux résultats différents

Phantom read

Définition

le même select, effectué plusieurs fois dans une transaction, donne des résultats différents cause : une autre transaction a, par exemple, ajouté des lignes à la table.

Isolement des transactions (transaction isolation)

Plusieurs niveaux en SQL :

Read uncommitted : permet les « dirty read ». En gros, pas d'isolement

Read committed : empêche les dirty reads, peut permettre les non-repeatable reads

Repeatable reads : empêche les non-repeatable reads, peut permettre les phantom-reads

Serializable : isolement parfait, tout se passe comme si les transaction s'effectuaient de manière séquentielle

Implémentation

On utilise typiquement des verrous :

- sur une ligne de table ou sur un ensemble de lignes
- verrous en lecture : plusieurs verrous en lecture sur la même donnée sont possible
- verrous en écriture : sur la même donnée, on ne peut avoir qu'un seul verrou en écriture. Un verrou en écriture n'autorise pas non plus de verrou en lecture
- une transaction qui ne peut obtenir de verrou est mise en attente
- risque potentiel : **interblocage**

Read uncommitted

- pas de verrouillage du tout
- pas ACID pour deux ronds

Read committed

- Pour les **écriture,** un verrou sur la donnée écrite est pris et conservé **jusqu'à la fin de la transaction**
- Pour les **lecture,** on prend un **verrou en lecture,** mais il est relâché dès que la lecture est faite
- le verrou en lecture protège dans le cas où une autre transaction T2 écrit. Mais dès que T2 se termine, les verrous sont disponibles

Repeatable read

- On garde des verrous en lecture et écriture sur toutes les données lues ou écrites par la transaction jusqu'à la fin de la transaction
- **Seules les lignes concernées sont verrouillées** ;
- donc, de nouvelles lignes, créées par d'autres transactions, ne sont pas verrouillées
- ... et seront visible si une requête les retourne (possibilité de phantom read)

Serializable

- **plus coûteux en temps/ressources** ;
- On utilise des verrous plus puissants :
- dans l'idéal, on verrouille toutes les lignes potentiellement affectées par un `where` :
- si on a une clause du type « where age $>$ 10 », on verrouille toutes les lignes qui ont cette propriétés
- certaines bases verrouillent simplement toute la table :
  - moins de concurrence
  - plus simple et plus rapide à implémenter
  - (mais peut ralentir les clients, qui attendent plus)

À Propos de l'interblocage

- Géré par les SGBD
- Plusieurs solutions. Soient deux transactions A et B
- on détecte le blocage lorsqu'il se produit,
- on annule l'une des deux transactions, par exemple B
- on termine A
- quand A est terminée, on rejoue B (grâce au journal de transaction), et on la termine ;
- certaines bases peuvent tout simplement lever une exception.

Isolement des transactions en JDBC

Deux méthodes intéressantes :

- Sur DatabaseMetaData : meta.supportsTransactionIsolationLevel(LEVEL) permet de savoir si un niveau donné est supporté
- Sur Connection : db.setTransactionIsolation(LEVEL) permet de fixer un niveau
- les niveaux sont des constantes de l'interface Connection

- le mécanisme dépend de ce que sait faire la base de données SQL ;
- lire sa documentation, ça varie beaucoup !

En Spring

L'isolement se configure sur l'annotation `Transaction` ;

- rester de préférence cohérent ;
- Valeurs possibles :
  - `Isolation.DEFAULT` : la valeur dépend en fait de la base de données ;
  - `Isolation.READ_UNCOMMITTED`
  - `Isolation.READ_COMMITTED`
  - `Isolation.REPEATABLE_READ`
  - `Isolation.SERIALIZABLE`

Démonstration

Le projet `demo-isolation` montre les différents problèmes rencontrés et la manière dont l'isolement permet de les régler.

Il comporte plusieurs petits programmes :

- adirtyRead : démo de Dirty Read ; rectifiable grâce à `Isolation.READ_COMMITTED` ;
- bNonRepeatableRead : démo de NON REPEATABLE READ ; rectifiable grâce à `Isolation.REPEATABLE_READ` ;
- cphantomRead : démo de PHANTOM READ ; rectifiable grâce à `Isolation.SERIALIZABLE`.

Propagation des transaction

Dans un cadre transactionnel, une méthode :

- peut être appelée en dehors d'une transaction ;
- peut être appelée alors qu'une transaction est déjà en cours ;
- peut participer à une transaction A alors qu'une transaction B existe...

Pour préciser le comportement des méthodes, l'annotation `@Transactional` fourni l'attribut `propagation`

Valeurs possibles de propagation

PROPAGATION_REQUIRED crée une transaction si nécessaire, utilise la transaction courante sinon (valeur par défaut) ;

PROPAGATION_SUPPORTS se déroule dans la transaction actuelle si elle existe ;

PROPAGATION_MANDATORY ne peut s'exécuter que si une transaction existe ;

PROPAGATION_REQUIRES_NEW s'exécute dans une nouvelle transaction, même s'il y en a une en cours ;

PROPAGATION_NOT_SUPPORTED suspend la transaction actuelle si elle existe, et s'exécute en dehors de celle-ci ;

PROPAGATION_NEVER lève une exception si une transaction existe ;

PROPAGATION_NESTED transaction imbriquée (son commit n'est définitif que si le commit de la transaction parente est effectué) ;

Support des propagations

`PROPAGATION_REQUIRES_NEW`, `PROPAGATION_NOT_SUPPORTED` et `PROPAGATION_NESTED` ne sont pas toujours disponibles.
Voir leur javadoc dans Spring.

Exemple

```
@Component @SessionScope @Transactional
public class Panier {
 @Autowired ProduitRepository repository;

 private List<ProduitDTO> produits = new ArrayList<ProduitDTO>();

 // pas besoin de transaction pour cette méthode !
 //  → évite d'en créer inutilement
 @Transactional(propagation=Propagation.PROPAGATION_SUPPORTS)
 public void addProduit(ProduitDTO dto) {
  produits.add(dto);
 }

 // l'annotation ci-dessous est inutile (valeur par défaut)
 @Transactional(propagation=Propagation.PROPAGATION_REQUIRED)
 public void sauverCommande() {
  Commande c = CommandeHelper.construireCommande(produits);
  repository.save(c);
 }
 ...
```

Le verrouillage

- Complément ou substitut aux mécanismes d'isolement
- verrouillage pessimiste : permet de verrouiller de manière absolue des données pour être le seul à les manipuler ;
- verrouillage optimiste : permet de vérifier que des données qu'on veut écrire n'ont pas été modifiées entre-temps.

Verrou optimiste

- On n'utilise pas de verrou, mais simplement un **numéro de version** sur les objets ;
- Dans une entité, on déclare un champ de version :
- On a une exception lors du commit si l'entité a été modifiée par ailleurs.

```
@Entity
public class Compte {

 @Version
 private int version;
```

VERROUILLAGE OPTIMISTE

- Il faut éviter les modifications concurrentes et les lectures incohérentes à tout prix ;
- Solution sûre : verrouillage pessimiste ; *très coûteux ;*
- Généralement, *nbr(read)* $>>>$ *nbr(write)* ;
- pour une entrée, modifications concurrentes très rares ;

- but : sécuriser les opérations ;
- **échouer explicitement en cas de modification concurrente** ;
- la modification concurrente est alors annulée (rollback) ;
- on doit alors recharger les données.

Verrou Optimiste

idée : versionner les éléments

- Quand l'objet est sauvé, le numéro de version est vérifié ;
- s'il a changé, l'objet a été modifié par une autre transaction → `OptimisticLockException`
- Dans ce cas, prendre des mesures correctives

```
@Entity
public class Student {
 @Id
 private Long id;
 @Version
 private long version;
...
}
```

Verrou pessimiste en JPA

Méthodes de l'entity manager : `em.lock(entity,mode)` ;
mode :

PESSIMISTIC_READ : verrou en lecture ;

PESSIMISTIC_WRITE : verrou en écriture ;

NONE : relâche le verrou

Une même entité peut être verrouillée par plusieurs verrous en lecture, mais un verrou en écriture est exclusif de tout autre verrou.

Verrou pessimiste en Spring

- annotation `@Lock` ;
- **se pose sur une requête (dans un repository)** ;
- prend comme argument un LockModeType :

  OPTIMISTIC verrou optimise en lecture ;
  OPTIMISTIC_FORCE_INCREMENT verrou optimiste en écriture ;
  PESSIMISTIC_READ verrou pessimiste en lecture ;
  PESSIMISTIC_WRITE verrou pessimiste en écriture
  PESSIMISTIC_FORCE_INCREMENT verrou pessimiste en écriture, avec incrément du numéro de version ;
  NONE pas de verrou.

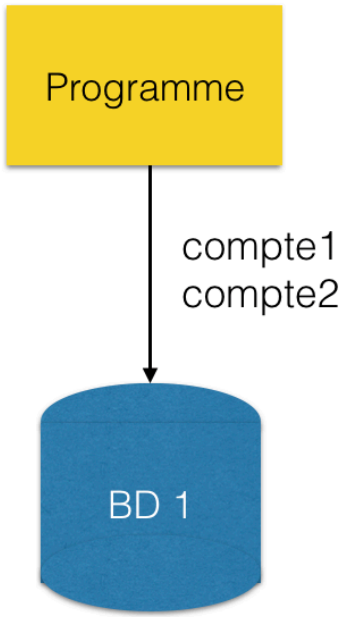
Verrou pessimiste en Spring

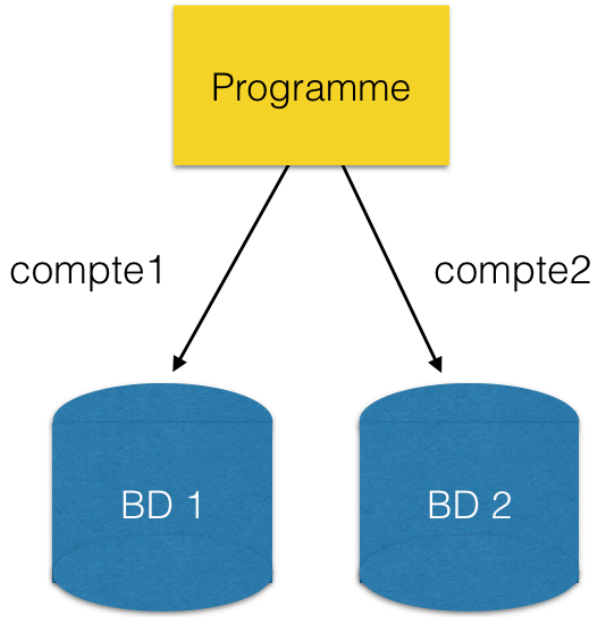
- Le verrou pessimiste porte sur les entités traitées par la requête ;
- une fois pris, il est relâché à la fin de la transaction ;
- exemple : projet `demo-lock` dans les exemples.

Quel verrou choisir ?

- verrou optimiste très efficace ;
- peu de cas d'erreur dans la plupart des « use cases ».
- mais nécessité de prévoir des OptimisticLockException
- verrou pessimiste coûteux
- mais sûr

Transactions locales et transactions globales





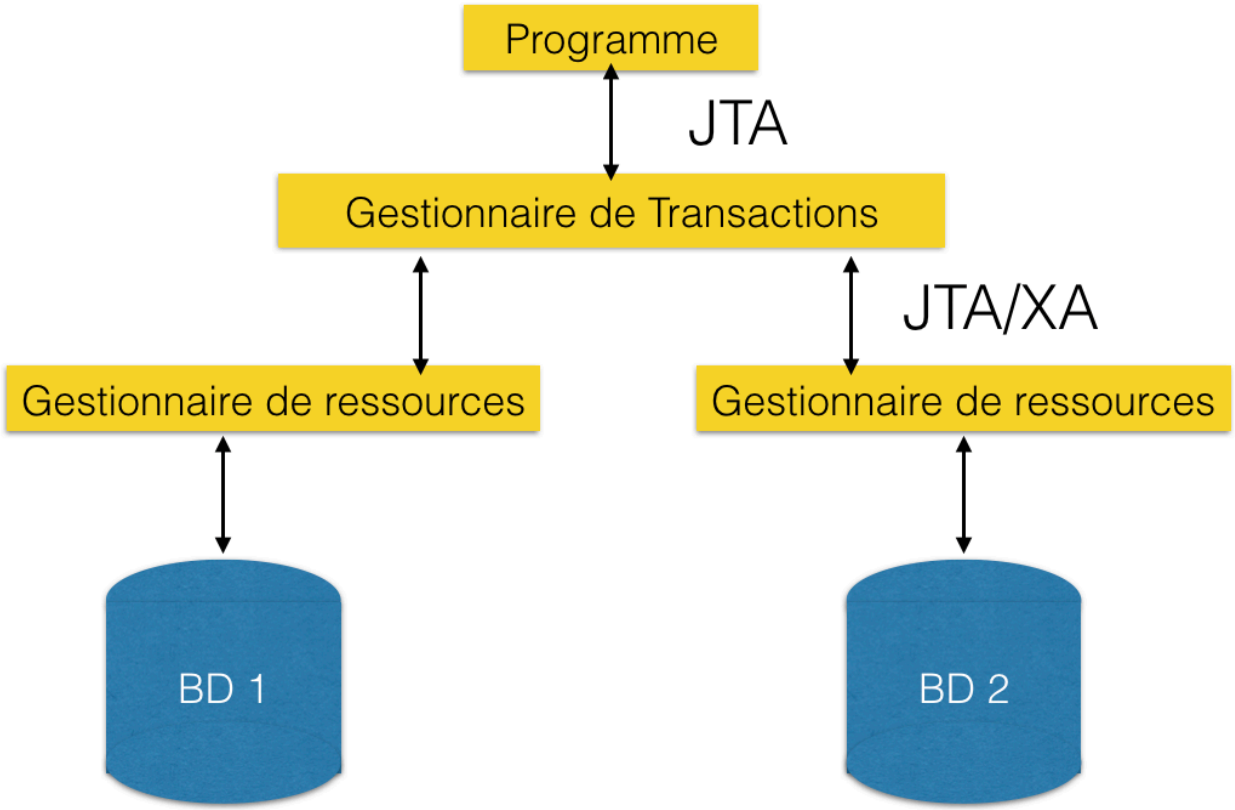
Transactions locales (plates) : sur une **une** seule ressource ;

Transactions globale : sur plusieurs ressources, éventuellement distribuées ; utilisation du « two-phase commit »

Transaction locale

- Modèle simple
- On démarre une transaction
- On exécute une série d'opérations
- On valide la transaction avec commit, ou on l'annule avec rollback
- en cas de plantage avant le commit, la transaction est annulée

Transactions globales



nécessitent un gestionnaire spécial, proposé par JTA dans les spécifications JEE.

Transactions globales

- prétendent garantir les propriétés ACID
- Utilisent le Two-Phase Commit :
  - Idée : avant le commit distribué, on demande à chacune des base si c'est ok
  - si ko : rollback de tout le monde
  - si ok : on demande à chacun de faire un com

Moniteur transactionnel

- Gère l'accès transactionnel aux bases de données
- support ACID des transactions
- partage de charge et de connexions
- reprise en cas de panne
- accès à des BD hétérogènes (NOSql, fichiers, XML...)
- API Standardisée

En Spring

Pas de support intégré des transactions globales...
Il faut utiliser une bibliothèque tierce : soit un serveur JEE, soit une bibliothèque spécialisée comme atomikos

Si on a plusieurs bases de données dans une application Spring :

- soit on les sépare complètement (gestionnaires de transactions indépendants), et tout fonctionne
- ... mais les opérations entre les bases ne sont pas transactionnelles ;
- soit on les utilise ensemble, dans la même transaction, et le gestionnaire de transaction doit gérer les transactions distribuées.
- exemple : projet `transactions-distribuees`

Présentation du projet « transactions-distribuees »

- dans ce projet, on gère deux banques ;
- chacune a sa base de données ;
- les transferts entre les deux banques doivent être transactionnels ;
- on utilise donc atomikos.

Architecture

- la banque 1 a son propre entityManager et son propre Repository
- la banque 2 a son propre entityManager et son propre Repository
- le transactionManager (fourni par Atomikos) est partagé ; il est configuré dans Spring par la classe TransactionConfig.
- le meilleur exemple de transaction partagée est dans `BanqueService` :

```
@Transactional(
    transactionManager = "transactionManager",
    rollbackFor = {CompteException.class})
@DependsOn("transactionManager")
public class BanqueService {
 @Autowired Banque1Repository repository1;
 @Autowired Banque2Repository repository2;
...
 public void virer(double somme) throws CompteException {
   Compte c1 =  repository1  .findById("1").get();
   Compte c2 =  repository2  .findById("2").get();
   c1.virerVers(c2, somme);
 }
```

Autres approches

Les concepteurs de Spring ne semblent pas très enthousiastes à propos des transactions globales.
Voir :
http://www.javaworld.com/article/2077963/open-source-tools/distributed-transactions-in-spring--with-and-without-xa.html pour des approches alternatives.

Message-Oriented Middleware

Définition

Message-Oriented Middleware : composant gérant la réception et la diffusion de messages entre divers éléments *logiciels*.

- permet de découpler l'envoi d'une requête de son traitement ;
- demande d'un traitement long, dont on ne veut pas attendre le résultat ;
- demande d'un traitement par un autre serveur ;
- envoi de mail ;
- diffusion d'un message à plusieurs destinataires ;
- les messages peuvent être point à point ou destinés à une liste de récepteurs ;
- les messages sont typiquement des *objets* ;
- spécification classique de MOM Java : JMS.

Déploiement

- Typiquement, le gestionnaire de message est un serveur auquel se connectent les émetteurs et les récepteurs de messages ;
- un même composant peut à la fois émettre des messages et en recevoir.
- le serveur s'occupe de supprimer les messages après un `commit` réussi, mais les conserve en cas de rollback, pour permettre un traitement ultérieur.

Message-Oriented Middleware et transactions

Le traitement des messages peut être transactionnel :

- réception/traitement puis émission de message ;
- couplage de traitement de message et d'action sur la base de données ;

Transaction JMS

- Une transaction JMS permet de grouper l'envoi de plusieurs messages ou la réception de plusieurs messages.
- Exemple d'envoi transactionnel :
  un client JMS envoie m1 et m2, mais ne souhaite pas qu'en cas de défaillance (du client) seul m1 soit expédié ;
- important : ne pas grouper un envoi de message **suivi** d'une réception dans la même transaction : blocage !!!

Exemple(s)

L'archive comporte deux projets :

demo-jmsserveur : logiciel qui intègre un serveur JMS activemq ; peut recevoir des messages.

demo-jmsclient : l'émetteur de messages.

Notez que les deux logiciels pourraient à la fois émettre et recevoir (par exemple : émission du résultat d'un traitement).

JMS : configuration

Serveur :

```
spring.activemq.broker-url=tcp://localhost:7171
spring.activemq.pool.enabled=true
spring.activemq.pool.max-connections=50
```

Client :

```
spring.activemq.packages.trustAll=true
spring.activemq.broker-url=failover:tcp://localhost:7171
spring.activemq.user=test
spring.activemq.password=test
logging.level.org.apache.activemq=info
```

JMS, configuration serveur

```
@Configuration @EnableJms
public class ServerConfig {
 // Valeur stockée dans application.properties
 @Value("${spring.activemq.broker-url}")
 String brokerUrl;


 // Configuration de la connexion
 @Bean
 public ActiveMQConnectionFactory amqConnectionFactory() {
  ActiveMQConnectionFactory factory =
                          new ActiveMQConnectionFactory(brokerUrl);
  factory.setPassword("test"); factory.setUserName("test");
  // Les objets qui pourront être sérialisés comme
  // contenu des messages sont ici :
  factory.setTrustedPackages(
                            Arrays.asList("glg203.demojms.model"));
  return factory;
 }
```

JMS, configuration serveur

```
 // Le gestionnaire de connexion (pourrait être externe)
 @Bean
 public BrokerService broker() throws Exception {
  BrokerService broker = new BrokerService();
  broker.addConnector(brokerUrl);
  broker.setPersistent(false); // pas de reprise en cas d'arrêt
  // Configuration de la connexion au serveur...
  AuthenticationUser user =
              new AuthenticationUser("test", "test", "");
  BrokerPlugin[] plugins = {
   new SimpleAuthenticationPlugin(Arrays.asList(user))
  };
  broker.setPlugins(plugins);
  // Fin de configuration
  return broker;
 }
}
```

Configuration du client

```
@Configuration
public class Config {
 String jmsUrl = "...";
 String login = "...";;
 String password = "...";

 // Le gestionnaire de transactions
 @Bean
 public TransactionManager transactionManager(ConnectionFactory cf) {
  return new JmsTransactionManager(cf);
 }

 // la connexion au serveur JMS.
 @Bean
 public ActiveMQConnectionFactory activeMQConnectionFactory() {
  ActiveMQConnectionFactory connectionFactory =
                  new ActiveMQConnectionFactory(login, password, jmsUrl);
  return connectionFactory;
 }
}
```

JMS : réception de message

Annotation `@JmsListener` sur un composant :

```
@Component
@Transactional
public class LoggerComponent {

 @JmsListener(destination = "log")
 public void receiveInfo(Message<LogInfo> message) {
  System.out.println("************************");
  System.out.println("* message reçu...");
  System.out.println("* ");
  System.out.println("* " + message.getHeaders());
  System.out.println("* " + message.getPayload());
  System.out.println("* ");
  System.out.println("************************");
 }
}
```

JMS : émission de message

```
@Service
@Transactional
public class MessageSenderService {
 private JmsTemplate jmsTemplate;

 @Autowired
 public MessageSenderService(ConnectionFactory connectionFactory) {
  jmsTemplate = new JmsTemplate(connectionFactory);
 }

 public void sendMessages(String message1, String message2) {
  jmsTemplate.setDeliveryMode(DeliveryMode.PERSISTENT);
  jmsTemplate.convertAndSend("log", new LogInfo("INFO", message1));
  if (StringUtils.isEmpty(message2)) {
   throw new RuntimeException("échec");
  }
   jmsTemplate.convertAndSend("log", new LogInfo("INFO", message2));
 }
```

- comme le mode est « persistent », si le serveur n'est pas disponible, on attend qu'il le soit ;
- en cas d'échec, rien n'est vraiment envoyé.

Attention !!

Comme souvent en Spring, et en particulier dans le cas présent, où de nombreux composants interviennent, le paramétrage est délicat.
Pour la pratique, je suggère de bien regarder les exemples.

Webographie

http://cedric.cnam.fr/~traversn/teaching/fip-BD/FIP_Concurrence.pdf http://www.subbu.org/articles/transactions/NutsAndBoltsOfTP.html
Armand Wilson « Distributed Transactions and Two-Phase Commit », SAP white paper (en particulier 2.2.8) http://lsrwww.epfl.ch/webdav/site/lsrwww/shared/Enseignement/SysRep07/Slides/JMS.pdf
http://camos.buniet.free.fr/LB/Distribue7.pdf
https://www.javaworld.com/article/2077963/distributed-transactions-in-spring--with-and-without-xa.html