

Transactions en Jakarta Enterprise Edition

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-14

Table des matières

1 Transactions	1	8.6 Implantation de <code>@Transactional</code> par CDI (interceptor)	6
2 Isolation des Transactions .	2	8.7 Qu'annoter avec <code>@Transactional?</code>	6
3 Transactions Locales vs Distribuées	2	8.8 <code>@TransactionalScoped</code> .	7
4 Gestion des Transactions en Java	2	9 Bonnes Pratiques 1. Gestion des Timeouts	7
5 EJB et CDI : Évolution des Composants Jakarta EE	2	9.1 Bonnes Pratiques 2. Granularité des Transactions	7
5.1 Migration EJB vers CDI	3	9.2 Bonnes Pratiques 3. Gestion des Erreurs . .	7
6 Les Transactions avec CDI .	3	9.3 Bonnes Pratiques 4. Optimisation Performance	8
6.1 Deux Approches de Gestion Transactionnelle	3	9.4 Bonnes Pratiques 5. Monitoring et Debug .	8
7 Transactions Gérées par le Bean (BMT)	3		
8 <code>@Transactional</code> Overview .	4		
8.1 Transactions Gérées par Conteneur (CMT)	4		
8.2 <code>@Transactional</code> Classe ou méthode	4		
8.3 Attributs de <code>@Transactional</code>	5		
8.4 Types de Transactions	5		
8.5 Exemple de Transaction Gérée par Conteneur	6		

1 Transactions

- **Transaction:** Unité de travail atomique
- **Propriétés ACID:**
 - **Atomicité:** Tout ou rien
 - **Cohérence:** État valide avant et après
 - **Isolation:** Indépendance des transactions
 - **Durabilité:** Changements persistants

2 Isolation des Transactions

• Problèmes d'Isolation

- **Dirty Read:** Lecture d'une donnée non validée (transaction non committée)

- **Non-Repeatable Read:** Deux lectures successives donnent des résultats différents

- **Phantom Read:** Nouvelles lignes apparaissent entre deux lectures

- **Lost Update:** Écrasement d'une mise à jour par une autre transaction

• Niveaux d'Isolation

READ_UNCOMMITTED - Plus bas niveau - Permet tous les problèmes - Performance maximale

READ_COMMITTED - Empêche dirty reads - Niveau par défaut - Bon compromis

REPEATABLE_READ - Empêche non-repeatable reads - Verrouillage lecture - Impact modéré

SERIALIZABLE - Isolation maximale - Empêche phantom reads - Performance réduite

3 Transactions Locales vs Distribuées

Transactions Locales

- S'exécutent sur une seule ressource
- Gérées par un seul gestionnaire de transactions
- Plus simples à implanter
- Exemple: Transaction sur une seule base de données

Transactions Distribuées

- Impliquent plusieurs ressources
- Nécessitent un protocole 2-Phase Commit
- Plus complexes mais plus flexibles
- Exemple: Transaction entre une DB ou plusieurs DB, une file de messages, ...

4 Gestion des Transactions en Java

• Local Transactions:

- Transactions sur une seule ressource
- JDBC, JPA, JMS, ...

• Java Transaction API (JTA):

- API standard pour la gestion des transactions
- Interface `UserTransaction`
- Interface `TransactionManager`

- Historiquement avec les Enterprise JavaBeans (EJB)
- Aujourd'hui avec CDI

5 EJB et CDI : Évolution des Composants Jakarta EE

Du EJB au CDI

1. Historique EJB

• Framework historique (1998) • Configuration par XML puis plus tard par annotations • Couplage fort • Tests unitaires difficiles	EJB	CDI	Commentaires
2. Évolution CDI (2009+)	<code>@Transactional</code>	<code>@Transactional</code>	sac-tions
• Configuration par annotations • Découplage facilité • Tests simplifiés	<code>@Scheduled</code>	<code>@Scheduled</code>	Timers

CDI et EJB peuvent coexister dans une application Jakarta EE

Remplacement progressif

1. **Services EJB**
 - Transactions (`@TransactionAttribute`)
 - Sécurité (`@RolesAllowed`)
 - Concurrency (`@Lock`)
 - Messaging (MDB)
2. **Appports CDI**
 - Contextes flexibles
 - Intercepteurs modulaires
 - Events/Observers
 - Qualifiers

5.1 Migration EJB vers CDI

EJB	CDI	Commentaires
<code>@Stateless</code>	<code>@ApplicationScoped</code>	Sans état
<code>@Stateful</code>	<code>@SessionScoped</code>	Avec état
<code>@Singleton</code>	<code>@ApplicationScoped</code> + <code>@Singleton</code>	Ins-tance unique
<code>@EJB</code>	<code>@Inject</code>	Injection

6 Les Transactions avec CDI

6.1 Deux Approches de Gestion Transactionnelle

Gestion Programmatique - Contrôle explicite par injection de `UserTransaction` - Style impératif - Granularité fine - Flexibilité maximale

Gestion Déclarative - Annotations `@Transactional` - Style déclaratif - Configuration simple - Gestion automatique

7 Transactions Gérées par le Bean (BMT)

- Contrôle programmatique explicite
- Démarrage et fin de transaction manuels où l'on veut
- Utilisation de l'interface `UserTransaction` par injection
- Plus flexible mais plus de responsabilité

```
public class MonBeanBMT
{
```

```

@Resource
UserTransaction utx;

    public void
maMethode() {
    try {
        utx.begin();
        // Code métier
        utx.commit();
    } catch (Exception
e) {
        try {
            utx.rollback();
        } catch
(SystemException se) {
            // Gestion
de l'erreur
        }
    }
}

```

8 @Transactional Overview

- Fournit une gestion déclarative des transactions
- Classe ou méthode annotées
- Alternative moderne aux annotations EJB

```

@Transactional(
    //Type de transaction
    //Les même que pour
les EJB

    value
= TxType.REQUIRED, //
Défaut

    // Déclenche
un rollback sur ces
exceptions

    rollbackOn =

```

```

{ SQLException.class },
    // Ne déclenche
pas de rollback sur ces
exceptions
    dontRollbackOn = { CustomException.class }
    // Par défaut,
toutes les exceptions
déclenchent un rollback
)
public class
CompteService {
    public void
transfer(long from, long
to, BigDecimal amount) {
        // Logique de
transfert
    }
}

```

8.1 Transactions Gérées par Conteneur (CMT)

- Gestion automatique par le conteneur Jakarta EE
- Configuration déclarative via annotations ou XML
- La transaction :
 - Commence au début d'une méthode publique
 - Peut être propagée à d'autres méthodes
 - Se termine à la fin de la méthode qui a commencé la transaction

8.2 @Transactional Classe ou méthode Niveau Classe

- S'applique à toutes les méthodes publiques
- Configuration par défaut pour la classe

- Peut être surchargée au niveau méthode
- Héritage automatique par sous-classes
- Configuration cohérente pour le service

Priorités d'Application

1. Annotation au niveau méthode
2. Annotation au niveau classe
3. Annotation sur interfaces
4. Configuration par défaut

Niveau Méthode

- Granularité fine du contrôle
- Surcharge des paramètres de classe
- Plus grande flexibilité
- Meilleure lisibilité du code

Considérations

- Éviter les mélanges excessifs
- Privilégier la cohérence classe
- Documenter les surcharges

8.3 Attributs de @Transactional

Attributs de Base

- **value** (TxType)
 - Définit le type de propagation
 - Détermine comportement avec transactions existantes
 - Par défaut: REQUIRED
- **timeout**
 - Durée maximale en secondes
 - 0 = pas de timeout
 - Dépend du conteneur

Gestion des Exceptions

- **rollbackOn**

- Liste des exceptions déclenchant rollback
- Surcharge le comportement par défaut
- Prioritaire sur dontRollbackOn

- **dontRollbackOn**

- Exceptions ne déclenchant pas de rollback
- Utile pour exceptions métier non critiques
- Secondaire par rapport à rollbackOn

8.4 Types de Transactions

- **REQUIRED** (défaut)

- Utilise transaction existante
- Crée nouvelle si aucune existe
- Usage: Opérations CRUD standard

- **REQUIRES_NEW**

- Crée toujours nouvelle transaction
- Suspend transaction existante
- Usage: Opérations indépendantes

- **SUPPORTS**

- Utilise transaction si existe
- Continue sans transaction sinon
- Usage: Opérations lecture seule

- **NOT_SUPPORTED**

- Suspend transaction existante
- S'exécute hors transaction
- Usage: Opérations non transactionnelles

- **MANDATORY**

- Exige transaction existante
- Exception si pas de transaction
- Usage: Opérations critiques
- **NEVER**
 - Erreur si transaction existe
 - S'exécute hors transaction
 - Usage: Services utilitaires

8.5 Exemple de Transaction Gérée par Conteneur

```
@RequestScoped
public class ComptBean
{
    @PersistenceContext
    private EntityManager
    em;

    @Transaction(TxType.REQUIRED)
    public
    void transferer(Long
    sourceId, Long destId,
    BigDecimal montant) {
        Compt source
        = em.find(Compt.class,
        sourceId);
        Compt dest
        = em.find(Compt.class,
        destId);

        if (source.getSolde().compareTo(montant)
        < 0) {
            throw new SoldeInsuffisantException();
        }

        source.setSolde(source.getSolde().subtract(montant));
        dest.setSolde(dest.getSolde().add(montant));

        em.merge(source);
        em.merge(dest);
    }
}
```

```
@Transaction(TxType.REQUIRES_NEW)
public
void journaliser(String
operation) {
    Journal journal
    = new Journal();
    journal.setOperation(operation);
    journal.setDate(LocalDateTime.now());
    em.persist(journal);
}
```

8.6 Implantation de @Transactional par CDI (interceptor)

1. Rôle de l'Intercepteur

- Intercepte appels méthodes annotées
- Gère cycle de vie transactionnel
- Wrapper transparent pour client
- Applique AOP (Aspect Oriented Programming)

2. Génération du Proxy

- Génération dynamique au démarrage
- Même interface que classe cible
- Injection transparente via CDI
- Chainage possible d'intercepteurs

8.7 Qu'annoter avec @Transactional?

- Opérations d'Écriture
 - Méthodes modifiant des données
 - @Transactional(REQUIRED)

- Opérations de Lecture
 - Méthodes consultation simple
 - Sans effet de bord
 - Performance privilégiée
 - `@Transactional(SUPPORTS)`

8.8 @TransactionalScoped

- Définit une portée liée au cycle de vie d'une transaction
- Bean créé au début de la transaction
- Détruit à la fin de la transaction

```
@TransactionalScoped
@Named
public
class TransactionBean
implements Serializable
{
    private static final
    long serialVersionUID =
    1L;

    private
    List<Operation>
    operations = new
    ArrayList<>();

    public void
    addOperation(Operation
    op) {
        operations.add(op);
    }

    // Le
    bean et ses données
    persistent pendant toute
    la transaction
}
```

9 Bonnes Pratiques

1. Gestion des Timeouts

- Définir des timeouts appropriés par type d'opération
- Monitorer les transactions longues
- Prévoir mécanisme de reprise sur timeout
- Éviter les timeouts trop courts ou trop longs

9.1 Bonnes Pratiques

2. Granularité des Transactions

- **Transaction Courte**
 - Durée minimale
 - Moins de conflits
 - Meilleure scalabilité
 - Préférer pour opérations CRUD simples
- **Transaction Longue**
 - À éviter si possible
 - Risque deadlocks
 - Impact performance
 - Uniquement si cohérence forte nécessaire

9.2 Bonnes Pratiques 3.

Gestion des Erreurs

- Différencier exceptions métier/technique
- Définir stratégie rollback claire
- Logger avant rollback
- Implémenter compensation si nécessaire

Note: `@TransactionalScoped` nécessite que les beans implémentent `Serializable`

9.3 Bonnes Pratiques 4. Optimisation Per- formance

- Éviter transactions imbriquées
- Minimiser le nombre de ressources
- Utiliser read-only quand possible
- Choix judicieux isolation level

9.4 Bonnes Pratiques 5. Monitoring et De- bug

- Activer logs transactionnels
- Surveiller métriques clés:
 - Durée moyenne
 - Taux rollback
 - Nombre deadlocks
 - Transactions actives
- JConsole, VisualVM, Application Server Console