

Java Persistence API (version 3.0 des EJB)

Robert Bourgeois

V.2370

Présentation de JPA

Ce document est la propriété exclusive de TELÉCOM. Il ne peut pas être utilisé sans autorisation.

Qu'est-ce que JPA ?

JPA est une API destinée à la persistance d'objets java (Entity Beans) dans une base de données relationnelle avec JDBC.

JPA fait partie de la spécification EJB 3 de JavaEE 5. Cette API est l'aboutissement des travaux autour de frameworks tels que Hibernate ou Toplink; elle vise à standardiser l'usage des frameworks de mapping objet / relationnel (ORM).

La transparence est un atout majeur de JPA car celle-ci utilise des objets Java ordinaires (POJO) avec des métadonnées standardisées. Ceci rend possible la réutilisation de composants qui n'ont pas été conçus pour être déployés avec un framework particulier.

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

3

Qu'est-ce que JPA ? (2)

Développée dans le cadre de la version 3.0 des EJB, cette API ne se limite pas aux EJB puisqu'elle peut aussi être mise en oeuvre dans des applications autonomes Java SE.

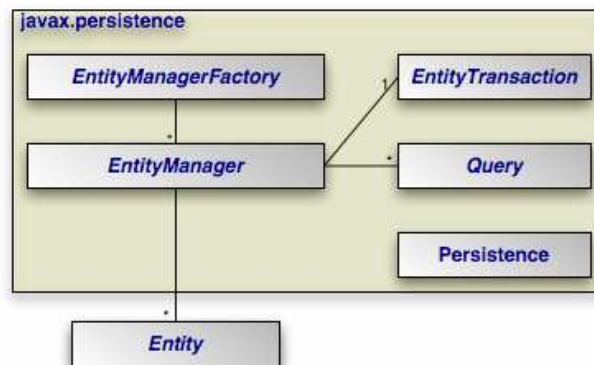
L'API propose un langage d'interrogation JPQL similaire à SQL mais utilisant des entités objets plutôt que des entités relationnelles de la base de données.

L'API Java Persistence repose sur le concept de gestionnaire d'entité (EntityManager) pour toute action de persistance.

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

4

Architecture de base de JPA



Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

5

Gestion et manipulation des Entity beans

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

6

Entity Manager

L'API EntityManager est utilisée pour accéder à une base de données et pour créer et manipuler des instances d'Entity Beans au sein d'un contexte de persistance (persistence context)

L'ensemble des classes d'Entity Beans accessibles à un EntityManager sont définies par une unité de persistance (persistence unit)

Un EntityManager peut être application-managed ou container-managed

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

7

Entity Beans

Un Entity Bean est un objet Java ordinaire avec des annotations définies dans le package javax.persistence.

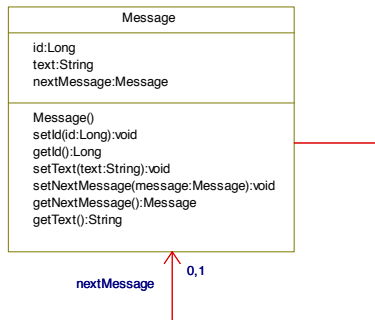
Une instance d'Entity Bean ne devient persistante en base de données que sur demande explicite faite à un EntityManager

Sauf ambiguïté, nous nommerons les objets Entity Beans des entités dans la suite du cours

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

8

Exemple d'Entity Bean



```
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;

@Entity
public class Message {

    @Id @GeneratedValue
    private Long id;

    private String text;

    @OneToOne(cascade = CascadeType.ALL)
    private Message nextMessage;

    Message() {}

    getters and setters ...
}
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

9

EntityManager application-managed

Ce type d'EntityManager est généralement utilisé dans les applications autonomes Java SE.

C'est le code applicatif qui se charge d'obtenir et de fermer explicitement l'EntityManager

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

10

Entity Manager application-managed (2)

L'EntityManager peut être invoqué dans une transaction JTA en cours (cas d'un JTA Entity Manager) ou c'est lui-même qui peut démarrer une transaction à l'aide de l'API EntityTransaction (cas d'un resource-local Entity Manager)

Dans le cas d'un resource-local EntityManager la transaction n'utilise pas JTA mais est directement liée à la ressource utilisée (par exemple une transaction JDBC)

Le type d'EntityManager (JTA ou resource-local) utilisé est défini au niveau de la configuration de l'EntityManagerFactory dans l'unité de persistance associée.

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

11

Entity Manager application-managed (3)

Généralement utilisé dans un environnement Java EE.

C'est le conteneur du serveur d'applications (conteneur d'EJB, Web, ou d'applications clientes) qui récupère habituellement l'EntityManager par injection de dépendances

L'EntityManager peut néanmoins être créé explicitement à partir d'une fabrique EntityManagerFactory

Un EntityManager container-managed requiert l'usage de transactions JTA

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

12

Contexte de persistance

Un contexte de persistance est un ensemble d'instances d'entités (une sorte de cache) rendues persistantes et gérées par un EntityManager

Un EntityManager représente un contexte de persistance, mais il arrive qu'un même contexte de persistance soit représenté et géré par plusieurs EntityManagers, à condition que ces EntityManagers aient été créés par la même fabrique (EntityManagerFactory)

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

13

Transaction-scoped persistence context

Le contexte de persistance s'initialise lorsque le code obtient un Entity Manager au sein d'une transaction déjà active et est détruit quand la transaction associée est validée ou annulée.

Quand le contexte est détruit, toutes les entités gérées sont détachées

Seuls les container-managed persistence contexts peuvent être transaction-scoped

Ceci revient à dire que seules les instances d'EntityManager injectées avec l'annotation @PersistenceContext ou son équivalent XML peuvent être transaction-scoped

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

14

Exemple d'injection d'un EntityManager

Dans un serveur EJB 3.0 un container-managed EntityManager est accessible par injection de dépendance

```
@Stateless
public class MessageBean implements MessageRemote {

    @PersistenceContext
    EntityManager em;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void saveMessages() {
        Message message = new Message("Hello World with EJB3");
        em.persist(message); // injection de l'EntityManager dans em
        Message nextMessage = new Message("Hi !");
        message.setNextMessage(nextMessage);
    }
}
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

15

Exemple d'injection d'un EntityManager (2)

Quand la méthode saveMessages() est invoquée, le conteneur d'EJB l'exécute dans le contexte d'une transaction JTA.

Une instance de Message est insérée dans le contexte de persistance. Cette instance restera gérée par l'EntityManager durant toute la durée de la transaction.

Ceci signifie que l'état de l'instance de Message sera enregistré en base de données quand la transaction sera terminée et validée. Le contexte de persistance est alors détruit et l'instance de Message est détachée

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

16

Extended persistence context

Un contexte de persistance peut aussi avoir une durée supérieure à une transaction.

Cette propriété est importante dans les situations où l'on veut maintenir une conversation avec la base de données sans utiliser une transaction trop longue (les transactions requièrent des ressources coûteuses comme les connexions JDBC et les locks en base de données)

Les contextes de persistance étendus peuvent être créés et gérés dans le code applicatif mais aussi par des EJB Session stateful

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

17

Entités détachées

Une entité gérée devient détachée lorsqu'un contexte de persistance est détruit.

Une conséquence intéressante de cet état est que l'entité détachée peut être sérialisée et transmise sur le réseau à un client distant.

Le client peut faire des modifications sur cette entité et la renvoyer au serveur pour être réattachée à un contexte de persistance et synchronisée avec la base de données.

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

18

Unité de persistance

Un EntityManager lie un ensemble de classes d'Entity Beans à une base de données. Cet ensemble est appelé une unité de persistance.

Les unités de persistance définies pour une application sont décrites dans un fichier nommé persistence.xml situé dans un répertoire META-INF :

- d'un composant jar accessible dans le classpath d'une application JavaSE
- d'un composant ear d'une application d'entreprise
- d'un module EJB (ejb.jar)
- d'une application Web (war)
- d'une application déployée dans un conteneur d'applications clientes (jar)

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

19

Unité de persistance (2)

Une unité de persistance définit et/ou configure :

- L'ensemble des classes d'Entity Beans concernées (ayant une image dans la base de données)
- Le fournisseur de persistance
(*provider – qui implémente javax.persistence.PersistenceProvider*)
- La source de données (*datasource ou connexion*)
- Le mode de gestion transactionnelle (JTA ou RESOURCE_LOCAL)
- Un ensemble de propriétés spécifiques au fournisseur de persistance

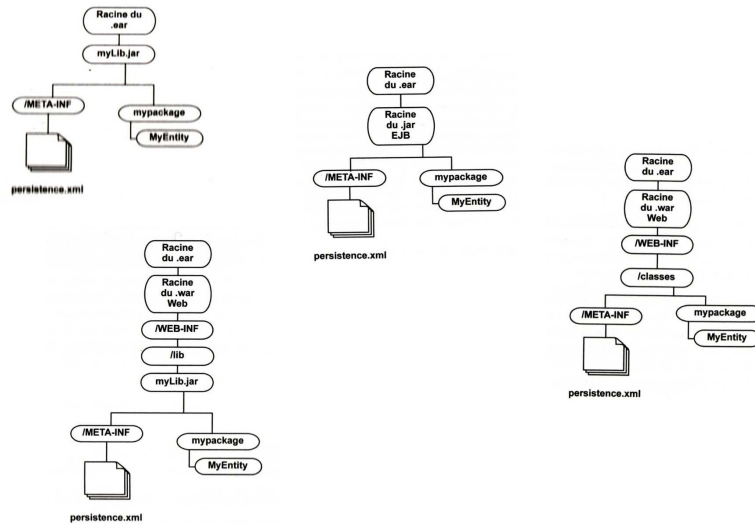
Par défaut, les classes d'Entity Beans (classes annotées @Entity), situées à la racine de l'archive jar contenant le fichier persistence.xml sont prises en compte (sauf si l'élément <exclude-unlisted-classes /> est présent)

Il est possible de spécifier des archives supplémentaires

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

20

Déploiement en environnement JavaEE



Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation.

21

Exemple pour une application JavaEE

```
<persistence 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_1_0.xsd"
version="1.0">

<persistence-unit name="pu" transaction-type="JTA">

<provider>org.hibernate.ejb.HibernatePersistence</provider>
<jta-data-source>java:jpaDS</jta-data-source>

<class>com.isnel.Message</class>

<properties>
<property name="hibernate.hbm2ddl.auto" value="create" />
<property name="hibernate.show_sql" value="false" />
<property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
<property name="hibernate.cache.provider_class"
value="org.hibernate.cache.NoCacheProvider" />
<property name="hibernate.transaction.manager_lookup_class"
value="org.hibernate.transaction.JBossTransactionManagerLookup" />
</properties>

</persistence-unit>
</persistence>
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation.

22

Déploiement en environnement JavaSE

Peu de différences existent en terme de packaging entre une application Java SE et une application Java EE. Les classes Entity Beans doivent se trouver dans un fichier jar et le fichier « persistence.xml » doit être placé dans le dossier « META-INF ».

Toutefois, aucune source de données (datasource) ne peut être définie au sein d'une application Java SE.

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

23

Exemple pour une application JavaSE

```
<persistence 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_1_0.xsd"
version="1.0">

<persistence-unit name="pu" transaction-type="RESOURCE_LOCAL">

<!-- <exclude-unlisted-classes /> -->

<properties>
<property name="hibernate.show_sql" value="false" />
<property name="hibernate.format_sql" value="false" />
<property name="use_sql_comments" value="false" />
<property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
<property name="hibernate.connection.driver_class" value="com.mysql.jdbc.Driver" />
<property name="hibernate.connection.username" value="root" />
<property name="hibernate.connection.password" value="rodejaphgh" />
<property name="hibernate.connection.url" value="jdbc:mysql://localhost:3306/jpadb" />
<property name="hibernate.cache.provider_class"
value="org.hibernate.cache.NoCacheProvider" />
</properties>
</persistence-unit>
</persistence>
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

24

Obtention d'un EntityManager

La seule façon possible de créer un EntityManager en JavaSE et de passer par une fabrique EntityManagerFactory

JavaEE donne la possibilité d'obtenir un EntityManager par injection de dépendance

Ce document est la propriété exclusive de TELECOMET. Il ne peut pas être utilisé sans autorisation

25

EntityManagerFactory en JavaSE

L'obtention d'une fabrique se fait par la méthode statique createEntityManagerFactory() de la classe Persistence
Cette méthode prend en paramètre le nom d'une unité de persistance ou une Map qui permet de surcharger les propriétés du fichier persistence.xml ou de les étendre.

Ce document est la propriété exclusive de TELECOMET. Il ne peut pas être utilisé sans autorisation

26

EntityManagerFactory en JavaEE

En JavaEE il est possible de récupérer une fabrique EntityManagerFactory par injection de dépendance

Il suffit de définir un attribut avec l'annotation @PersistenceUnit qui prend en attributs :

- le nom d'une unité de persistance configurée dans le fichier persistence.xml
- un nom qui permettra de retrouver si besoin est la fabrique via JNDI

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

27

Exemple

```
@Stateless
public class ManageAuctionBean implements ManageAuction {

    @PersistenceUnit(unitName = "auctionDB")
    EntityManagerFactory auctionDB;
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

28

Obtention d'un contexte de persistance

- ❑ Un contexte de persistance peut être obtenu à partir d'une fabrique EntityManagerFactory par l'opération createEntityManager () ou createEntityManager(Map properties) qui permet de surcharger les propriétés du fichier persistence.xml ou de les étendre
- ❑ L'EntityManager retourné est de type application-managed et représente un extended persistence context
- ❑ Si l'EntityManagerFactory est de type JTA, il faut explicitement lier l'EntityManager à la transaction courante par la méthode joinTransaction() car sinon les entités gérées ne seront pas synchronisées avec la base de données à la fin de la transaction.

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

29

Exemple

L'EntityManager ici obtenu à partir de la fabrique injectée dans auctionDB est application-managed, donc le conteneur ne se charge pas de synchroniser la base de données avec le contenu du contexte de persistance, ni ne ferme celui-ci.

Inversement, si la méthode joinTransaction () était appelée, la méthode close() serait sans effet.

```
@Stateless
public class ManageAuctionBean implements ManageAuction {
    @PersistenceUnit(unitName = "auctionDB")
    EntityManagerFactory auctionDB;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public Item findAuctionByName(String name) {
        EntityManager em = auctionDB.createEntityManager();
        //em.joinTransaction();
        ...
        em.flush(); em.close();
        return item;
    }
}
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

30

Obtention d'un contexte de persistance (2)

Dans le cas d'un container-managed EntityManager, celui-ci peut être injecté directement dans un EJB (ou une servlet) en utilisant l'annotation @PersistenceContext

L'attribut unitName permet de préciser l'unité de persistance à utiliser.

L'attribut type permet de spécifier le type de contexte de persistance injecté (PersistenceContextType.TRANSACTION par défaut)

Par défaut, un transaction-scoped persistent context est donc associé à la transaction courante et ceci jusqu'à ce qu'elle se termine

Ce document est la propriété exclusive de TELECOMET. Il ne peut pas être utilisé sans autorisation

31

Exemple

```
@Stateless
public class MessageBean implements MessageRemote {

    @PersistenceContext
    private EntityManager em;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void saveMessages() {
        Message message = new Message("Hello World with EJB3");
        em.persist(message);
        Message nextMessage = new Message("Hi !");
        message.setNextMessage(nextMessage);
    }
}
```

Ce document est la propriété exclusive de TELECOMET. Il ne peut pas être utilisé sans autorisation

32

Exemple (2)

Voici une variation qui utilise deux bases de données – c'est-à-dire deux unités de persistance

```
@Stateless
public class ManageAuctionBean implements ManageAuction {

    @PersistenceContext(unitName = "auctionDB")
    private EntityManager auctionEM;

    @PersistenceContext(unitName = "auditDB")
    private EntityManager auditEM;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void createAuction(String name, BigDecimal price) {
        Item newItem = new Item(name, price);
        auctionEM.persist(newItem);
        auditEM.persist( new CreateAuctionEvent(newItem) );
        ...
    }
}
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

33

Obtention d'un contexte de persistance (3)

Un EntityManager de type PersistenceContentType.EXTENDED n'a de sens que s'il est utilisé au sein d'un EJB Session stateful. En effet c'est le seul composant qui garde un état conversationnel avec le client entre deux appels de méthodes.

Les EJB Session stateless et les EJB Message Driven sont gérés dans un pool et il n'y aurait aucun moyen pour clore le contexte de persistance étendu.

Dans le cas d'un EJB Session stateful le contexte de persistance étendu est créé en même temps que l'EJB (avant l'appel de la méthode callback @PostConstruct) et fermé automatiquement à la suppression de l'instance d'EJB. Ceci signifie que toute entité du contexte de persistance reste attachée et gérée aussi longtemps que l'EJB est actif.

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

34

Exemple

```
@Stateful
public class MessageBean implements MessageRemote {

    @PersistenceContext(unitName= "pu", type = PersistenceContextType.EXTENDED)
    private EntityManager em;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void saveMessages() {
        Message message = new Message("Hello World with EJB3");
        em.persist(message);
        Message nextMessage = new Message("Hi !");
        message.setNextMessage(nextMessage);
    }
}
```

Ce document est la propriété exclusive de TELECOMET. Il ne peut pas être utilisé sans autorisation

35

Propagation du contexte de persistance

Ne concerne que les container-managed entity managers.

Pour un EntityManager container managed transaction-scoped (commun dans un environnement Java EE), la propagation du contexte de persistance est la même que celle de la transaction JTA. En d'autres mots, les EntityManagers container-managed transaction-scoped retrouvés dans une transition JTA donnée se partagent tous le même contexte de persistance.

A noter que les contextes de persistance ne sont jamais partagés entre EntityManagers qui ne proviennent pas de la même fabrique EntityManagerFactory.

Si un EJB (Session ou Message Driven) avec un contexte de persistance transaction-scoped appelle un EJB Session stateful avec un contexte étendu dans le même transaction JTA, une exception IllegalStateException is levée.

Ce document est la propriété exclusive de TELECOMET. Il ne peut pas être utilisé sans autorisation

36

Propagation du contexte de persistance (2)

Si un EJB Session stateful avec un contexte de persistance étendu

- appelle un autre EJB Session (stateful ou stateless) avec un contexte transaction-scoped dans la même transaction JTA, le contexte de persistance est propagé.
- appelle un autre EJB Session (stateful ou stateless) dans une autre transaction JTA, le contexte de persistance n'est pas propagé.
- instancie un autre EJB Session stateful avec un contexte de persistance étendu, le contexte de persistance est hérité par le second EJB
- appelle un autre EJB Session stateful avec un contexte de persistance étendu différent, dans la même transaction, une exception `IllegalStateException` est levée

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

37

Interagir avec un EntityManager

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

38

Cycle de vie d'un Entity Bean

Un Entity Bean peut prendre quatre états par rapport à un EntityManager :

- **new** (ou transient) : l'objet n'est associé à aucun contexte de persistance et n'a pas d'image en base de données.
- **managed** : l'objet est inclus dans le contexte de persistance, possède une image dans la base de données et ses modifications sont gérées par l'EntityManager.
- **detached** : l'objet possède une image dans la base de données mais il échappe au contrôle de l'EntityManager car le contexte de persistance d'où il provient a été vidé ou fermé.
- **removed** : l'objet est encore inclus dans un contexte de persistance mais est destiné à en être retiré ainsi que de la base de données .

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

39

Interface EntityManager

L'interface `javax.persistence.EntityManager` offre des opérations d'ajout, modification, suppression, recherche d'entités et un accès à l'API Query

- `clear()` : void - EntityManager
- `close()` : void - EntityManager
- `contains(Object arg0)` : boolean - EntityManager
- `createNamedQuery(String arg0)` : Query - EntityManager
- `createNativeQuery(String arg0)` : Query - EntityManager
- `createNativeQuery(String arg0, Class arg1)` : Query - EntityManager
- `createQuery(String arg0)` : Query - EntityManager
- `equals(Object obj)` : boolean - Object
- `find(Class<T> arg0, Object arg1)` : T - EntityManager
- `flush()` : void - EntityManager
- `getClass()` : Class<?> - Object
- `getDelegate()` : Object - EntityManager
- `getFlushMode()` : FlushModeType - EntityManager
- `getReference(Class<T> arg0, Object arg1)` : T - EntityManager
- `getTransaction()` : EntityTransaction - EntityManager
- `hashCode()` : int - Object
- `isOpen()` : boolean - EntityManager
- `joinTransaction()` : void - EntityManager
- `lock(Object arg0, LockModeType arg1)` : void - EntityManager
- `merge(T arg0)` : T - EntityManager
- `notify()` : void - Object
- `notifyAll()` : void - Object
- `persist(Object arg0)` : void - EntityManager
- `refresh(Object arg0)` : void - EntityManager
- `remove(Object arg0)` : void - EntityManager
- `setFlushMode(FlushModeType arg0)` : void - EntityManager

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

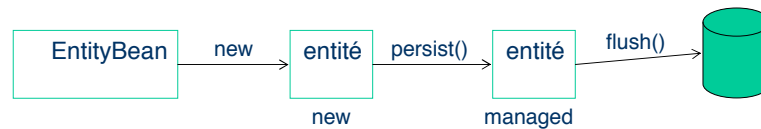
40

Enregistrer les entités

Pour enregistrer une entité en base de données il faut tout d'abord créer une instance *i* d'un `EntityBean` puis appeler la méthode `persist(i)` de l'`EntityManager`.

L'effet de cette méthode est de faire passer *i* dans l'état "géré" (managed), c'est-à-dire de l'insérer dans le contexte de persistance de l'`EntityManager`.

L'enregistrement en base de données peut être fait immédiatement ou retardé jusqu'à la terminaison de la transaction en fonction du mode de synchronisation choisi. Il est toujours possible de forcer l'enregistrement en base de données dans une transaction en appelant la méthode `flush()`.



41

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

Enregistrer les entités (2)

Il n'est possible d'appeler la méthode `persist(i)` en dehors d'une transaction que si le contexte de persistance est de type étendu.

Dans ce cas l'insertion est mise en file d'attente jusqu'à ce que le contexte soit associé à une transaction

Si une entité a des relations avec d'autres entités, ces entités liées peuvent aussi être enregistrées dans la base de données si le mode de cascade approprié a été spécifié.

42

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

Enregistrer les entités (3)

Si la méthode `persist()` est appelée avec comme argument un objet qui n'est pas un `EntityBean` déclaré dans l'unité de persistance, une exception `IllegalArgumentException` sera levée.

De même, si cette méthode est appelée avec un `transaction-scoped` persistent context et qu'aucune transaction n'est active, une exception `TransactionRequiredException` est levée.

Rappel : Lors de l'injection d'un contexte de persistance étendu celui-ci est automatiquement associé à une transaction. Dans le cas de l'utilisation manuelle d'une fabrique `EntityManagerFactory` il faut appeler la méthode `joinTransaction()` pour associer le contexte à la transaction courante.

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

43

Exemple

```
@Stateful
public class MessageBean implements MessageRemote {

    @PersistenceContext(unitName= "pu", type =
PersistenceContextType.EXTENDED)
    private EntityManager em;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void saveMessages() {
        Message message = new Message("Hello World with EJB3");
        em.persist(message);
        Message nextMessage = new Message("Hi !");
        message.setNextMessage(nextMessage);
    } // enregistrement des deux entités liées en base de données
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

44

Retrouver des entités

Les objets sauvegardés peuvent être retrouvés et insérés dans le contexte de persistance grâce à leur clé primaire ou par une requête JPQL.

L'EntityManager offre deux opérations pour retrouver des objets à partir de leur clé primaire :

- `<T> find(Class<T> entity, Object primaryKey)`
- `<T> getReference(Class<T> entity, Object primaryKey)`

La méthode `find()` retourne null si aucune entité n'est identifiée par la clé primaire alors que la méthode `getReference()` lève une exception `EntityNotFoundException`.

La méthode `getReference` ne garantit pas l'initialisation de l'objet récupéré

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

45

Exemple

```
EntityManager em = emf.createEntityManager();

Message c = em.find(Message.class, 1L);
System.out.println("Message found : " + c.getText());

Query q = em.createQuery("from Message m order by m.text asc");

List messages = q.getResultList();

System.out.println( messages.size() + " message(s) found:" );

for (Object m : messages) {
    Message loadedMsg = (Message) m;
    System.out.println(loadedMsg.getText());
}
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

46

Modifier les entités

Lorsqu'une entité est récupérée par `find()` ou `getReference()`, celle-ci est gérée par le contexte de persistance (`EntityManager`) jusqu'à ce qu'il soit fermé.

Il est possible de changer les propriétés de cette instance (sauf bien sûr l'id), les modifications seront synchronisées automatiquement

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

47

Ré-attacher une entité détachée

Une entité `i` détachée d'un contexte de persistance parce que celui-ci a été fermé peut-être ré-attachée dans un nouveau contexte avec l'opération `merge(i)`

Les modifications apportées à l'entité lorsqu'elle était détachée sont alors enregistrées en base de données au moment du flush du nouveau contexte

L'entité récupérée par `merge(i)` se retrouve dans le nouveau contexte de persistance mais c'est une copie de `i` si `i` n'existait pas déjà dans le contexte de persistance. Dans ce cas `i` reste dans l'état détaché

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

48

Exemple

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
Message message = new Message("Hello World with JPA");
em.persist(message);

Message a = em.find(Message.class, 1L);
Message b = em.find(Message.class, 1L);
System.out.println(a == b); // true
tx.commit();
em.close();

// New extended persistence context
EntityManager newEm = emf.createEntityManager();

Message c = newEm.find(Message.class, 1L);
Message d = newEm.merge(a);
System.out.println(a == c); // false
System.out.println(d == c); // true
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

49

Supprimer les entités

Une entité *i* peut être supprimée de la base de données en utilisant la méthode `remove(i)`

i devient détachée, mais la suppression de l'entité en base de données ne se fait qu'au moment de l'appel de la méthode `flush()` ou à la fermeture du contexte de persistance

Il est possible de ré-attacher une entité *i* détachée par la méthode `remove(i)` si le contexte de persistance n'a pas encore été fermé. Il faut alors appeler la méthode `persiste(i)`

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

50

Autres opérations de l'EntityManager

La méthode `refresh(i)` permet de resynchroniser une entité à partir de la base de données.

La méthode `lock(i)` permet de gérer la concurrence d'accès aux données (ce point sera étudié plus loin)

La méthode `contains(i)` permet de savoir si l'entité `i` est déjà présente dans le contexte de persistance

La méthode `clear()` permet de vider le contexte de persistance courant (les entités gérées deviennent détachées)

Ne pas oublier d'enregistrer les modifications des entités gérées avant d'appeler la méthode `clear()` !

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

51

Flush d'un contexte de persistance

Le flush d'un contexte de persistance (synchronisation de la base de données avec ce contexte) se fait lors de l'appel de la méthode `flush()` de l'Entity Manager ou du commit d'une transaction.

Attention ! `flush()` et `commit()` sont deux opérations différentes. `flush()` envoie les requêtes SQL à la base de données alors que `commit()` valide les transactions au niveau de la base de données.

JPA autorise ses implémentations à effectuer des synchronisations à d'autres moments.

Par exemple, Hibernate synchronise la base de données avec le contexte de persistance avant que toute requête ne soit effectuée.

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

52

Flush d'un contexte de persistance (2)

La synchronisation de la base de données avant toute requête peut poser des problèmes de performance.

La méthode `setFlushMode()` de l'API `EntityManager` permet de contrôler le flush du contexte de persistance.

Son argument de type `FlushModeType` peut prendre la valeur `AUTO` (valeur par défaut) ou `COMMIT`, ce qui a pour effet de supprimer le flush automatique avant les requêtes (plus performant mais optimiste)

Exemple avec un application-scoped `EntityManager` :

```
EntityManager em = emf.createEntityManager();
em.setFlushMode(FlushModeType.COMMIT);
EntityTransaction tx = em.getTransaction();
tx.begin();
...
tx.commit();
em.close();
```

53

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

Suppression des flushes automatiques

Un flush automatique du contexte de persistance étendu peut dans certains cas poser des problèmes dans une conversation entre un EJB Session Stateful et un client.

La solution préconisée par la spécification EJB 3.0 pour se prémunir contre les flushes automatiques pendant une conversation et de rendre toutes les étapes de la conversation non-transactionnelles, sauf la dernière.

C'est donc la dernière transaction qui synchronise la base de données au moment du commit.

Un conseil : écrivez vos objets DAO comme des EJB Session stateful si vous concevez votre contrôleur comme un EJB Session stateful !

54

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

Exemple

```
@Stateful
@Transactional(TransactionalAttributeType.NOT_SUPPORTED)
public class ManageAuctionBean implements ManageAuction {
    @PersistenceContext(type = PersistenceContextType.EXTENDED)
    EntityManager em;

    public Item getAuction(Long itemId) {
        return em.find(Item.class, itemId);
    }

    public boolean sellerHasEnoughMoney(User seller) {
        boolean sellerCanAffordIt = (Boolean)
            em.createQuery("select...").getSingleResult();
        return sellerCanAffordIt;
    }

    @Remove
    @Transactional(TransactionalAttributeType.REQUIRED)
    public void endAuction(Item item, User buyer) {
        // Set winning bid
        // Charge seller
        // Notify seller and winner
        item.setBuyer(...);
    }
}
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

55

Dirty checking automatique

Un ORM comme Hibernate ne synchronise pas chaque objet présent dans le contexte de persistance avec la base de données au moment d'un flush.

On appelle dirty checking la stratégie utilisée pour détecter quels objets ont été modifiés par l'application.

Une entité dont les modifications n'ont pas encore été propagées à la base de données est considéré comme dirty.

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

56

Méthodes du cycle de vie d'un Entity Bean

Les méthodes du cycle de vie d'un Entity Bean (*callbacks*) sont des méthodes appelées par le conteneur lorsque certains événements se produisent sur un Entity Bean

Il existe sept annotations associées aux différents événements du cycle de vie d'un Entity Bean :

- @PrePersist ou @PreRemove invoquées avant l'exécution des méthodes persist() et remove()
- @PostPersist ou @PostRemove invoquées après l'exécution des méthodes persist() et remove().
- @PreUpdate ou @PostUpdate invoquées avant et après la synchronisation de la base de données
- @PostLoad invoquée après chargement de l'entité dans le contexte de persistance ou après un refresh()

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

57

Méthodes "callback" (2)

Les méthodes "callback" annotées de l'Entity Bean doivent avoir le format void <METHOD> ()

On peut déclarer ces méthodes dans une classe annexe pour ne pas surcharger la classe de l'Entity Bean.

Cette classe doit alors être spécifiée par l'annotation @EntityListeners au niveau de la classe de l'Entity Bean

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

58

Exemple

```
public class MessageListener {

    @PostPersist
    public void alerte(Message m) {
        System.out.println("Clef du nouveau Message : " + m.getId());
    }
}

@Entity
@EntityListeners({MessageListener.class})
public class Message {

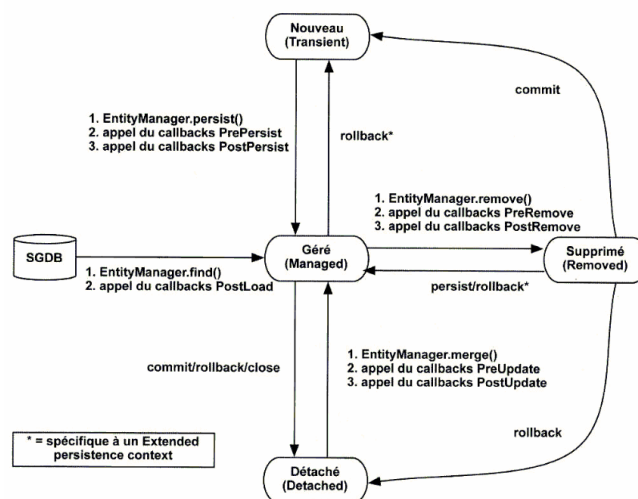
    @Id @GeneratedValue
    private Long id;

    ...
}
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation


59

Cycle de vie d'un Entity Bean



Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation


60



Ecriture des Entity Beans

Ce document est la propriété exclusive de TEL-ECOLET. Il ne peut pas être utilisé sans autorisation.

61



La classe d'un Entity Bean

Les Entity Beans sont des Java Beans marqués avec l'annotation `@Entity`

L'annotation `NamedQuery` permet d'associer une requête SQL à cet Entity Bean

L'état persistant de l'Entity Bean est représenté par ses variables d'instance (private ou protected) accessibles par des accesseurs

La classe peut être abstraite ou concrète et spécialiser une autre classe (Entity Bean ou non)

Ce document est la propriété exclusive de TEL-ECOLET. Il ne peut pas être utilisé sans autorisation.

62

La classe d'un Entity Bean (2)

La classe d'un Entity Bean doit cependant respecter certaines règles :

- La classe et ses propriétés ne peuvent être finales
- Si une instance peut être accédée de manière distante, sa classe doit implémenter Serializable
- La classe doit posséder au moins un constructeur sans arguments
- La classe doit posséder un champ qui servira de clé primaire.
- Les champs multi-valués doivent être déclarés comme des interfaces (Collection, Set, List ou Map)

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

63

ORM des Entity Beans

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

64

Mapping d'un Entity Bean

L'image d'un Entity Bean en base de données est généralement une seule table dont le nom est par défaut celui de l'Entity Bean

L'annotation (optionnelle) @Table permet de préciser certains paramètres

Attributs	Rôle
name	Nom de la table
catalog	Catalogue (BD)
schema	Schéma de la table
uniqueConstraints	Contraintes d'unicité sur une ou plusieurs colonnes

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

65

Identité des Entity Beans

Un Entity Bean a une identité en base de données (clé primaire) et a une existence propre; il peut exister indépendamment de toute autre entité.

Une référence à un Entity Bean est enregistrée comme telle dans la base de données (clé étrangère).

Les autres classes Java qui ne sont pas des Entity Beans (String, Integer, classes ordinaires, ...) n'ont pas d'identité en base de données

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

66

Identité et égalité

En Java, l'identité d'un objet est définie par sa classe et son adresse en mémoire. Deux variables ont la même valeur si elles référencent le même objet en mémoire. Ceci peut être testé avec l'opérateur ==

Deux objets sont égaux s'ils respectent une certaine contrainte définie par la méthode equals(Object o). Les classes qui ne surchargent pas cette méthode héritent de celle de la classe Object qui compare l'identité des objets référencés.

Deux objets stockés dans une base de données relationnelle sont identiques s'ils sont représentés par le même tuple, c'est-à-dire s'ils partagent la même table et ont la même clé primaire.

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

67

Identité en base de données

L'annotation @Id sur un champ de l'Entity Bean, ou mieux, sur son accesseur en lecture, le définit comme identifiant (clé primaire) en base de données

Il existe deux types d'identifiants : simple et composite

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

68

Identifiant simple

Les types simples acceptés sont :

- Les types de base (int, long, char, ...)
- Les classes correspondantes (Integer, Long, ...)
- Les classes String et Date

L'annotation `@GeneratedValue` indique la façon de générer la clé

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

69

Génération automatique de clé

Si la clé est de type numérique entier, l'annotation `@GeneratedValue` indique que la clé primaire sera générée automatiquement par le SGBD

Cette annotation peut avoir un attribut `strategy` qui indique comment la clé sera générée (il prend ses valeurs dans l'énumération `GeneratorType`)

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

70

Stratégies de génération de clé

Type	Stratégie
IDENTITY	Utilisation de la colonne identity de certaines base de données. L'identifiant renvoyé est de type long, short ou int.
SEQUENCE	L'identifiant renvoyé est de type long, short ou int . Exploite l'annotation @SequenceGenerator.
TABLE	Utilise une table dédiée qui stocke les clés des tables générées. Exploite l'annotation @TableGenerator
AUTO	Utilise automatiquement l'un des moyens ci-dessus en fonction des possibilités du SGBD (valeur par défaut)

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

71

Attributs de @SequenceGenerator

Attributs	Rôle
name	Nom identifiant le SequenceGenerator
sequenceName	Nom de la séquence dans la base de données
initialValue	Valeur initiale de la séquence
allocationSize	Tranche de valeurs utilisée pour l'incrémentation de la clé (50 par défaut)

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

72

Identifiant composite

Suppose la définition d'une classe de clé primaire.
Cette classe doit :

- Définir les propriétés devant être liées
- Etre publique, serialisable et avoir un constructeur sans argument
- Surcharger les méthodes equals et hashCode
- Etre déclarée classe de clé primaire composite à l'aide de l'annotation @Embeddable

Ce document est la propriété exclusive de TELECOMET. Il ne peut pas être utilisé sans autorisation

73

Identifiant composite (2)

Il existe deux façons d'utiliser cette classe au sein d'un Entity Bean :

- La première est de définir une propriété avec la classe de la clé primaire. Cette propriété doit être annotée avec @EmbeddedId (et non Id)
- La seconde fait référence à la classe de la clé primaire via l'annotation @IdClass. La classe de l'Entity Bean déclare explicitement les composantes de la clé primaire annotées chacune par @Id

Ce document est la propriété exclusive de TELECOMET. Il ne peut pas être utilisé sans autorisation

74

Les champs persistants

Un jeu d'annotations standard est défini par JPA, applicables soit aux propriétés de l'Entity Bean, soit aux colonnes des tables correspondantes

Les annotations concernant les champs se mettent soit au niveau de la déclaration du champ soit de sa méthode d'accès en lecture (« getter »). C'est la position de l'annotation @Id qui détermine ce choix pour tous les champs de la classe

L'annotation des « getters » est meilleure, car certaines propriétés ne correspondent pas à des colonnes

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

75

Annotations liées aux propriétés simples

Tout propriété de l'Entity Bean non marquée de l'annotation @Transient est considérée comme persistante et toute propriété de l'Entity Bean est considérée comme annotée avec @Basic.

Cette annotation permet de spécifier la stratégie de récupération pour une propriété

Attributs	Rôle
fetch	LAZY : la valeur est chargée uniquement lors de son utilisation EAGER : la valeur est toujours chargée (valeur par défaut)
optional	Indique que la valeur de la colonne peut être nulle

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

76

Lazy loading

Si l'attribut fetch de l'annotation @Basic est LAZY, la propriété annotée est chargée de manière lazy (paresseuse) , c'est-à-dire que cette propriété ne sera pas lue en base de données au moment de la récupération de l'entité mais lors du premier accès à la propriété, par une requête complémentaire automatique.

Si l'accès à la propriété non initialisée est fait alors que l'entité est détachée une exception peut être levée.

Si l'attribut fetch de l'annotation @Basic est EAGER la propriété annotée est initialisée au moment de la récupération de l'entité (valeur par défaut)

Noter que le gain apporté par le lazy loading pour une propriété simple est très faible en terme de performance et l'accès en mode lazy devrait être évité.

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

77

Mapping de propriétés primitives

Type ou classe Java	Standard SQL built-in type
java.lang.Integer ou int	INTEGER
java.lang.Long ou long	BIGINT
short or java.lang.Short	SMALLINT
float or java.lang.Float	FLOAT
double or java.lang.Double	DOUBLE
java.math.BigDecimal	NUMERIC
java.lang.String	CHAR(x)
byte or java.lang.Byte	TINYINT
boolean or java.lang.Boolean	BIT
boolean or java.lang.Boolean	CHAR(1) {'Y' ou 'N'} ou {'T' ou 'F'}

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

78

Annotation des propriétés temporelles

L'annotation `@Temporal` permet de fournir des informations sur la façon dont les propriétés encapsulant des données temporelles (`Date` et `Calendar`) sont associées aux colonnes dans la table (`date`, `time` ou `timestamp`).

La valeur par défaut est `timestamp`.

```
@Temporal(TemporalType.DATE)
private Calendar dateNaissance;
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

79

Mapping des propriétés temporelles

Type ou classe Java	Standard SQL built-in type
java.util.Date or java.sql.Date	DATE
java.util.Date or java.sql.Time	TIME
java.util.Date or java.sql.Timestamp	TIMESTAMP
java.util.Calendar	TIMESTAMP
java.util.Calendar	DATE

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

80

Annotation de propriétés complexes

L'annotation `@Lob` (Large Object) permet le mapping d'une propriété sur une colonne de type Blob (Binary LOB) ou Clob (Character LOB) selon le type de la propriété :

- Blob pour les tableaux de byte ou Byte ou les objets sérializables
- Clob pour les chaînes de caractères et les tableaux de caractères char ou Char

Fréquemment ce type de propriété est chargé de façon LAZY.

```
@Lob
@Basic(fetch = FetchType.LAZY)
private byte[] photo;
```

Ce document est la propriété exclusive de TELECOMET. Il ne peut pas être utilisé sans autorisation

81

Mapping de propriétés complexes

Type ou classe Java	Standard SQL built-in type
byte[]	VARBINARY
java.lang.String	CLOB
java.sql.Clob	CLOB
java.sql.Blob	BLOB
Toute classe Java class qui implémente java.io.Serializable	VARBINARY
java.lang.Class	VARCHAR
java.util.Locale	VARCHAR
java.util.TimeZone	VARCHAR
java.util.Currency	VARCHAR

Ce document est la propriété exclusive de TELECOMET. Il ne peut pas être utilisé sans autorisation

82

Annotation des propriétés énumérées

L'annotation `@Enumerated` permet d'associer une propriété de type énumération à une colonne de la table sous la forme d'un nombre ou d'une chaîne de caractères.

Cette forme est précisée en paramètre de l'annotation grâce à l'attribut `EnumType` qui peut avoir comme valeur `EnumType.ORDINAL` (valeur par défaut) ou `EnumType.STRING`.

```
public enum Genre { HOMME, FEMME}

@Enumerated(EnumType.STRING)
private Genre genre;
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

83

Annotation des colonnes simples

L'annotation `@Column` permet de paramétrer les colonnes.

Attributs	Rôle
name	Nom de la colonne
table	Nom de la table dans le cas d'un mapping multi-tables
unique	Indique si la valeur est unique dans la table
nullable	Indique si la colonne accepte une valeur nulle
insertable	Indique si la colonne doit être prise en compte dans les requêtes de type SQL INSERT
updatable	Indique si la colonne doit être prise en compte dans les requêtes de type SQL UPDATE
columnDefinition	Précise le DDL de définition de la colonne
length	Indique la taille d'une colonne de type chaîne de caractères (par défaut 255)
precision	Indique le nombre de chiffres maximum d'une colonne de type numérique
scale	Indique le nombre de chiffres après le séparateur (point) pour un décimal

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

84

Classes imbriquées

JPA permet d'agréger dans un Entity Bean des classes Java qui ne sont pas des Entity Beans mais dont les propriétés se retrouveront dans la table correspondant à l'Entity Bean.

Le champ de l'Entity Bean faisant référence à la classe imbriquée doit être annoté avec `@Embedded` et la classe imbriquée doit être annotée avec `@Embeddable`.

Les conflits de noms entre les propriétés de la classe imbriquée et l'Entity Bean peuvent être résolus avec l'annotation `@AttributeOverrides`

Ce document est la propriété exclusive de TELECOMET. Il ne peut pas être utilisé sans autorisation

85

Exemple

```
@Embeddable
public class Address implements java.io.Serializable {
    private String street;
    private String city;
    private String state;

    public String getStreet( ) { return street; }
    public void setStreet(String street) { this.street = street; }
}

    public String getCity( ) { return city; }
    public void setCity(String city) { this.city = city; }

    public String getState( ) { return state; }
    public void setState(String state) { this.state = state; }
}
```

Ce document est la propriété exclusive de TELECOMET. Il ne peut pas être utilisé sans autorisation

86

Exemple (2)

```
@Entity
@Table(name="CUSTOMER_TABLE")

public class Customer implements java.io.Serializable {
    private long id;
    private String firstName;
    private String lastName;
    private Address address;
    ...
    @Embedded
    @AttributeOverrides({
        @AttributeOverride
        (name="street",
        column=@Column(name="CUST_STREET")),
        @AttributeOverride(name="city",
        column=@Column(name="CUST_CITY")),
        @AttributeOverride(name="state",
        column=@Column(name="CUST_STATE"))
    })
    public Address getAddress( ) {
        return address;
    } ...
}
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

87

Mapping sur plusieurs tables

Il peut être utile de projeter un Entity Bean sur plusieurs tables dans le cas où la base de données existe déjà et ne correspond pas tout à fait au modèle objet

Les annotations `@SecondaryTable` et `@SecondaryTables` permettent de préciser qu'une ou plusieurs autres tables seront utilisées dans le mapping.

Pour utiliser cette fonctionnalité, la seconde table doit posséder une jointure entre sa clé primaire et une ou plusieurs colonnes de la première table

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

88

Exemple

Supposons que ces deux tables existent en base de données :

```
create table CUSTOMER_TABLE
(
    CUST_ID integer Primary Key Not Null,
    FIRST_NAME varchar(20) not null,
    LAST_NAME varchar(50) not null
);

create table ADDRESS_TABLE
(
    ADDRESS_ID integer primary key not null,
    STREET varchar(255) not null,
    CITY varchar(255) not null,
    STATE varchar(255) not null
);
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

89

Exemple (2)

```
@Entity
@Table(name="CUSTOMER_TABLE")@SecondaryTable(name="ADDRESS_TABLE",
    pkJoinColumns={
        @PrimaryKeyJoinColumn(name="ADDRESS_ID")})
public class Customer implements java.io.Serializable {
    private long id;
    private String firstName;
    private String lastName;
    private String street;
    private String city;
    private String state;
    ...
    @Column(name="STREET", table="ADDRESS_TABLE")
    public String getStreet( ) { return street; }
    public void setStreet(String street) { this.street = street; }

    @Column(name="CITY", table="ADDRESS_TABLE")
    public String getCity( ) { return city; }
    public void setCity(String city) { this.city = city; }

    @Column(name="STATE", table="ADDRESS_TABLE")
    public String getState( ) { return state; }
    public void setState(String state) { this.state = state; }
    ...
}
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

90

Les champs relationnels

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

91

Association One to One

Utilisée pour lier deux Entity Beans indissociables

Une prière solution consiste à utiliser la même valeur pour les clés primaires des deux entités

```
public class Item{  
    ...  
    @OneToOne  
    @PrimaryKeyJoinColumn  
    private Borrowing borrowing;  
}
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

92

Association One to One (2)

Une deuxième solution consiste à utiliser une clé étrangère

```
public class Item{
    @OneToOne
    @JoinColumn(name="BORROWING_ID")
    private Borrowing borrowing;
    ...
}
```

Pour rendre la relation bidirectionnelle, il faut utiliser l'attribut `mappedBy` qui déclare que la table possédant la relation est celle ayant l'attribut `borrowing`, donc ici celle correspondant à `Item`

```
public class Borrowing{
    ...
    @OneToOne(mappedBy = "borrowing")
    private Item item;
    ...
}
```

93

Ce document est la propriété exclusive de TELCOBLET. Il ne peut pas être utilisé sans autorisation

Utilisation d'une table d'association

Utile pour récupérer une base de données existante

```
public class Borrowing{

    @OneToOne
    @JoinTable(
        name="ITEM_BORROWING",
        joinColumns = @JoinColumn(name = "BORROWING_ID",
            unique=true),
        inverseJoinColumns = @JoinColumn(name = "ITEM_ID",
            unique=true)
    )
    private Item item;
    ...
    // Getter/setter methods
}
```

94

Ce document est la propriété exclusive de TELCOBLET. Il ne peut pas être utilisé sans autorisation

Association Many to One

```
public class Title {  
    ...  
    @ManyToOne  
    @JoinColumn(name = "ITEM_ID", nullable = false)  
    private Item item;  
    ...  
}
```

Pour rendre la relation bidirectionnelle, il faut utiliser l'annotation `@OneToMany` et l'attribut `mappedBy`

```
public class Item {  
    ...  
    @OneToMany(mappedBy = "item")  
    private Set<Title> titles= new HashSet<Title>();  
}
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

95

Association Many to Many

```
Public class Uer @ManyToMany  
@JoinTable(  
    name = "USER_ROLE",  
    joinColumns = {@JoinColumn(name = "ROLE_ID")},  
    inverseJoinColumns = {@JoinColumn(name =  
"USER_ID")}  
)  
private Set<Role> = roles new HashSet<Role>();
```

Pour rendre la relation bidirectionnelle, il faut utiliser l'annotation `@OneToMany` et l'attribut `mappedBy`

```
@ManyToMany(mappedBy = "roles")  
private Set<User> users= new HashSet<User>();
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

96

Ordre dans les collections

L'ordre d'une liste n'est pas nécessairement préservé dans la base de données

L'annotation `@OrderBy` indique dans quel ordre sont récupérées les entités associées

Il faut préciser un ou plusieurs attributs qui déterminent les critères d'ordonnement (l'ordre peut être précisé par `ASC` ou `DESC` - `ASC` par défaut)

Si aucun attribut n'est précisé, l'ordre sera celui de la clé primaire

```
@Entity
public class User{
    ...
    @OneToMany(mappedBy="user")
    @OrderBy("firstName ASC, lastName ASC")
    public List<Reservation> getReservations() {
```

97

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

Opérations en cascade

Les annotations `@OneToOne`, `@OneToMany`, `@ManyToOne` et `@ManyToMany` possèdent l'attribut `cascade`.

Celui-ci spécifie les opérations qui, appliquées aux entités, doivent être propagées aux entités associées.

Quatre types d'opérations peuvent être propagées. Ces opérations peuvent être regroupées dans l'énumération `CascadeType` :

- `CascadeType.PERSIST`
- `CascadeType.MERGE`
- `CascadeType.REMOVE` (uniquement `@OneToOne` et `@OneToMany`)
- `CascadeType.REFRESH`
- `CascadeType.ALL`

98

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

Exemple

```
@Entity
public class Message {

    @Id @GeneratedValue
    private Long id;

    private String text;

    @OneToOne(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    private Message nextMessage;
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

99

Relation de spécialisation entre Entity Beans

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

100

Stratégies

JPA supporte trois stratégies pour représenter la spécialisation :

- soit une seule table est attachée à une classe et toutes ses sous-classes
- soit une table correspond à chaque classe
- ou encore une table est seulement attachée aux classes concrètes (les "feuilles")

Une seule clé primaire doit être définie pour toute la hiérarchie.

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

101

Une table unique

Sans doute la stratégie la plus utilisée

Elle est performante et permet le polymorphisme mais elle induit beaucoup de valeurs NULL dans les colonnes si la hiérarchie est complexe

L'Entity Bean racine est repéré par l'annotation @Inheritance.

L'annotation @DiscriminatorColumn permet de préciser les détails de cette colonne, en particulier la classe du discriminateur (INTEGER, CHAR ou STRING)

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

102

Exemple

```
@Entity @Inheritance(strategy=
InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name= "role"
discriminatorType=DiscriminatorType.STRING)
public abstract class User {...}
private int id;
@Id public int getId() ...

@Entity @DiscriminatorValue("admin")
public class Administrator extends User {
...
}

@Entity @DiscriminatorValue("member")
public class Member extends User {
...
}
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

103

Une table par classe concrète

Chaque classe concrète est liée à sa propre table, ce qui signifie que toutes les propriétés de la classe (y compris les propriétés héritées) sont incluses dans la table liée à cette entité

L'avantage se situe au niveau de l'insertion, car l'ajout n'est fait que dans une seule table.

L'inconvénient majeur est la lourdeur des requêtes polymorphiques et la duplication des colonnes pour les propriétés héritées

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

104

Exemple

```
@Entity @Inheritance(strategy=
InheritanceType.TABLE_PER_CLASS)

public abstract class User {...}
private int id;
@Id public int getId() ...

@Entity
public class Administrator extends User {
...
}

@Entity
public class Member extends User {
...
}
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

105

Tables jointes

Toutes les classes sont représentées par leur table.

La liaison entre la classe racine et les classes filles se fait via les clés primaires.

L'héritage est ici déclaré avec la stratégie `InheritanceType.JOINED`

L'avantage est d'être proche du modèle objet, mais cette stratégie requiert l'utilisation de plusieurs jointures lors des requêtes

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

106

Comparatif des stratégies

Stratégie	SINGLE_TABLE	TABLE_PER_CLASS	JOINED
Avantages	Aucune jointure, donc très performant	Performant en insertion	Intégration des données proche du modèle objet
Inconvénients	Organisation des données non optimale	Polymorphisme lourd à gérer	Utilisation intensive des jointures, donc baisse de performance

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

107

Transactions et concurrence

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

108

Transactions

Qu'arrive t-il si une transaction (Cas d'utilisation) ne peut aboutir parce qu'une panne se produit au niveau système ? Il doit être possible de spécifier que toutes les étapes de la transaction doivent aboutir ou sinon aucune. Si une seule échoue, l'ensemble doit échouer.

Cette propriété porte le nom d'atomicité (atomicity). Toutes les opérations exécutées forment une unité de travail atomique.

Par ailleurs, les systèmes autorisent généralement de multiples utilisateurs à travailler de manière concurrente avec les mêmes ressources sans compromettre l'intégrité (integrity) et l'exactitude (correctness) des données. L'exécution d'une transaction ne devrait pas être visible des autres transactions s'exécutant de manière concurrente.

Plusieurs stratégies sont possibles pour assurer cette isolation des transactions.

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

109

Transactions (2)

Les transactions ont d'autres qualités importantes, telles que la consistance (consistency) et la durabilité (durability).

La consistance signifie qu'une transaction travaille sur un ensemble consistant de données : un ensemble de données est caché aux autres transactions concurrentes et laissé dans un état cohérent et propre lorsque la transaction est terminée.

L'exactitude d'une transaction est à la charge de l'application, tandis que la consistance est à la charge de la base de données.

La durabilité signifie qu'une fois la transaction terminée toutes les modifications opérées ne seront pas perdues si le système tombe ensuite en panne.

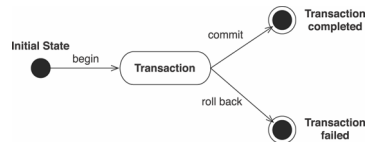
L'ensemble de ces critères de qualité des transactions est connu sous le nom de critères "ACID"

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

110

Transactions en base de données

Toutes les requêtes SQL en base de données se font à l'intérieur d'une transaction et les transactions sont atomiques : une transaction aboutit (commit) ou est annulée (rolled back).



Dans un environnement non géré par le conteneur l'API JDBC est utilisée pour marquer les limites des transactions. La méthode `setAutoCommit(false)` est appelée sur une connexion JDBC pour démarrer une transaction et `setAutoCommit(true)` est appelée à la fin de la transaction. Il est possible à tout instant de forcer un rollback en appelant la méthode `rollback()`.

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

111

Exemple JDBC

```
public void buyBooks(ShoppingCart cart) throws OrderException {  
    try {  
        getConnection();  
        con.setAutoCommit(false);  
  
        ...  
  
        con.commit();  
        con.setAutoCommit(true);  
        releaseConnection();  
    } catch (Exception ex) {  
        try {  
            con.rollback();  
            releaseConnection();  
            throw new OrderException("Transaction failed: " +  
                ex.getMessage());  
        } catch (SQLException sqx) {  
            releaseConnection();  
            throw new OrderException("Rollback failed: " +  
                sqx.getMessage());  
        }  
    }  
}
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

112

Transactions Resource-local

L'appellation resource-local s'applique à toutes les transactions contrôlées directement par le code applicatif et qui ne participent pas à un système global de gestion transactionnelle.

Les transactions sont gérées par le système de gestion natif de la ressource utilisée.

Dans le cas d'une base de données une transaction de type resource-local transaction s'exprime sous forme de transaction JDBC

Ce document est la propriété exclusive de TELECOMET. Il ne peut pas être utilisé sans autorisation

113

Gestion des transactions avec l'API JTA

Dans un système qui utilise plusieurs bases de données il n'est pas possible de se limiter à JDBC pour assurer l'atomicité des transactions.

Il est nécessaire d'utiliser un moniteur transactionnel qui est capable de gérer plusieurs ressources dans une même transaction.

En Java le service de gestion transactionnelle correspond à JTA (Java Transaction API) . La gestion applicative d'une transaction se fait en particulier avec les méthodes begin(), commit() et rollback() de l'interface **UserTransaction**

Cette API peut être utilisée de manière déclarative dans un serveur d'applications Java EE.

Avec JPA, l'interface **EntityTransaction** permet de démarrer et d'achever la transaction à laquelle le gestionnaire d'entités prend part.

Dans un environnement avec conteneur, l'interface EntityTransaction de JPA est directement liée à l'interface UserTransaction de JTA.

Ce document est la propriété exclusive de TELECOMET. Il ne peut pas être utilisé sans autorisation

114

Exemple de gestion applicative d'une transaction avec JPA

```
EntityManager em = null;
EntityTransaction tx = null;

try {
    em = emf.createEntityManager();
    tx = em.getTransaction();
    tx.begin();
    ...
    tx.commit();
} catch (RuntimeException ex) {
    try {
        tx.rollback();
    } catch (RuntimeException rbEx) {
        log.error("Couldn't roll back transaction", rbEx);
    }
    throw ex;
} finally {
    em.close();
}
```

Ce document est la propriété exclusive de TELECOMET. Il ne peut pas être utilisé sans autorisation.

115

Exceptions

Les exceptions levées avec JTA sont des sous-classes de `RuntimeException`. Toute exception invalide le contexte de persistance courant et il n'est pas possible de continuer à utiliser un `EntityManager` une fois qu'une exception a été levée.

Toute exception levée par une méthode de l'`EntityManager` déclenche un rollback automatique de la transaction courante.

Toute exception levée par une méthode de Query déclenche un rollback automatique de la transaction courante, sauf pour les exceptions de classe `NoResultException` et `NonUniqueResultException`.

Le code de l'exemple précédent attrape toutes les exceptions et provoque un roll back de la transaction.

Ce document est la propriété exclusive de TELECOMET. Il ne peut pas être utilisé sans autorisation.

116

Transactions JTA container-managed

Le contexte de persistance d'un EntityManager container-managed a une portée limitée à la transaction JTA

C'est-à-dire que toutes les requêtes SQL exécutées à l'intérieur d'une transaction JTA le sont sur une connexion JDBC gérée par la transaction.

La synchronisation de la base de données et la fermeture du contexte de persistance se font quand la transaction JTA est validée (commits). Il est possible d'utiliser plusieurs EntityManagers pour accéder à différentes bases de données dans une transaction.

Il est possible de gérer manuellement les transactions des EJB en annotant leur classe @TransactionManagement(TransactionManagementType.BEAN)

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

117

Exemple

```
@Stateless
public class MessageBean implements MessageRemote {

    @PersistenceContext(unitName= "pu")
    private EntityManager em;

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void saveMessages() {
        Message message = new Message("Hello World with EJB3");
        em.persist(message);
        Message nextMessage = new Message("Hi !");
        message.setNextMessage(nextMessage);
    }
}
```

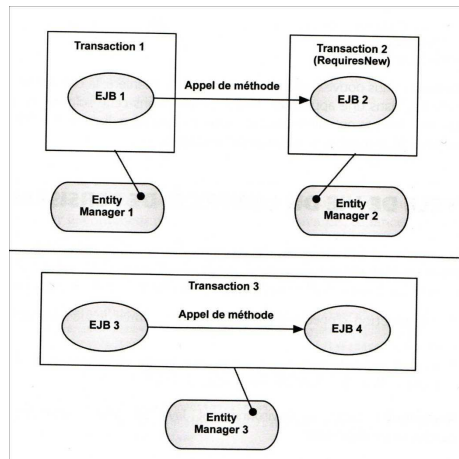
La valeur de TransactionAttributeType de l'attribut TransactionAttribute pour la méthode saveMessages() exige que tous les accès en base de données se fassent dans une transaction.

Si aucune transaction n'est active quand cette méthode est appelée, une nouvelle transaction est démarrée pour cette méthode et la transaction se termine avec la méthode. La portée du contexte de persistance est alors celui de la méthode.

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

118

Transactions JTA container-managed (2)



Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

119

Exceptions

Les règles de traitement des exceptions sont les mêmes que pour les transactions de type resource-local .

Toutefois, il n'est pas nécessaire d'attraper ou de propager toute exception levée par les opérations JPA – laissez le conteneur s'en occuper.

Deux options sont possibles pour provoquer un roll back si une exception applicative telle que `MonMessageNotValidException` est levée :

- Il est possible d'attraper l'exception et d'appeler l'objet `UserTransaction` pour provoquer un roll back.
- Il est aussi possible d'ajouter l'annotation `@ApplicationException(rollback = true)` à la classe `MonMessageNotValidException` – le conteneur saura que vous voulez faire un roll back automatique quand cette exception sera levée

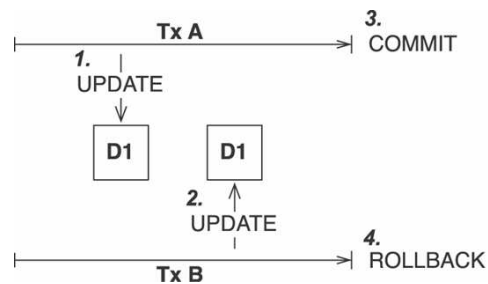
Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

120

Isolation des transactions

Une modification peut être perdue (lost update) si deux transactions modifient un tuple en base de données et que l'une d'elles échoue.

Ceci se produit quand le système n'implémente pas le verrouillage (locking). Les transactions concurrentes ne sont pas isolées.

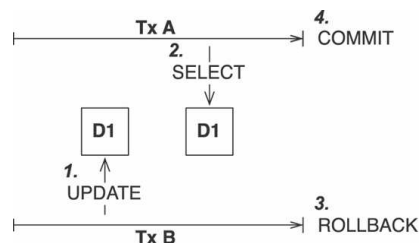


121

Isolation des transactions (2)

Une lecture sale (dirty read) se produit si une transaction lit les modifications faites par une autre transaction qui n'a pas encore été validée.

Ceci est dangereux, car les modifications faites par la seconde transaction peuvent ensuite être annulées et des données invalides peuvent être utilisées par la première transaction.

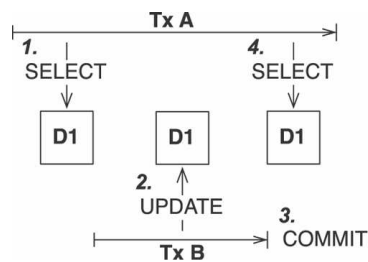


122

Isolation des transactions (3)

Une lecture non répétable (unrepeatable read) se produit si une transaction lit un tuple deux fois et trouve des valeurs différentes. Ceci peut se produire si une autre transaction modifie le tuple entre deux lectures de la première.

Un lecture non répétable peut se produire si deux transactions lisent un même tuple simultanément puis le modifient : les modifications faites par la première qui se termine sont perdues.



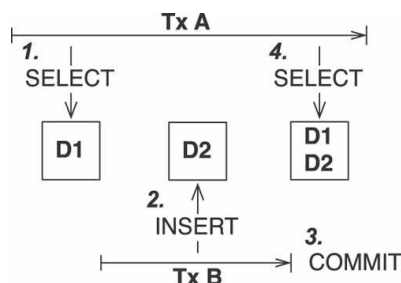
123

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

Isolation des transactions (4)

Une lecture fantôme (phantom read) se produit quand une transaction effectue deux fois une requête et que le premier résultat (result set) inclut des tuples qui n'étaient pas visibles dans le premier ou que certains tuples ont été supprimés du premier.

Cette situation peut avoir été produite par une autre transaction qui a inséré ou supprimé des tuples entre les deux requêtes.



124

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

ANSI transaction isolation levels

Les niveaux d'isolation standards sont définis par la norme ANSI SQL, mais ils ne sont pas particuliers aux bases de données SQL.

JTA définit exactement les mêmes niveaux

Un niveau d'isolation élevé peut provoquer une forte dégradation des performances et affecter l'extensibilité d'une application

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

125

ANSI transaction isolation levels (2)

A system that permits dirty reads but not lost updates is said to operate in read uncommitted isolation. One transaction may not write to a row if another uncommitted transaction has already written to it. Any transaction may read any row, however. This isolation level may be implemented in the database-management system with exclusive write locks.

A system that permits unrepeatable reads but not dirty reads is said to implement read committed transaction isolation. This may be achieved by using shared read locks and exclusive write locks. Reading transactions don't block other transactions from accessing a row. However, an uncommitted writing transaction blocks all other transactions from accessing the row.

A system operating in repeatable read isolation mode permits neither unrepeatable reads nor dirty reads. Phantom reads may occur. Reading transactions block writing transactions (but not other reading transactions), and writing transactions block all other transactions.

Serializable provides the strictest transaction isolation. This isolation level emulates serial transaction execution, as if transactions were executed one after another, serially, rather than concurrently. Serializability may not be implemented using only row-level locks. There must instead be some other mechanism that prevents a newly inserted row from becoming visible to a transaction that has already executed a query that would return the row.

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

126

Contrôle optimiste de la concurrence

Une approche optimiste suppose que les conflits entre les modifications de données seront rares.

Le contrôle optimiste de la concurrence se contente de détecter les erreurs au moment où les données sont écrites.

Généralement les applications multi-utilisateurs se basent sur un contrôle optimiste de la concurrence avec un niveau d'isolation "read-committed"

Ce niveau d'isolation apporte les meilleures performances et d'extensibilité (*scalability*)

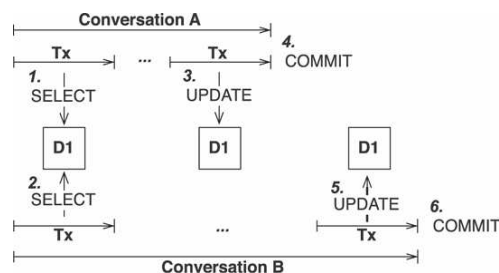
Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

127

Contrôle optimiste de la concurrence (2)

Pour illustrer le contrôle optimiste de concurrence, supposons que deux transactions lisent un objet en base de données et le modifient toutes deux.

Grâce au niveau d'isolation "read-committed" aucune des deux transactions ne fera de lecture sale. Cependant les lectures seront non-répétables car des modifications peuvent être perdues.



Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

128

Contrôle optimiste de la concurrence (3)

Trois stratégies sont alors possibles :

- Le dernier commit gagne — Les deux transactions se terminent et aucune erreur n'est signalée
- Le premier commit gagne — Une erreur est signalée à la transaction qui se termine en dernier. Il est alors possible de redémarrer cette transaction complètement après avoir rafraîchi les données initiales.
- Fusion des modifications. Comme dans le cas précédant la transaction qui aboutit en dernier est invalidée, mais l'utilisateur peut appliquer les modifications de manière sélective, sans reprendre l'ensemble de la transaction

Java Persistence permet un verrouillage optimiste qui lève une exception en cas de conflit de synchronisation en base de données.

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

129

Gestion de version avec JPA

La spécification JPA suppose l'accès concurrent des données se fait de manière optimiste, avec gestion de version (versioning)

Pour rendre automatique la gestion de version d'un Entity Bean il faut lui ajouter une propriété avec l'annotation @Version

La version, d'une entité peut être de type numérique ou timestamp.

Le gestionnaire de persistance incrémente la version d'une entité quand celle-ci est modifiée et une comparaison de version est faite au moment de la synchronisation. en base de données. Une exception est levée si un conflit est détecté.

```
@Entity
public class Item {
    ...
    @Version
    @Column(name = "OBJ_VERSION")
    private int version;
    ...
}
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

130




JPQL

(Université de nice - Richard Grin)

Ce document est la propriété exclusive de TELECOMET. Il ne peut pas être utilisé sans autorisation

131



Recherche par clé primaire

find et getReference (de EntityManager)
permettent de retrouver une entité en donnant son identificateur dans la BD

La méthode find() retourne null si aucune entité n'est identifiée par la clé primaire alors que la méthode getReference() lève une exception EntityNotFoundException.

Ce document est la propriété exclusive de TELECOMET. Il ne peut pas être utilisé sans autorisation

132

Exemples de requêtes JPQL

from Employe as e

select e from Employe as e

select e.nom, e.salaire from Employe e

select e from Employe e where e.departement.nom = 'Direction'

select d.nom, avg(e.salaire) from Departement d join d.employes e
group by d.nom having count(d.nom) > 5

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

133

Type du résultat

L'expression de la clause select peut être :

- une (ou plusieurs) expression « entité », par exemple un employé (**e par exemple**)
- une (ou plusieurs) expression « valeur », par exemple le nom et le salaire d'un employé (**e.nom par exemple**)

Si la requête renvoie des entités, elles sont automatiquement gérées par le GE (toute modification sera répercutée dans la base)

L'expression ne peut être une collection

(**d.employes par exemple**), bien que TopLink le permette !

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

134

Obtenir le résultat de la requête

Pour le cas où une seule valeur ou entité est renvoyée, le plus simple est d'utiliser la méthode `getSingleResult()` de l'interface `Query` qui renvoie un `Object`

La méthode lance des exceptions s'il n'y a pas exactement une entité qui correspond à la requête :

- `EntityNotFoundException`
- `NonUniqueResultException`

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

135

Obtenir le résultat de la requête (2)

Si plusieurs valeurs ou entités peuvent être renvoyées, il faut utiliser la méthode `getResultList()` de l'interface `Query`

Elle renvoie une liste « raw » (non générique) des résultats, instance de `java.util.List`, éventuellement vide si le résultat est vide

Un message d'avertissement est affiché durant la compilation si le résultat est rangé dans une liste générique (`List<Employee>` par exemple)

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

136

Type d'un élément du résultat

Le type d'un élément de la liste (ou de l'unique valeur ou entité renvoyée) est :

- Object si la clause select ne comporte qu'une seule expression
- Object[] si elle comporte plusieurs expressions

Ce document est la propriété exclusive de TELECOMET. Il ne peut pas être utilisé sans autorisation.

137

Exemples

```
String s = "select e from Employee as e";  
Query query = em.createQuery(s);  
List<Employee> listeEmployes =  
(List<Employee>)query.getResultList();
```

```
texte = "select e.nom, e.salaire " + " from Employee as e";  
query = em.createQuery(texte);  
List<Object[]> liste =  
(List<Object[]>)query.getResultList();  
for (Object[] info : liste) {  
    System.out.println(info[0] + " gagne " + info[1]);  
}
```

Ce document est la propriété exclusive de TELECOMET. Il ne peut pas être utilisé sans autorisation.

138

Interface Query

Représente une requête

Une instance de Query (d'une classe implémentant Query) est obtenue par les méthodes `createQuery`, `createNativeQuery` ou `createNamedQuery` de l'interface `EntityManager`

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

139

Types temporels

On a vu que les 2 types java temporels du package `java.util` (`Date` et `Calendar`) nécessitent une annotation `@Temporal`

Ils nécessitent aussi un paramètre supplémentaire pour la méthode `setParameter`

Exemple

```
@Temporal(TemporalType.DATE)
private Calendar dateEmb;
em.createQuery("select e from employe e" + " where e.dateEmb
between ?1 and ?2").setParameter(1, debut, TemporalType.DATE)
.setParameter(2, fin, TemporalType.DATE).getResultList();
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

140

Types de requête

Requêtes dynamiques dont le texte est donnée en paramètre de **createQuery**

Requêtes natives (appelées aussi requêtes SQL) particulières à un SGBD ou trop complexes pour JPA; requête SQL (pas JPQL) avec tables et colonnes (pas classes et attributs)

Requêtes nommées dont le texte est donnée dans une annotation de l'entité concernée et dont le nom est passé en paramètre de **createNamedQuery** ; une **requête nommée** peut être dynamique ou native

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

141

Paramètres des requêtes

Un paramètre peut être désigné par son numéro (**?n**) **ou par son nom (:nom)**

Les valeurs des paramètres sont données par les méthodes **setParameter**

Les paramètres sont numérotés à partir de 1

Un paramètre peut être utilisé plus d'une fois dans une requête

L'usage des paramètres nommés est recommandé (plus lisible)

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

142

Requête nommée

Seules les entités peuvent contenir des définitions de requêtes nommées

Une requête nommée peut être mise dans n'importe quelle entité, mais on choisira le plus souvent l'entité qui correspond à ce qui est renvoyé par la requête

Le nom de la requête nommée doit être unique parmi toutes les entités de l'unité de persistance ; on pourra, par exemple, préfixer le nom par le nom de l'entité : `Employe.findAll`

Les requêtes nommées peuvent être analysées et précompilées par le fournisseur de persistance au démarrage de l'application, ce qui peut améliorer les performances

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

143

Exemple de requête nommée

```
@Entity
@NamedQuery (name = "findNomsEmployes", query = "select e.nom
from Employe as e where upper(e.departement.nom) = :nomDept")
public class Employe extends Personne {
...

Query q = em.createNamedQuery("findNomsEmployes");
```

```
Query q = em.createQuery("select e from Employe as e " + "where
e.nom = ?1");

query.setParameter(1, "Dupond");

Query query = em.createQuery("select e from Employe as e " +
"where e.nom = :nom");

query.setParameter("nom", "Dupond");
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

144

Plusieurs requêtes nommées

Si une classe a plusieurs requêtes nommées,
il faut les regrouper dans une annotation
@NamedQueries :

```
@NamedQueries({  
    @NamedQuery(...),  
    @NamedQuery(...),  
    ...  
})
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

145

Clauses d'un select

Les mots-clés **select, from, distinct, join,...** sont insensibles à la casse

```
select : type des objets ou valeurs renvoyées  
from : où les données sont récupérées  
where : sélectionne les données  
group by : regroupe des données  
having : sélectionne les groupes (ne peut exister sans clause  
group by)  
order by : ordonne les données
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

146

Polymorphisme dans les requêtes

Toutes les requêtes sont polymorphes : un nom de classe dans la clause from désigne cette classe et toutes les sous-classes

Exemple :

```
select count(a) from Article as a
```

compte le nombre d'instances de la classe Article et de tous les sous-classes de Article

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

147

Expression de chemin

Les requêtes peuvent contenir des expressions de chemin pour naviguer entre les entités en suivant les associations déclarées dans le modèle objet (les annotations @OneToOne, @OneToMany, ...)

La notation « pointée » est utilisée

Une navigation peut être chaînée à une navigation précédente à la condition que la navigation précédente ne donne qu'une seule entité (OneToOne ou ManyToOne)

Dans le cas où une navigation aboutit à plusieurs entités, il est possible d'utiliser la clause join étudiée plus loin pour obtenir ces entités

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

148

Exemples

Si e est un alias pour Employe,

- « e.departement » désigne le département d'un employé
- « e.projets » désigne la collection de projets auxquels participe un employé

```
select e.nom from Employe as e
where e.departement.nom = 'Qualité'
```

e.projets.nom n'est pas autorisé car e.projets est une collection (voir clause join)

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

149

distinct

Dans une clause select, indique que les valeurs dupliquées sont éliminées (la requête ne garde qu'une seule des valeurs égales)

```
select distinct e.departement from Employe e
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

150

Il est possible de renvoyer des instances d'une classe dont le constructeur prend en paramètre des informations récupérées dans la base de données

La classe doit être désignée par son nom complet (avec le nom du packaging)

Exemple :

```
select new p1.p2.Classe(e.nom, e.salaire) from Employe e
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

151

Restriction : la condition doit porter sur l'expression de regroupement ou sur une fonction de regroupement portant sur l'expression de regroupement

Par exemple, la requête suivante provoque une exception :

```
select d.nom, avg(e.salaire) from Departement d join d.employes  
e group by d.nom having avg(e.salaire) > 1000
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

152

Sous-requête (1)

Les clauses where et having peuvent contenir des sous-requêtes

Exemple :

```
select e from Employe e where e.salaire >= (select e2.salaire
from Employe e2 where e2.departement = 10)
```

Ce document est la propriété exclusive de TELÉCOM. Il ne peut pas être utilisé sans autorisation

153

Sous-requête (2)

{**ALL** | **ANY** | **SOME**} (*sous-requête*) fonctionne comme dans SQL

Exemple :

```
select e from Employe e where e.salaire >= ALL ( select
e2.salaire from Employe e2 where e2.departement =
e.departement)
```

Ce document est la propriété exclusive de TELÉCOM. Il ne peut pas être utilisé sans autorisation

154

Sous-requête synchronisée

Une sous-requête peut être synchronisée avec une requête englobante

Exemple :

```
select e from Employee e where e.salaire >= ALL (select
e2.salaire from Employee e2 where e2.departement =
e.departement)
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

155

Sous-requête - exists

[not]exists fonctionne comme avec SQL

Exemples :

```
select emp from Employee e where exists (select ee from Employee
ee where ee = e.epouse)

select e.nom, e.departement.nom, e.superieur.departement.nom
from Employee e
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

156

Jointure

Une jointure permet de combiner plusieurs entités dans un select

Rappel : il est possible d'utiliser plusieurs entités dans une requête grâce à la navigation

Une jointure est le plus souvent utilisée pour résoudre les cas (interdit par JPA) où

- l'expression du select serait une collection
- la navigation partirait d'une collection

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

157

Types de jointures

Il existe plusieurs types de jointures :

- jointure interne (jointure standard **join**)
- jointure externe (**outer join**)
- jointure avec récupération de données en mémoire (**join fetch**)
- jointure « à la SQL » dans un where pour joindre suivant des champs qui ne correspondent pas à une association (where e1.f1 = e2.f2)

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

158

Exemple

Si on veut tous les employés d'un département, la requête suivante n'est pas permise par la spécification JPA (bien que TopLink l'autorise) :

```
select d.employees from Departement d where d.nom = 'Direction'
```

Une jointure est nécessaire

```
select e from Departement d join d.employees e where d.nom = 'Direction'
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

159

Autres exemples

```
select e.nom from Departement d join d.employees e where d.nom = 'Direction'
```

```
select e.nom, parts.projet.nom from Employe e join e.participations parts
```

```
select e.nom, d.nom from Employe e, Departement d where d = e.departement
```

```
select e, p from Employe e join e.participations parts join parts.projet p
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

160

Jointure externe

```
select e, d from Employe e left join e.departement d
```

ramènera aussi les employés qui ne sont pas associés à un département

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

161

join fetch

Permet d'éviter le problème des « N + 1 selects »

L'entité placée à droite de join fetch sera créée en mémoire en même temps que l'entité de la clause select

Le select SQL généré sera une jointure externe qui récupérera les données de toutes les entités associées en même temps que les données des entités principales de la requête

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

162

Exemple

```
select e from Employe e join fetch e.departement
```

Cette requête récupérera tous les employés mais, en plus, l'appel de la méthode `getDepartement()` ne provoquera aucune interrogation de la base de données puisque le « join fetch » aura déjà chargé tous les départements des employés

L'exemple suivant précharge les collections de participations aux projets

```
String texteQuery = "select e " + " from Employe as e " + "  
join fetch e.participations";  
Query query = em.createQuery(texteQuery);  
listeEmployes = (List<Employe>)query.getResultList();
```

Il existe aussi des variantes « outer join » de join fetch pour récupérer dans le select des entités non jointes à une autre entité

163

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

Doublons possibles avec join fetch

La requête SQL lancée par un join fetch fait une jointure pour récupérer les entités préchargées

Ensuite, les entités préchargées sont enlevées

des lignes du résultat pour qu'elles n'apparaissent pas dans le résultat du query

Ce traitement, imposé par la spécification de JPA, peut occasionner des doublons dans le résultat si le select renvoie des valeurs

Pour les éviter, il faut ajouter l'opérateur `DISTINCT` dans le texte de la requête ou placer le résultat dans une collection de type `Set`

164

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

Produit cartésien

Le préchargement par join fetch de plusieurs collections d'une même entité peut occasionner un phénomène nuisible aux performances

En effet, le select SQL généré par le fournisseur de persistance peut récupérer un produit cartésien des éléments des collections

Le fournisseur s'arrange pour ne garder que les informations nécessaires mais le select peut renvoyer un très grand nombre de lignes qui vont transiter par le réseau

Dans le cas où les collections contiennent de nombreux éléments il faut donc vérifier avec les logs du fournisseur si le select généré renvoie effectivement un trop grand nombre de lignes et changer de stratégie de récupération si c'est le cas (récupérer séparément les informations sur les collections)

En effet, si 2 collections contiennent 20 éléments, et si 1000 entités principales sont renvoyées le select renverra 400.000 lignes ! (au lieu de 40.000 si on ramène les informations avec 2 join fetch séparés)

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

165

Exemple

Une entité principale ep avec 2 associations de l'entité principale avec d'autres entités e2 et e3

Pour récupérer en mémoire les entités principales et associées, le select généré risque d'être du type suivant (consulter les logs du fournisseur de persistance) :

```
select ep.*, e2.*, e3.* from EP ep left outer join E2 e2 on  
ep.id_e2 = e2.id left outer join E3 e3 on ep.id_e3 = e3.id
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

166

Fonctions

Fonctions de chaînes de caractères : concat substring, trim, lower, upper, length, locate (localiser une sous-chaîne dans une autre)

Fonctions arithmétiques : abs, sqrt, mod, size (d'une collection)

Fonctions de date : current_date, current_time, current_timestamp

Fonctions de regroupement : count, max, min, avg

Spécification JPA pour plus d'informations,

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

167

Travail avec les collections

Une expression chemin d'un select peut désigner une collection

Exemples :

■ departement.employees

■ facture.lignes

La fonction size donne la taille de la collection

La condition « is [not] empty » est vraie si la collection est [n'est pas] vide

La condition « [not] member of » indique si une entité appartient à une collection

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

168

Parcours d'une collection

Le plus simple est d'utiliser un join dans le from avec un alias (« l » dans l'exemple cidessous) pour désigner un élément qui parcourt la collection :

```
select distinct l.produit from Facture as f join f.lignes as l
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

169

Pagination du résultat

Query setMaxResults(int n) : indique le nombre maximum de résultats à retrouver

Query setFirstResult(int n) : indique la position du 1er résultat à retrouver (numéroté à partir de 0)

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

170

Enchaînement des méthodes

Les méthodes `setParameter`, `setMaxResults` renvoient le Query modifié

On peut donc les enchaîner

Exemple :

```
em.createQuery(texteQuery).setParameter(nomParam, valeurParam)
.setFirstResult(30 * i + 1).setMaxResults(30).getResultList();
```

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

171

Opérations de modification en volume

Si on veut augmenter de 5% les 1000 employés de l'entreprise il serait mauvais de récupérer dans 1000 instances les données de chaque employés, de modifier le salaire de chacun, puis de sauvegarder les données

Un simple ordre SQL

```
update employe set salaire = salaire * 1.05
```

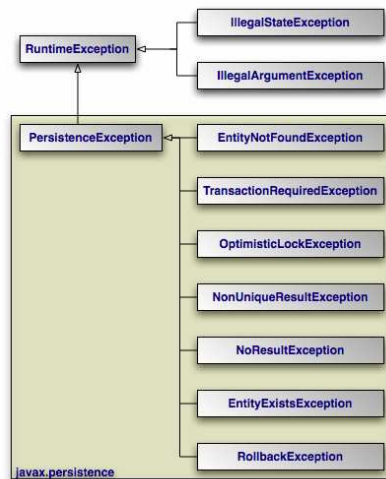
sera énormément plus performant

Les modifications doivent être faites dans une transaction

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

172

Exceptions JPA



Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

173

Exceptions

Toutes les exceptions de type PersistenceException provoquent un rollback, sauf

- NonUniqueResultException et
- NoResultException

Les RuntimeException provoquent un rollback de la transaction en cours

Les exceptions contrôlées sont enveloppées dans une PersistenceException (non contrôlée) et provoquent un rollback

Ce document est la propriété exclusive de TELECOM. Il ne peut pas être utilisé sans autorisation

174

Références



Entreprise JavaBeans 3.0 par Bill Burke et Richard Monson-Haefel. Editions O'Reilly 2006



Java Persistence with Hibernate par Christian Bauer et Gavin King. Editions Manning 2007



Java Persistence et Hibernate par Anthony Patricio. Editions Eyrolles 2008



EJB3 Des concepts à l'écriture de code. Guide du développeur par le laboratoire SUPINFO des technologies Sun. Editions Dunod 2006

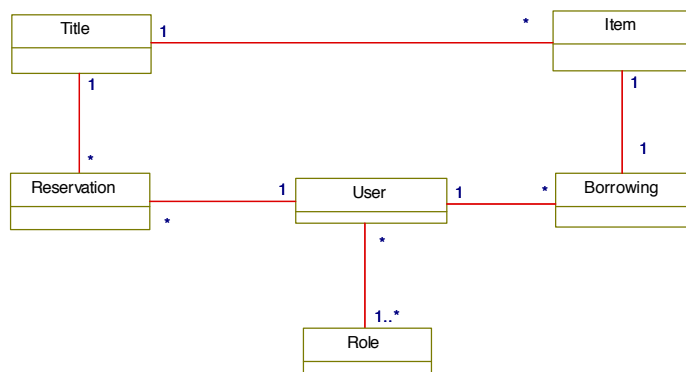


JBoss SEAM Simplicity and power beyond Java EE par Michael Juntao et Thomas Heute. Editions Prentice Hall 2007

Ce document est la propriété exclusive de TELCOLEUT. Il ne peut pas être utilisé sans autorisation

175

Exemple Library



Ce document est la propriété exclusive de TELCOLEUT. Il ne peut pas être utilisé sans autorisation

176