

# JavaEE : Enterprise Java Bean

Jonathan Lejeune

Sorbonne Université/LIP6-INRIA

SRCS – Master 1 SAR 2019/2020

**sources :**

Développons en Java, Jean-Michel Doudoux

Cours précédent de Lionel Seinturier

Wikipédia

Cours de Jérôme Hugues et Ada Diaconescu (Telecom ParisTech)

# Présentation des EJB

## Définition

Composants logiciels côté serveur pour la plateforme de développement Java EE.

## Les EJBs en pratique

- Un ensemble de :
  - deux interfaces Java (facultatives depuis EJB 3 mais recommandées)
  - une classe Java d'implantation
- Le tout regroupé dans un module (archive .jar) contenant un descripteur de déploiement (fichiers .xml).

## Types d'EJB

- Les EJB de session (*Session beans*)
- les EJB entité (*Entity beans*)
- les EJB orientés messages (*Message driven beans*)

## EJB 2 (2001)

Profite du succès de JavaEE mais :

- programmation trop compliquée, lourde et contraignante :
  - Création de plusieurs interfaces et classes Java
  - Implémentation de méthodes callback généralement vides et inutiles ejbActivate, ejbLoad, ejbPassivate, ...
  - L'interface de l'EJB doit hériter de EJBObject ou EJBLocalObject
  - tests et debug difficile
- Descripteur de déploiement complexe
  - trop de fichiers XML à écrire, maintenir et comprendre

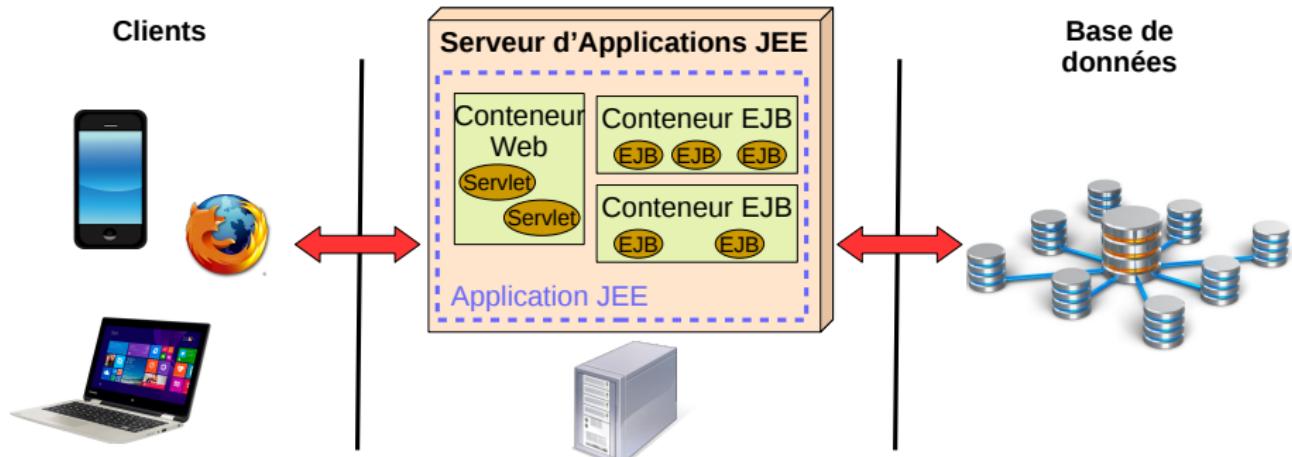
# EJB 2 vs. EJB 3

## EJB 3 (2006)

- Rétrocompatibilité
- Programmation simplifiée grâce aux **annotations** de Java 5 :
  - Les EJB sont des POJO (Plain Old Java Object) annotés  
⇒ Plus besoin d'implémenter une interface de l'API EJB
  - Les interfaces métiers sont facultatives et sont des POJI annotés
  - Injection de dépendance facilitant l'obtention d'instance EJB du même conteneur
- Descripteur de déploiement optionnel :
  - Fichiers XML simplifiés voire supprimés

Application	Métrique	EJB 2	EJB 3	Gain
RosterApp	Nb Classes	17	7	<b>59%</b>
	Nb lignes de code	987	716	<b>27%</b>
	Nb fichiers XML	9	2	<b>78%</b>
	Nb lignes de XML	792	26	<b>97%</b>

# EJB et architecture 3-tier



## Adaptation parfaite pour les client légers

- **Présentation** : affichage et saisie des données par le client
- **Traitements** : le serveur d'appli. hébergeant les conteneurs d'EJB
- **Stockage** : une base de données dédiée pour la persistance des info

## Caractéristiques

- Entité obligatoire pour exécuter un EJB
- Propose des services qui assure la gestion :
  - du cycle de vie du bean
  - de l'accès au bean via le réseau (notification auto à l'arrivée d'un message)
  - de la sécurité des accès (ex : cryptage)
  - des accès concurrents
  - de la synchronisation des données d'un objet avec une BD
  - du contrôle de la charge :
    - fabrique de nouvelles instances
    - surveillances de instances inactives
  - des transactions

Toute entité externe au serveur communiquent indirectement avec l'EJB  
**via le conteneur**

## API interdite d'utilisation par le programmeur

- Thread
- flux I/O
- code natif (méthode `native`)
- API graphique : AWT, Swing, JavaFx

**Seul le conteneur peut utiliser ces API**

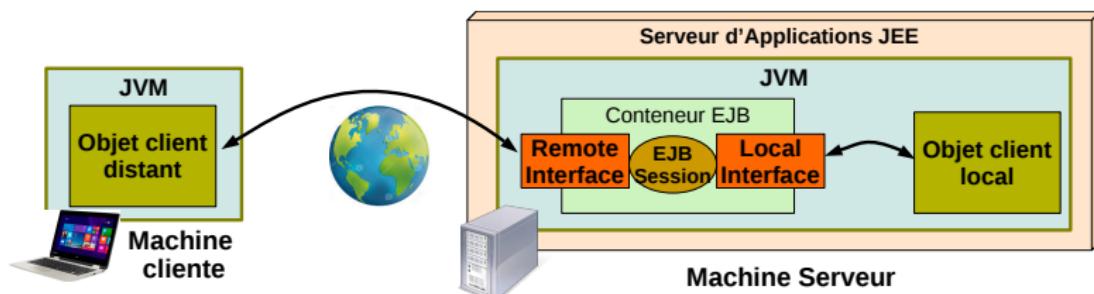
## Caractéristiques

- Représente un ou plusieurs traitements (= services fournis à un client)
- Peuvent faire appel à/être utilisés par d'autres composants :
  - Autres EJB session
  - EJB Entity
  - POJO
  - Servlet
  - ...
- Fortement recommandé d'y associer une interface :
  - locale
  - et/ou distante
- 3 types d'EJB session :
  - Stateless
  - Stateful
  - Singleton

# Les interfaces des EJB Session

## Deux types d'interfaces

- Distante :
    - les services (méthodes) offerts par l'EJB à ses clients
    - Interface Java avec l'annotation `@Remote`
    - Utilisation du réseau + Java RMI
  - Locale :
    - les services offerts par l'EJB aux clients présent dans la même JVM
    - Interface Java avec l'annotation `@Local`
    - Peut être identique ou différente à l'interface distante
- ⇒ Optimisation des performances car évite le réseau



# Les EJB Session stateless

## Caractéristiques

- Sans état :
  - 1 instance par invocation
  - Pas d'information conservée entre 2 appels successifs
- Le conteneur gère un pool d'instance qui sont utilisées au besoin  
⇒ s'adapte à la montée en charge

## L'annotation @Stateless

S'utilise sur une classe Java et possède des attributs optionnels :

- String name : nom de l'EJB (défaut = nom de la classe)
- String description : description de l'EJB

## Les annotations de méthodes de la classe (callback événementiel)

- @PostConstruct : appelé après la construction
- @PreDestroy : appelé avant la destruction

# Les EJB Session stateless : un exemple

## L'interface

```
@Remote  
public interface CalculRemote{  
    public int add(int a, int b);  
}  
  
@Local  
public interface CalculLocal{  
    public int add(int a, int b);  
    public int minus(int a, int b); //service local uniquement  
}
```

## La classe

```
@Stateless // ou bien par ex. @Stateless(name=calculette)  
public class CalculBean implements CalculRemote, CalculLocal{  
    public int add(int a, int b){ return a + b; }  
    public int minus(int a, int b){ return a - b; }  
}
```

# Les EJB Session stateful

## Caractéristiques

- Maintient d'états entre deux invocations d'un même client
  - 1 instance dédiée à chaque client
  - Expiration au bout d'un délai d'inactivité
- **Attention** : l'état n'est pas persistant  
⇒ données perdues à la destruction du bean ou à l'arrêt du serveur

## L'annotation @Stateful

S'utilise sur une classe Java et possède des attributs optionnels :

- `String name` : nom de l'EJB (défaut = nom de la classe)
- `String description` : description de l'EJB

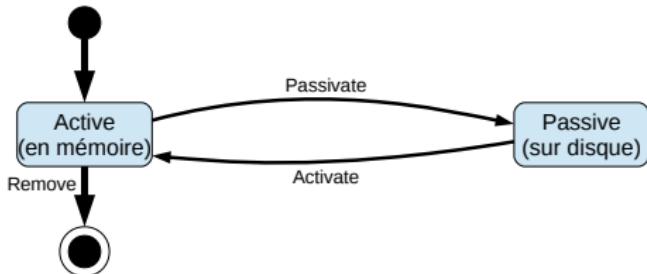
## L'annotation de méthode @Remove

Indiquer un traitement à faire en fin de session.

# Les EJB Session stateful : activation/désactivation

## Principe

Le conteneur peut décider sérialiser/désérialiser sur disques des EJB Stateful pour gérer son espace mémoire  
⇒ Similaire au mécanisme de swap d'un OS



## Les annotations de méthodes de la classe (callback événementiel)

- @PostConstruct, @PreDestroy
- @PostActivate , @PrePassivate
- @Remove

## Les EJB Session stateful : un exemple

```
@Stateful
public class CartBean implements CartRemote{
    private List<Object> items = new ArrayList<>();
    private List<Long> quantities = new ArrayList<>();

    public void addItem(int res, int qte){ .... }
    public void removeItem(int ref){....}

    @Remove
    public void confirmOrder(){ ... }
}
```

# Les EJB Session singleton

## Caractéristiques

- Stateful ayant une seule instance pour tous les clients
- Apports :
  - Exécution de code au lancement ou à l'arrêt de l'application
  - Partage de données avec gestion des accès concurrents
- **Attention** : état conservé durant toute la durée de vie de l'application  
⇒ données perdues à la fin de l'application ou de la JVM

## Les annotations sur la classe

- `@Singleton` : déclaration de la classe comme singleton
- `@Startup` : si présent, la classe sera instanciée au lancement de l'application  
**Attention** : n'impose pas d'ordre de lancement
- `@DependsOn` : permet de spécifier une dépendance avec d'autres EJB.  
⇒ assure que le conteneur démarrera les dépendances avant

## EJB Session singleton : exemple

```
@Singleton
@Startup
@DependsOn({"MonAutreBean"})
public class MonCache implements MonCacheRemote{
    private Map<String, Object> cache;

    @PostConstruct
    public void init(){
        this.cache=new HashMap<String, Object>();
    }
    @Override
    public Object get(String k){ return cache.get(k);}

    @Override
    public Object put(String k, Object v){ return cache.put(k,v);}

    @PreDestroy
    public void fin(){System.out.println("Requiem\u00d7Cachus");}
}
```

## Stratégies Container Managed Concurrency (par def.)

Le conteneur gère les accès concurrents via l'annotation `@Lock`

- Chaque méthode possède un verrou de type read ou write
  - read : accessible par plusieurs thread simultanément
  - write (par def.) : méthode en exclusion mutuelle
- Si l'annotation est sur la classe alors ceci agit comme valeur par défaut sur toutes les méthodes de la classe.

## Stratégies Bean Managed Concurrency

Accès concurrents à la charge du développeur (`synchronized`, `volatile`, ...)

La stratégie est précisée avec l'annotation de classe `@ConcurrencyManagement`

# EJB Session singleton : gestion de la concurrence (exemple)

```
@Singleton  
@Startup  
@DependsOn({"MonAutreBean"})  
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)  
@Lock(LockType.READ)  
public class MonBean implements MonbeanRemote{  
  
    @Lock(LockType.WRITE)  
    public void f(){...}  
  
    public void g(){....} //verrou en lecture  
}
```

# Nommer les EJB session avec JNDI

## Attribution des noms

Tout EJB session est enregistré dans un annuaire avec un nom unique attribué automatiquement par le conteneur au moment du déploiement

## Un format standard quelque soit le serveur d'application

`java:global/<app_name>/<module_name>/<bean_name>!<interface_name>`

- `app_name` : nom de l'application JavaEE dans laquelle le module de l'EJB est packagé
- `module_name` : nom du module dans lequel l'EJB est packagé
- `bean_name` : nom du bean (attribut *name* des annotations d'EJB session)
- `interface_name` : nom de l'interface exposée (interface remote)

# Accès à un EJB session par un client (hors conteneur)

## Principe

- Initialiser un contexte JNDI
- Récupérer une instance de mandataire d'une instance d'EJB Session avec lookup
- Caster l'instance dans le type de l'interface voulue (remote ou locale)

```
public class ClientEJB{  
    public static void main(String args[]){  
        Context context = new InitialContext();  
        String url="java:global/mon_appli/mon_module/Foo!FooRemote";  
        Object o = context.lookup(url);  
        FooRemote foo = (FooRemote) o;  
    }  
}
```

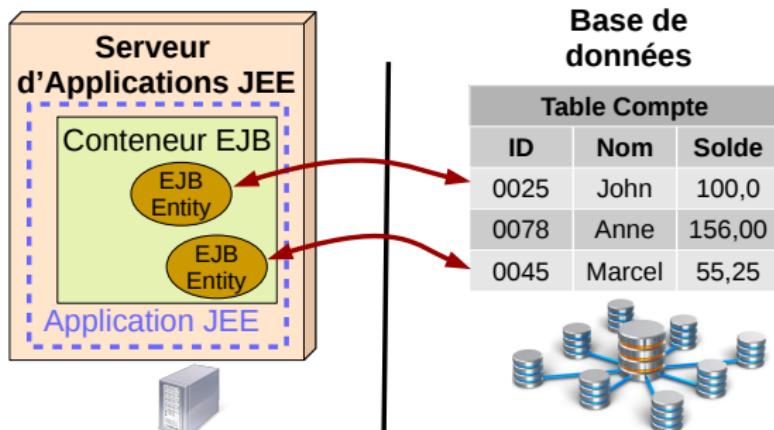
Si le client n'est pas sur la même machine

```
Properties p = new Properties();  
p.setProperty("org.omg.CORBA.ORBInitialHost", "<host_serv>");  
p.setProperty("org.omg.CORBA.ORBInitialPort", "<port_serv>");  
Context context = new InitialContext(p);
```

# Les EJB Entity

## Caractéristiques

- Représentation d'une donnée manipulée par l'application
  - Stockée sur support persistant accessible via JDBC (ex : SGBD)
  - Permet une correspondance objet Java et Tuple relationnel
  - Possibilités de définir des clés, des relations, des recherches  
⇒ on manipule des objets Java, plutôt que des requêtes SQL
- Mis en œuvre grâce aux annotations et à Java Persistence API (JPA)



## Java Persistence API (JPA)

- Mapping Obj/Rela avec une abstraction plus élevée que JDBC
  - des POJO annotés
  - un gestionnaire d'entités qui assure la correspondance Obj/Rela
    - création/modif/suppression d'objet vers la BD et vice-versa

## Codage d'une entity

Classe POJO (implantant généralement `Serializable`) qui :

- doit avoir un constructeur sans argument
- doit être annoté avec `@Entity`
  - argument optionnel `name` pour préciser le nom de l'entité dans les requêtes (défaut = nom de la classe)
- Avoir au moins une propriété l'annotation `@Id` (clé primaire)
  - ⇒ S'associe à l'attribut ou bien à son getter
  - ⇒ Types autorisés : primitifs (et type objet correspondant) `String` et `Date`

## Annotation de classe

- `@Table(name=". . .")` : précise le nom de la table concernée du SGBD  
Par def. : l'entité est liée à la table de la BD ayant le même nom de la classe

## Annotation de champs

- `@Column(name=". . .")` : Associe le champs de la table à la propriété  
Par def. : les champs de l'entité sont liées aux champs de la table ayant le même nom
- `@Transient` : Ne pas tenir compte du champs pour le mapping
- `@Id` : Définie le(s) champs de clé primaire
- `@GeneratedValue` : générer automatiquement la clé primaire.
  - attribut `strategy` : TABLE, SEQUENCE, IDENTITY ou AUTO (par def.)
  - attribut `generator` : Nom du générateur à utiliser

## EJB Entity : exemple

```
@Entity
public class Compte implements Serializable{

    private static final long serialVersionUID=1L;

    @Id
    @GeneratedValue
    private long id;
    private String nom;
    private double value;

    public Compte() {}

    public long getId(){return id;}
    public void setId(long id){this.id=id;}
    public long getNom(){return nom;}
    public void setNom(String nom){this.nom=nom;}
    public long getValue(){return value;}
    public void setValue(double value){this.value=value;}
}
```

## Principe

Avoir une clé primaire sur plusieurs champs

## Comment faire ?

- 1) Définir une classe encapsulant l'ensemble des champs de clé primaire :
  - doit implanter Serializable
  - doit posséder un constructeur sans argument
  - redéfinir les méthodes equals() et hashCode()
- 2) Deux solutions pour la relier cette classe à l'entité :
  - Annotation avec `@IdClass`
  - Annotation avec `@Embeddable` et `@EmbeddedId`

**Impossible de demander la génération automatique de clé primaire composée.**

## Clé primaire à plusieurs colonnes : exemple avec @IdClass

```
public class NomPrenom implements Serializable{
    private String nom;
    private String prenom;

    public NomPrenom(){}
    // + getter et setter sur nom et prenom
    public boolean equals(Object o){ ...}
    public int hashCode(){return (nom + prenom).hashCode();}
}
```

```
@Entity
@IdClass(NomPrenom.class)
public class Compte implements Serializable{
    @Id
    private String nom;
    @Id
    private String prenom;
    private double value;
    ...
}
```

Clé primaire à plusieurs colonnes :  
exemple avec @Embeddable et @EmbeddedId

```
@Embeddable
public class NomPrenom implements Serializable{
    private String nom;
    private String prenom;

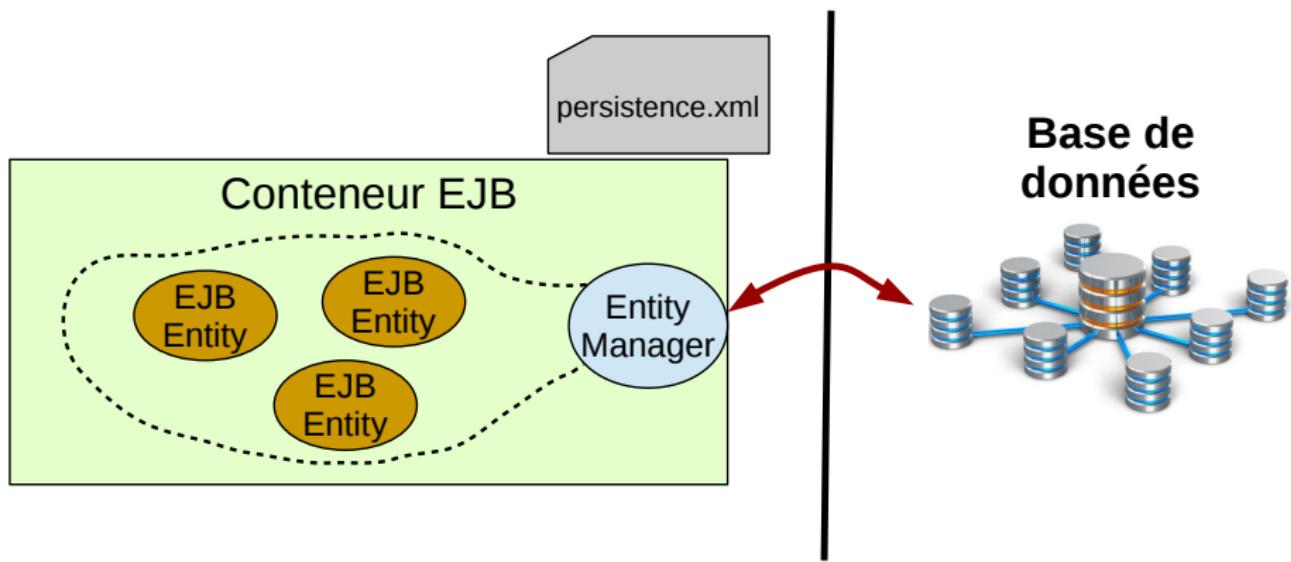
    public NomPrenom(){}
    // + getter et setter sur nom et prenom
    public boolean equals(Object o){ ...}
    public int hashCode(){return (nom + prenom).hashCode();}
}
```

```
@Entity
public class Compte implements Serializable{
    @EmbeddedId
    private NomPrenom id;
    private double value;
    ...
}
```

# Utiliser les EJB entity

## Problème

Comment assurer la correspondance entre les instances d'entité et la base de données ? ⇒ Utilisation d'un gestionnaire d'entités



## Caractéristiques

- Objet qui assure l'interaction entre des EJB entity et la BD :
  - lecture
  - recherche
  - mise à jour : ajout, modification, suppression
- Associé à un contexte de persistance qui :
  - défini comment se connecter à la BD
  - liste les classes d'entités gérées
  - est configuré dans un fichier *persistence.xml* propre au conteneur d'EJB
- Peut s'adresser à la BD via un gestionnaire de transaction (Commit/rollback)

# Mise à jour des données sur la BD

## ATTENTION

Les mises à jour (ajout, modif, suppression) sur la BD ne sont pas immédiates

## Deux modes de répercutions

- mode AUTO : māj reportées automatiquement dans la BD avant chaque requête
- mode COMMIT : māj effectuées lors du commit de la transaction
  - ⇒ moins d'échange avec la BD donc plus performant
  - ⇒ alourdi le code

## Forcer la synchronisation

- flush : force l'écriture de l'entité sur la BD
- refresh : remet à jour l'entité à partir de la BD

# Obtention d'une instance d'EntityManager

## En Java EE

- Ajout d'une injection de dépendance :
  - ⇒ Déclaration d'un attribut de type EntityManager
  - ⇒ Attribut annoté par @PersistenceContext(unitName="...")
- Le conteneur gère automatiquement :
  - l'instanciation de l'attribut
  - l'appel à la méthode close().

```
@Stateless
public class MyBean implements MyBeanRemote{

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

}
```

## Principe

- Instancier l'occurrence de l'entité (`new`)
- Initialiser les propriétés de l'entité (setters)
- Si besoin définir les relations avec d'autres entités
- Utiliser la méthode `persist()` du gestionnaire

Mode AUTO

```
...
public void addCompte(
    String nom,
    String , prenom){
    Compte c = new Compte();
    c.setNom(nom);
    c.setPrenom(prenom);
    c.setValue(0.0)
    em.persist(c);
}
```

Mode COMMIT

```
...
public void addCompte(
    String nom,
    String , prenom){
    Compte c = ....
    EntityTransaction t;
    t = em.getTransaction();
    t.begin();
    em.persist(c);
    t.commit();
}
```

# Rechercher une occurrence dans la BD (1/2)

## Via la classe EntityManager

- `T find(Class<T> c, Object id) :`  
renvoie une Entité de type T avec la clé primaire id  
renvoie `null` si non trouvé
- `T getReference(Class<T> c, Object id) :` même chose mais jette une exception `EntityNotFoundException` si non trouvé

Dans les deux cas :

- `c` doit être une classe d'entité.
- `id` doit correspondre au type de la clé primaire de `c`

```
public Compte getCompte(long id){  
    Compte c = em.find(Compte.class, id);  
    if(c != null){  
        System.out.println("Valeur = " + c.getValue())  
    }  
    return c;  
}
```

## Via des requêtes JPQL

- Repose sur
  - l'appel à la méthode `Query createQuery(String req)` du gestionnaire
  - le langage de requête Java Persistence Query Language.
- Execution de la requête via :
  - `Object getSingleResult()` : si un seul résultat attendu
  - `List getResultList()` : si on souhaite récupérer l'ensemble des résultats renvoyés par la requête

```
public Compte getCompte(long id){  
    String s = "SELECT c FROM Compte c"  
    Query q = em.createQuery(s);  
    Compte c = (Compte) q.getSingleResult();  
    if(c != null){  
        System.out.println("Valeur = "+c.getValue())  
    }  
    return c;  
}
```

# Modifier une occurrence

## Sur une entité déjà gérée par le gestionnaire

```
public void incr(long id){  
    Compte c = em.find(Compte.class, id);  
    c.setValue(c.getValue() + 1);  
    em.flush();  
}
```

## Par Fusion de deux entités

- Une entité A gérée par le gestionnaire
- Une entité B non gérée de même clé primaire que A
- la fonction `merge` qui modifie A en fonction des différences avec B

```
public void setCompte(Compte b){  
    Compte a = em.find(Compte.class, b.getId());  
    em.merge(b);  
}
```

# Supprimer une occurrence

## Principes

- Obtenir une instance sur l'entité à supprimer
- Appeler la fonction `remove`

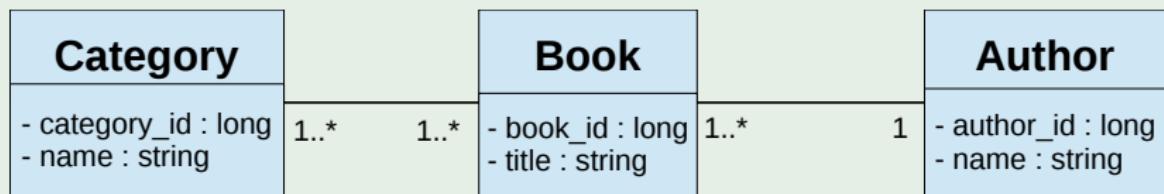
```
public void removeCompte(long id){  
    Compte compte = em.find(Compte.class, id);  
    em.remove(compte);  
}
```

# Lien relationnel entre entité

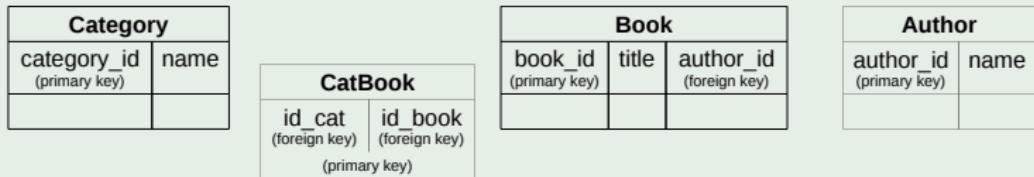
## Problème

Comment respecter le schéma relationnel d'une base de données entre les différentes entités ?

## Exemple



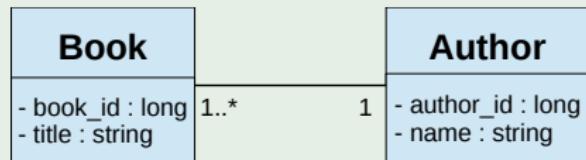
## Exemple : tables résultantes de la BD



# Relation 1-N

## La classe Author

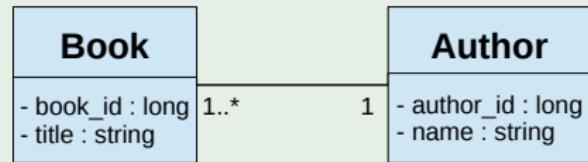
```
@Entity  
public class Author{  
    @Id  
    private long id;  
    private String name;  
    private Collection<Book> books;  
  
    public Author(){ books= new ArrayList<Book>();}  
  
    @OneToMany(mappedBy="author")  
    //nom de l'attribut referencant l'auteur dans la classe Book  
    public Collection<Book> getbooks(){return books;}  
    //et autres getter/setter  
}
```



# Relation 1-N

## La classe Book

```
@Entity  
public class Book{  
    @Id  
    private long id;  
    private Author author;  
    private String title;  
  
    public Book(){}
  
  
    @ManyToOne  
    @JoinColumn(referencedColumnName="id")// (id de l'author)  
    public Author getAuthor(){return author;}  
    ....  
}
```

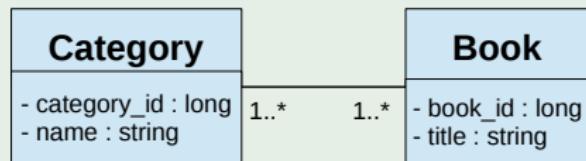


# Relation N-N

## La classe Category

```
@Entity
public class Category{
    @Id
    @Column(name="category_id")
    private long id;

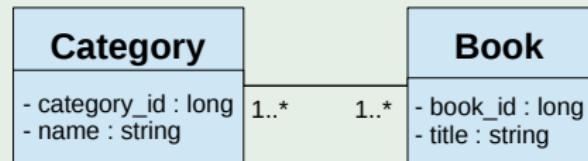
    @ManyToMany
    @JoinTable(name="CatBook",
        joinColumns=@JoinColumn(
            name="id_cat",
            referencedColumnName="category_id"),
        inverseJoinColumns=@JoinColumn(
            name="id_book",
            referencedColumnName="book_id"))
    )
    private Set<Book> books;
    ...
}
```



# Relation N-N

## La classe Book (suite)

```
@Entity
public class Book{
    @Id
    @Column(name="book_id")
    private long id;
    private Author author;
    private String title;
    ...
    @ManyToMany(mappedBy="books")
    private Set<Category> categories;
}
```



## Objectif

Utiliser le conteneur pour assurer l'injection de dépendances de certaines ressources requises :

- utilisation d'annotation sur des attributs
  - l'invocation de l'annuaire du serveur via JNDI + cast fait automatiquement par le conteneur
- ⇒ développement simplifié

## Annotations indiquant une dépendance

- `@EJB` : référence vers un autre EJB
- `@Ressource` : dépendance vers une ressource externe (DataSource JDBC, destination JMS, ...)
- `@PersistenceContext` : dépendance vers un EntityManager
- `@WebServiceRef` : dépendance vers un service web

## Dépendance @EJB

Attributs :

- Class beanInterface : objet class de l'interface de l'EJB
- String beanName : nom de l'EJB (paramètre name de @Stateful ou @Stateless)

```
@Stateless
public class FooBean implements FooRemote{
    @EJB
    private CalculLocal calcul;
}
```

# Message-driven Bean (MDB)

## Principe

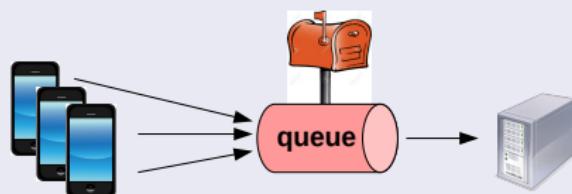
Interaction par envoi de message asynchrone (en se basant sur JMS)



## Deux modes de communication

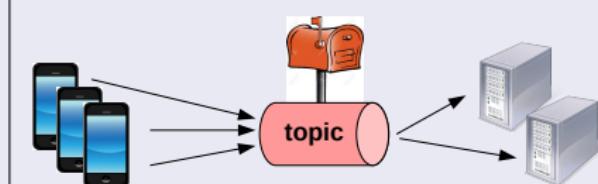
### queue

$N$  clients vers 1 serveur



### topic

$N$  clients vers  $M$  serveurs



# Message-driven Bean (MDB)

## Caractéristiques

- consomme des messages asynchrones
- pas d'état ( $\equiv$  EJB session stateless)
- Toutes les instances d'une même classe de MDB sont équivalentes
- peut traiter les messages de clients différents

## Quand les utiliser ?

- éviter les appels bloquants
- découpler client et serveur
- besoin de fiabilité : protection crash du serveurs

## Notion de producteur/consommateur

## Types d'objets manipulés

- ConnectionFactory : créer des connexion vers 1 queue/topic
- Connection : représente une connexion vers 1 queue/topic
- Session : période de temps pour l'envoi de messages dans 1 queue/topic
- Destination : représente 1 queue/topic

# MDB : mise en œuvre producteur

```
@Stateless
public class HelloMDB implements HelloMDBRemote {
    @Resource(mappedName="jms/QueueConnectionFactory")
    private static ConnectionFactory factory;

    @Resource(mappedName="jms/Queue")
    private static Destination dest;
    public HelloMDB() { }

    public void disBonjour() {
        Connection connection = factory.createConnection();
        Session session =
            connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        MessageProducer producer = session.createProducer(dest);
        TextMessage message = session.createTextMessage("Coucou");
        producer.send(message);
        session.close();
        connection.close();
    }
}
```

# MDB : mise en œuvre consommateur

```
@MessageDriven(mappedName="jms/Queue", activationConfig = {  
    @ActivationConfigProperty(propertyName = "acknowledgeMode",  
        propertyValue = "Auto-acknowledge"),  
  
    @ActivationConfigProperty(propertyName = "destinationType",  
        propertyValue = "javax.jms.Queue"),  
  
    @ActivationConfigProperty(propertyName = "destination",  
        propertyValue = "jms/Queue")  
})  
public class SimpleMessageBean implements MessageListener {  
    @Resource  
    private MessageDrivenContext mdc;  
  
    public SimpleMessageBean() {}  
  
    public void onMessage(Message message) {  
        System.out.println("MESSAGE RECU");  
    }  
}
```

# Invocation asynchrone des EJB session

## Comment rendre une invocation asynchrone ?

- via un thread
  - ⇒ API INTERDITE pour JavaEE
- via un EJB orienté message
  - ⇒ peu adapté à l'appel de méthodes
  - ⇒ difficile d'avoir un retour sur le traitement

## L'annotation @Asynchronous

- applicable sur :
  - une méthode ⇒ indique qu'elle est invocable de manière asynchrone
  - une classe ou interface ⇒ toutes les méthodes sont asynchrones
- Contrainte sur le retour de la méthode :
  - soit `void` : aucun retour connu du client
  - soit `Future<T>` : le client peut avoir un contrôle sur l'état de l'exécution et obtenir la valeur de retour ou le déclenchement d'une exception

# Définition d'un service asynchrone

```
@Stateless  
@LocalBean  
public class BeanAsynchrone implements BeanAsynchroneLocal {  
    public BeanAsynchrone(){}  
  
    @Override  
    @Asynchronous  
    public Future<Boolean> isPairAsync(int i) {  
        Thread.sleep(10000); //simule un long traitement  
        return new AsyncResult<Boolean>(i%2==0);  
    }  
}
```

## Le type java.util.concurrent.Future<T>

- `boolean cancel(boolean)` : tenter d'annuler un traitement
- `boolean isDone()` : tester si le traitement est terminé
- `boolean isCancelled()` : tester si le traitement a été annulé
- `V get()` : obtenir le résultat final du traitement (**bloquant**)

## Invocation d'un service asynchrone : exemple

```
@Stateless
@LocalBean
public class BeanClient implements BeanClientRemote {
    public BeanClient(){}
    
    @EJB
    private BeanAsynchrone ejb;
    
    @Override
    public Boolean isPair(int i) {
        Future<Boolean> res = ejb.isPairAsync(i);
        
        //ON peut faire autre chose en attendant le resultat final
        
        return res.get();
    }
}
```

# Les Timer

## Intérêt

Permettre de déclencher régulièrement ou périodiquement une fonctionnalité

## Deux choses à définir

- une méthode de callback définissant un traitement à exécuter
- une date ou une période de déclenchement

```
@Singleton  
@localBean  
@Startup  
public class Periodique{  
    @Schedule(/*attributs (cf. apres)*/)  
    public void callbackA(){ /*traitement A*/ }  
  
    @Schedule(/*attributs (cf. apres)*/)  
    public void callbackB(){ /*traitement B*/ }  
}
```

# Les Timer : l'annotation @Schedule

## Définition

Permet de déclarer un timer auprès du conteneur

## Huit attributs

nom attribut	Valeurs possibles
hour	"0" à "23"
minute	"0" à "59"
second	"0" à "59"
dayOfMonth	"1" à "31" : jour du mois {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"} : un jour précis "-1" à "-7" : nombre de jours avant la fin du mois
dayOfWeek	"0" à "7" : jour de la semaine (0 = 7 = dimanche) {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"}
Month	"1" à "12" : le mois de l'année {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"}
Year	année sur 4 chiffres

# Les Timer : l'annotation @Schedule

## Valeurs possibles des attributs

- valeur simple (ex : hour="20")
- toutes les valeurs possibles (ex : minute="\*")
- liste (ex : dayofWeek="Mon,Wed,Thu")
- plage (ex : year="2015-2019")
- incrémentation x/y = incrémentation de y unités à partir de la valeur x jusqu'à valeur max (ex : minute="\*/10")

## Exemples

Le 1er de chaque mois à 6h du matin	@Schedule(hour="6", dayOfMonth="1")
du lundi au vendr. à 10h du soir	@Schedule(dayOfWeek="Mon-Fri", hour="22")
Tous les vendredi à 22h30	@Schedule(dayOfWeek="Fri",hour="22",minute="30")
Toutes les heures de chaque lundi	@Schedule(dayOfWeek="Mon", hour="*")
trois jours avant la fin du mois à 23h	@Schedule(dayOfMonth="-3", hour="23")
tous les quart d'heure à partir de midi	@Schedule(minute="*/15", hour="12/1")

## Définition

- Succession d'unités de traitements utilisant une ou plusieurs ressources tout en assurant leur intégrité
- 4 propriétés fondamentales :
  - **Atomicité** : toutes les instructions s'exécutent correctement ou aucune ne s'exécute
  - **Cohérence** : l'état du système reste valide à chaque transaction
  - **Isolation** : toute mise à jour de données pendant transaction ne doivent pas être modifiées en dehors de la transaction
  - **Durabilité** : une transaction confirmée demeure enregistrée

## Mise en œuvre avec les EJB

- Une transaction concerne l'intégralité des traitement d'une méthode
- Deux modes de gestions des commit/rollback :
  - soit par le bean : à coder explicitement dans la méthode avec l'API JTA
  - soit automatiquement par le conteneur (mode par def.)

# Gestion des transaction via JTA

## Principe

Démarcation explicite avec begin/commit/rollback

```
@Stateless  
@TransactionManagement(TransactionManagementType.BEAN)  
public class MyBean implements MyBeanRemote {  
  
    @PersistenceContext private EntityManager em;  
  
    @Resource private UserTransaction ut;  
  
    public void transfert(){  
        try{  
            ut.begin();  
            Account a1 = em.find(Account.class, "Bob");  
            Account a2 = em.find(Account.class, "Anne");  
            a1.credit(10.5);  
            a2.debit(10.5);  
            ut.commit();  
        } catch(Exception e){ ut.rollback();}  
    }  
}
```

# Gestion des transaction par le conteneur

## Principe

Toute méthode est considérée comme un bloc transactionnel (Commit en fin de méthode)

```
@Stateless  
@TransactionManagement(TransactionManagementType.CONTAINER)  
public class MyBean implements MyBeanRemote {  
  
    @PersistenceContext private EntityManager em;  
  
    @Resource private SessionContext sc;  
    @TransactionAttribute(/*cf. plus loin*/)  
    public void transfert(){  
        try{  
            Account a1 = em.find(Account.class, "Bob");  
            Account a2 = em.find(Account.class, "Anne");  
            a1.credit(10.5);  
            a2.debit(10.5);  
        }catch(Exception e){ sc.setRollbackOnly();}  
    }  
}
```

# L'annotation @TransactionAttribute

## Caractéristiques

- Permet de préciser dans quel contexte transactionnel une méthode d'un EJB sera invoquée
- 6 possibilités :
  - REQUIRED
  - REQUIRES\_NEW
  - SUPPORTS
  - NOT\_SUPPORTED
  - MANDATORY
  - NEVER

## 2 cas pour le bean appelant

- soit il s'exécute dans une transaction
- soit il s'exécute en dehors de tout contexte transactionnel

# Granularité des transaction : REQUIRED

## Principe

Si l'appelant s'exécute dans une transaction, l'appelé s'y s'insère  
Sinon, l'appelé débute une nouvelle transaction

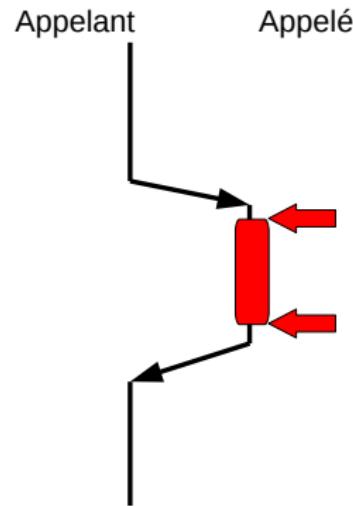
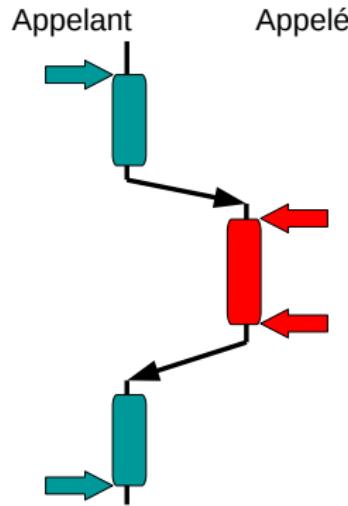


# Granularité des transaction : REQUIRES\_NEW

## Principe

Une nouvelle transaction est systématiquement créée

Si l'appelant s'exécute dans une transaction, l'appelé y imbrique la sienne



# Granularité des transaction : SUPPORTS

## Principe

Si l'appelant s'exécute dans une transaction, l'appelé s'y s'insère  
Sinon, l'appelé s'exécute en dehors de toute transaction.



# Granularité des transaction : NOT\_SUPPORTED

## Principe

L'appelé s'exécute toujours en dehors d'une transaction.



# Granularité des transaction : MANDATORY

## Principe

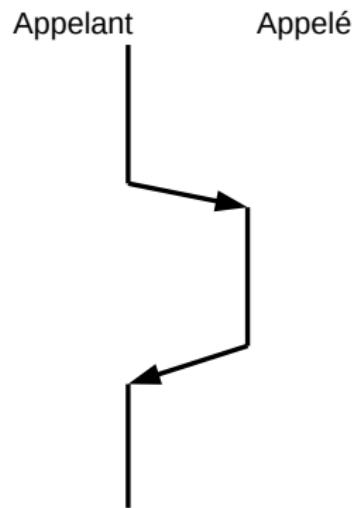
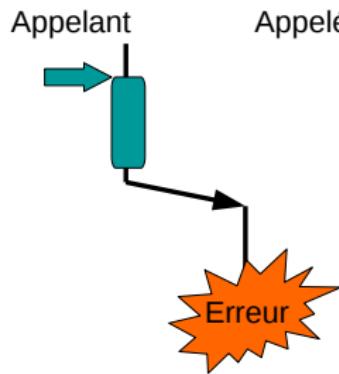
Si l'appelant s'exécute dans une transaction, l'appelé l'utilise  
Sinon, l'appelé lève une exception.



# Granularité des transaction : NEVER

## Principe

L'appelant ne doit pas s'exécuter dans une transaction.



# EJB Lite : version allégée des EJB

Feature	EJB Lite	EJB
Stateless beans	✓	✓
Stateful beans	✓	✓
Singleton beans	✓	✓
Message driven beans	✗	✓
No interfaces	✓	✓
Local interfaces	✓	✓
Remote interfaces	✗	✓
Asynchronous invocation	✗	✓
Interceptors	✓	✓
Declarative transaction	✓	✓
Programmatic transaction	✓	✓
Timer Service	✗	✓
EJB 2.x support	✗	✓
CORBA interoperability	✗	✓