Java Enterprise Edition
EJB3 / Deuxième partie

Matthieu EXBRAYAT
Master 2 RIA
Université Louis Pasteur

Plan

- EJB Entity / suite
- Message Driven Bean
- Cycle de vie et callbacks
- Transactions et sécurité
- Intercepteurs
- Timers

EJB Entity : la suite

- Clés simples et clés composites
- Héritage
- Relations entre entités
- EJB QL
- Compléments sur le mapping
- Tables existantes / tables créées

Classe de clés

- Pas nécessaire en général pour les types simples
  - Peut harmoniser, mais allourdit
  - Mapping automatique des types simples
- Utile pour les clés composites
- Mise en place
  - Création d'une classe simple, mais sérializable, avec constructeur sans paramètres et avec hachage et égalités bien définis
  - @IdClass

Exemple : la classe clé

```
public class EmployePK implements java.io.Serializable {
private String nom;
private String prenom;
public EmployePK(){
}
public EmployePK(String nom,String prenom){
this.nom=nom;this.prenom=prenom;
}
// getters / setters
...
public boolean equals(Object obj) {...}
public int hashCode(){...}
```

Exemple : la classe entity

```
@Entity
@IdClass(EmployePK.class)
public class Employe implements java.io.Serializable{
        private String nom;
        private String prenom;
        private int age;

        @Id
        public String getNom(){...}
        public String setNom(String nom){...}
        @Id
        public String getNom(){...}
        public String setNom(String nom){...}
...
```

Autre approche : encapsulation

```
@Embeddable
public class EmployePK implements java.io.Serializable {
private String nom;
private String prenom;
public EmployePK(){
}
public EmployePK(String nom,String prenom){
this.nom=nom;this.prenom=prenom;
}
// getters / setters
...
public boolean equals(Object obj) {...}
public int hashCode(){...}
```

Côté classe

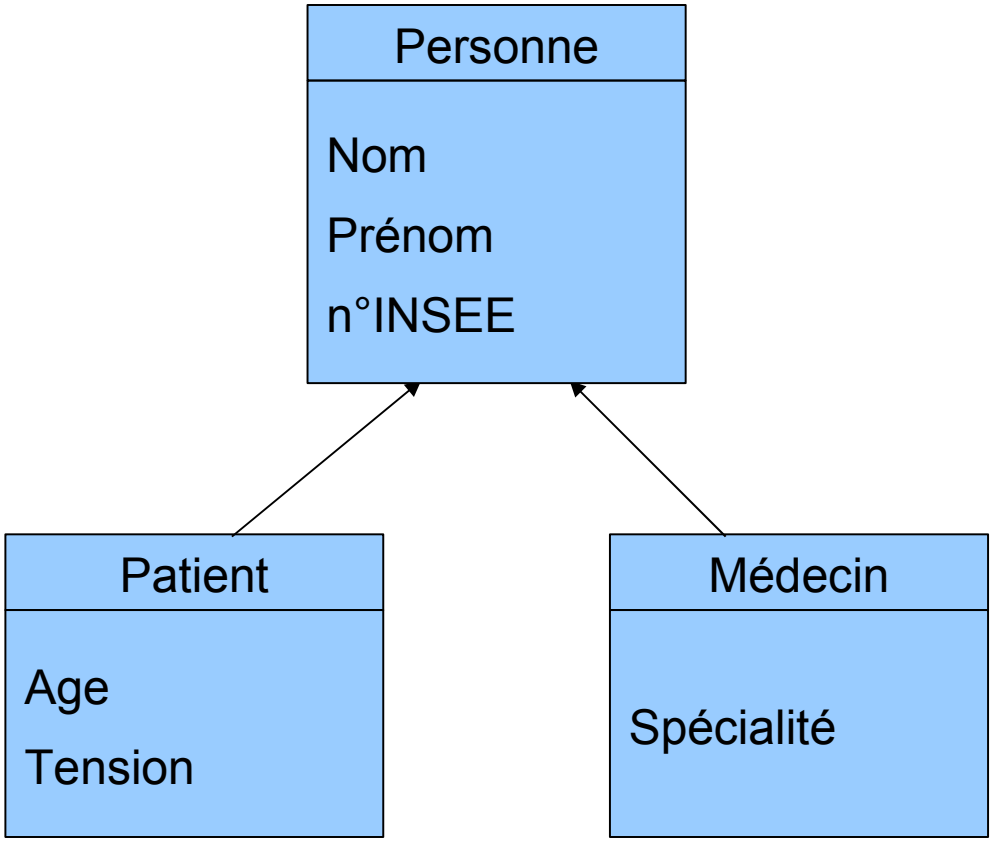
```
@Entity
public class Employe implements java.io.Serializable{
        private EmployePK pk;
        private int age;

        @EmbeddedId
        public String getPk(){...}
        public String setPk(EmployePK pk){...}
...
```

Héritage

- Possible (et simple) dans les entity EJB3.0
- Trois approches
  - Table unique
  - Table par classe
  - Table par attributs spécifiques
- Cas particulier
  - Superclasse non persistante...

Exemple


Personne
Nom
Prénom
n°INSEE
Patient
Age
Tension
Médecin
Spécialité

Solution Table Unique (1)

```
@Entity
@Table(name= « PERSONNE »)
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name= « DISCR »,
        discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue(« PERSONNE »)
public class Personne {
        private String insee;
        private String nom; // Uniquement attributs de Personne
        ...
}
```

Solution Table Unique (2)

```
@Entity
@DiscriminatorValue(«PATIENT»)
public class Patient extends Personne {
    private int age;
    private int[] tension;
    ...
}
```

```
@Entity
@DiscriminatorValue(«MEDECIN»)
public class Medecin extends Personne {
    private String specialite;
    ...
}
```

Solution Table par Classe

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Personne {
        private String nom; // Uniquement attributs de Personne
        ...
}
```

```
@Entity
public class Patient extends Personne{
        ...
}
```

```
@Entity
public class Medecin extends Personne {
        ...
}
```

Solution Multitables + Jointures

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED
public class Personne {
        private String nom; // Uniquement attributs de Personne
        ...
}
```

```
@Entity
public class Patient extends Personne {
        ...
}
```

```
@Entity
@PrimaryKeyJoinColumn(name=« MED_PK »)
public class Medecin extends Personne{
        ...
}
```

Comparaison des 3 approches

- Table unique
  - + : simple, efficace (chargement)
  - - : normalisation, attributs des sous classes doivent être *nullables*
- *Une table par classe*
  - *+ : contraintes possibles, adéquation base pré-existante*
  - *- : normalisation, colonnes redondantes, relations polymorphiques ?*
- *Eclatement entre tables + jointures*
  - *+ : contraintes possibles, normalisé, adéquation base pré-existante*
  - *- : coût reconstruction*

Superclasse non persistée

```
@MappedSuperclass
public class Personne {
        private String id;
        ...
        @Id
        public String getId (){...}
        ...
}
```

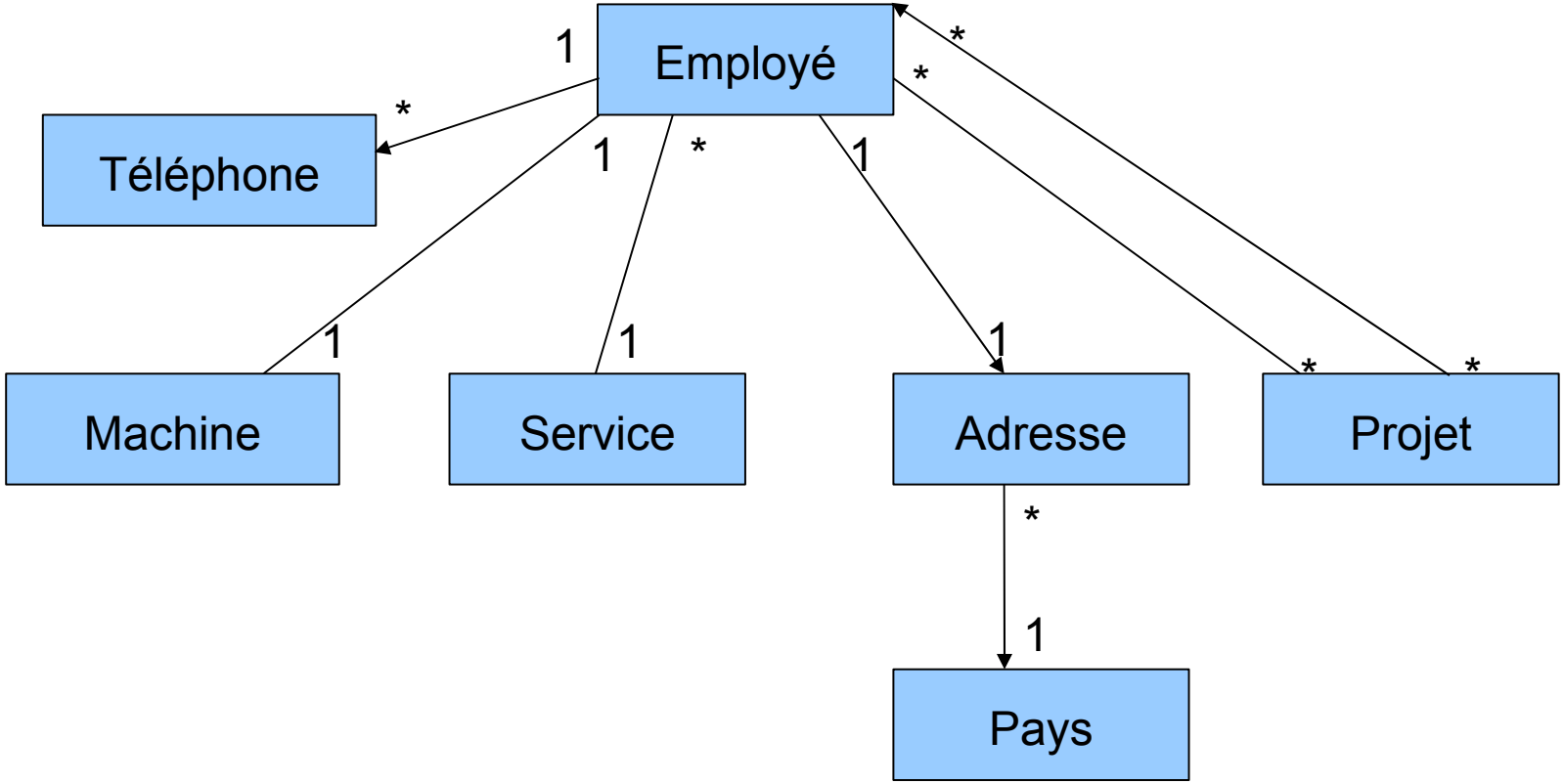
```
@Entity
@Inheritance(strategy=InheritanceType.JOINED »)
public class Patient extends Personne {
        ...
}
```

```
@Entity
@PrimaryKeyJoinColumn(name=« MED_PK »)
public class Medecin extends Personne{
        ...
}
```

Relations entre entités

- Il existe un mapping pour chaque type de relation existant entre deux classes
  - One-to-one mono directionnelle
  - One-to-one bidirectionnelle
  - One-to-many monodirectionnelle
  - Many-to-one monodirectionnelle
  - One-to-many bidirectionnelle
  - Many-to-many monodirectionnelle
  - Many-to-many bidirectionnelle

Exemple



One-to-one MD

```
@Entity
public class Employe implements java.io.Serializable {
...
private Adresse adrEmp;
...
@OneToOne // création auto d'un attribut « addresse_id »
public Adresse getAdrEmp() {
        return this.adrEmp;
}
public void setAdrEmp(Adresse adrEmp) {
        this.adrEmp=adrEmp;
}
// Rien de particulier côté addresse
```

O2O MD : Colonne de jointure

- @JoinColumn(name= « ... »)
  - Crée une colonne de ce nom dans la table de Employe
- @PrimaryKeyJoinColumn
  - Même PK dans les deux tables
- Il y a (presque) toujours une solution automatique (y compris en O2M et M2M)...

One-to-one BD

```
@Entity
public class Machine ... {
...
private Employe emp;
...
@OneToOne(mappedBy=« machEmp »)
public Employe getEmp(){
return emp;
}
public void setEmp (Employe emp) {
this.emp=emp;
}
```

```
@Entity
public class Employe... {
...
private Machine machEmp;
...
@OneToOne
@JoinColumn(...) // Optionnel
public Machine getMachEmp() {
return machEmp;
}
public void setMachEmp(Machine machEmp) {
...
}
```

Gestion des relations

- L'EM ne se pose pas beaucoup de questions...
- Ce sont des POJO...
- **Il faut faire à la main les sets des deux côtés !**
- Employe emp=new Employe();

  Machine m =new Machine();

  m.setEmp (emp);

  emp.setMachEmp(m);

One-to-Many MD

```
@Entity
public class Employe ... {
...
private Collection<Telephone> tels=
                new ArrayList<Telephone>();
...
@OneToMany(cascade={CascadeType.ALL})
@JoinColumn(name= « EMP_ID »)
public Collection<Telephone> getTels(){
...
}
public void setTels(Collection<Telephone> tels) {
...
}
```

```
@Entity
public class Telephone ... {
...
// Ne voit pas Employe !
```

Many-to-One MD

```
@Entity
public class Adresse ... {
...
Pays p;
...
@ManyToOne
@JoinColumn(name= « PAYS_ID »)
public Pays getPays() {
...
}
public void setPays(Pays p){
...
}
```

```
@Entity
public class Pays ... {
...
```

One-to-Many BD

```
@Entity
public class Employe ... {
...
private Service serv;
...
@ManyToOne
@JoinColumn(name= « SERV_ID »)
public Service getService() {
...
}
public void setService(Service serv){
...
}
```

```
@Entity
public class Service ... {
...
private Collection<Employe> emps=
        new ArrayList<Employe>();
...
@OneToMany(mappedBy= « serv »)
public Collection<Employe>getEmps(){
...
}
public void setEmps(Collection<Employe>emps){
...
}
```

Many-to-Many Bidirectionnel

```
@Entity
public class Projet ... {
...
private Collection<Employe> emps=
        new ArrayList<Employe>();
...
@ManyToMany
@JoinTable(name= « EMP_PROJ »;
        joinColumns={@JoinColumn(name= « PROJET_ID »),
                @InverseJoinColumn={@JoinColumn(name= « EMP_ID »)})
public Collection<Employe> getEmps() {
...
}
public void setEmps(Collection<Employe> emps) {
...
}
```

M2M BD (suite)

```
@Entity
public class Employe ... {
...
private Collection<Projet> proj=
        new ArrayList<Projet>();
...
@ManyToMany(mappedBy=  « emps »);
public Collection<Employes> getEmps() {
...
}
...
```

Many-to-Many Monodirectionnel

```
@Entity
public class Projet ... {
...
private Collection<Employe> histoEmps=
        new ArrayList<Employe>();
...
@ManyToMany
@JoinTable(name= « HISTO_EMP_PROJ »;
        joinColumns={@JoinColumn(name= « PROJET_ID »),
                @InverseJoinColumn={@JoinColumn(name= « EMP_ID »)})
public Collection<Employe> getHistoEmps() {
...
}
public void setHistoEmps(Collection<Employe> histoEmps) {
...
}
```

Côté Employé ?

Actions en cascade

- Particulièrement justifiée en O2O MD
- Disponible pour toutes relations
- @*TypeRelation*(CascadeType.*XXX*)
  - ALL, PERSIST, MERGE, REMOVE, REFRESH

ORDER BY

- Type List permet tri
- @OrderBy(« *attribut* ASC | DESC »)
- Tri possible sur plusieurs attributs

Relation avec attributs

- Travers du concepteur « relationnel »...
- Du point de vue relationnel :
  - Une table avec clé composite + attributs
- Du point de vue objet :
  - Est-ce une relation entre classes ?
  - Solution ?

EJB QL

- Localiser des données persistentes ?
  - Via l'EM
  - Avec quel langage ?
- EJB QL
  - SQL-like
  - orienté objet (le recours aux jointures est limité)

API Query

```
package javax.persistence;
public interface Query {
   public List getResultList( );
   public Object getSingleResult( );
   public int executeUpdate( );
   public Query setMaxResults(int maxResult);
   public Query setFirstResult(int startPosition);
   public Query setHint(String hintName, Object value);
   public Query setParameter(String name, Object value);
   public Query setParameter(String name, Date value, TemporalType temporalType);
   public Query setParameter(String name, Calendar value, TemporalType temporalType);
   public Query setParameter(int position, Object value);
   public Query setParameter(int position, Date value, TemporalType temporalType);
   public Query setParameter(int position, Calendar value, TemporalType temporalType);
   public Query setFlushMode(FlushModeType flushMode);
}
```

Entity Manage API (requêtes)

```
Query    createQuery(String qlString)
Query    createNamedQuery(String name)

Query    createNativeQuery(String sqlString)
Query    createNativeQuery(String sqlString, Class resultClass)
Query    createNativeQuery(String sqlString, String resultSetMapping)
```

Requête : premiers pas

- Retour = 1 entity
  - Query q=em.createQuery(«from Employe e where e.id=123);
    Employe e = (Employe)q.getSingleResult();
  - EntityNotFoundException, NonUniqueResultException
- Retour = liste
  - Query q=em.createQuery(«from Employe e
    where e.prenom= «Jean»);
    List l = q.getResultList();
- IllegalStateException (RunTime)

Requête paramétrée

- Paramètre nommé
  - Query q=em.createQuery(« from Employe e where e.id=:id »);
    q.setParameter(« id »,123);
- Paramètre numéroté
  - Query q=em.createQuery(« from Employe e where e.id=?1 »);
    q.setParameter(1,123);

Sélection des résultats

- Nombre d'entity
  - q.setMaxResults(max);
- Premier entity
  - q.setFirstResult(index);
- Remarque « utile » : empilement d'invocation
  - q.setMaxResult(10).getResultList();

EJB QL

- Syntaxe complète
  - SELECT OBJECT (e) FROM Emplye AS e
- + simple
  - SELECT e FROM Employe e
- Encore + simple
  - FROM Employe e
- Notion de schéma abstrait (name=...)

Récupération d'attributs

- Attribut d'entité
  - Select c.prenom From Employe e
- Attribut de « relation » (cardinalité 1)
  - Select c.service.nomservice From Employe e
  - Select c.service.batiment.adresse From ...
- Type des retours ?

Retour d'objets construits

- Construction d'un objet dans SELECT
- Exemple : classe NomPers (nom,prenom)
  - Extraite de Employe
  - Select new NomPers(e.nom,e.prenom) From Employe e
- Intérêt ?

Sélection et cardinalité multi

- La récupération directe par chemin n'est possible qu'avec une cardinalité 1
- En cardinalité n : IN ou [INNER] JOIN
  - Select p From Employe e, IN (e.projet) p
  - Select p From Employe e JOIN e.projet p
  - Select p.intitule From Employe e JOIN e.projet p
- LEFT JOIN, FETCH JOIN, DISTINCT...

WHERE

- Clause WHERE similaire à SQL
  - LIKE, BETWEEN, =, AND, OR ...
  - [NOT] IN (liste valeurs) : sens classique
  - IS [NOT] NULL : attributs et paramètres
    - c.adresse IS NOT NULL
    - :age IS NOT NULL
  - IS [NOT] EMPTY : liste
  - [NOT] MEMBER OF : dans un liste (relation)

Expressions Fonctionnelles

LOWER(Sttring)

UPPER(String)

TRIM([[LEADING | TRAILING | BOTH] [Trim_char] FROM] String)

CONCAT(String1,String2)

LENGTH(String)

LOCATE(String1,String2 [,start])

SUBSTRING(String,start,length)

ABS(number)

SQRT(double)

MOD(int,int)

CURRENT_DATE

CURRENT_TIME

CURRENT_TIMESTAMP

Agrégation

- COUNT
- MAX, MIN
- AVG,SUM
- GROUP BY, HAVING

```
SELECT e.nom, COUNT(p)
FROM Employe e JOIN Projet
```

Et encore...

- ALL, ANY, SOME
- EXISTS
- UPDATE et DELETE
  - UPDATE Employe e SET e.salaire=e.salaire+10

    WHERE EXISTS (SELECT e From Employe WHERE ...)
- Quand avoir recours à UPDATE et DELETE ?
- Dangers ?

Requêtes natives

- Pour des appels directs en SQL (cas particuliers)
- createNativeQuery(String sql) : retours «simples»
- createNativeQuery(String sql, Class classe) : retour classe d'entité (complète !)
  - Query q=em.createNativeQuery(« SELECT nom, prenom,age From Employe », Employe.class)
- Possibilité de requêtes natives complexes ...

Requêtes Nommées

- Requêtes créées « à part »
  - @NamedQueries
  - @NamedQuery(name= « », query= « »)
  - Avant le début de la classe
- Autre possibilité : descripteur de déploiement
- Intérêt ?

Exemple

```
@NamedQueries({
      @NamedQuery(name= « salaireMoyen »,
                        query= « SELECT AVG(e.salaire) From Employe e
                                    WHERE e.age BETWEEN :age1 AND :age2 »)
})
@Entity
public class Employe {....}

Query q=em.createNamedQuery(« salairemoyen »);
q.setParameter(« age1 »,30);
q.setParameter(« age2 »,45);
```

Mapping des tables et colonnes

- Par défaut :
  - Nom des classes et attributs
  - Relations : concaténation
- A la main
  - @Table, @Column
  - Table préexistante ou table crée ?

@Column

- name
- unique
- nullable
- insertable, updatable
- columnDefinition (e.g. « integer »)
- table (mapping multi tables)
- length (chaine)
- precision, scale (numériques)

Génération de clé

- @TableGenerator
  - @TableGenerator (name= « nomgen »,
    table= « nomtable »,
    pkColumnName= « nomCol1 »,
    valueColumnName= « nomCol2 »,
    pkColumnValue= « maCle »)
  - @Id @GeneratedValue (strategy=GenerationType.TABLE,
    generator= « nomgen »)
- Une même table peut servir pour plusieurs clés >>

  pkColumnValue

Génération de clé (suite)

- @SequenceGenerator
  - @SequenceGenerator(name= «nomGen»,

    sequenceName= « tableSeq »)
  - @Id

    @GeneratedValue(strategy=GenerationType.SEQUENCE,

    generator= « nomGen »)
- En général, 1 séquence par clé

Encore quelques tags...

- @Transient
  - Attribut non persisté (intérêt ?)
- @Temporal(type)
  - type=TemporalType.DATE / TIME / TIMESTAMP
- @Lob + @Basic(fetch=FetchType.LAZY

Mapping multitable

- Un entité sur plusieurs tables ?
  - Schéma pré existant ou aspect pratique
  - Divergences entre modèle objet et relationnel (granularité...)
- @SecondaryTable
- @SecondaryTables

Exemple

```
@Entity
@Table(name= « EMPLOYE »)
@SecondaryTables({ // seulement si plusieurs
    @SecondaryTable(name= «ADRESSE»,
        pkJoinColumns={@PrimaryKeyJoinColumn(name= «ADRESSE_ID»)}),
    @SecondaryTable(name= «TELEPHONE»,
        pkJoinColumns={@PrimaryKeyJoinColumn(name= «PHONE_ID»)})
})
...
@Column(name= « CODE », table= « ADRESSE »)
public String getCode() {...
```

Objets encapsulés

- Démarche « inverse »
- Structuration hiérarchique de l'entity
- Mapping à plat dans une seule table
- @Embeddable / @Embedded
- cf clés encapsulées (très similaire...)

Message Driven Bean

- Session et entity suffisent pour gérer les objets à invocation classique
- Cas des communications asynchrones
  - e.g. : log, traitement batch, appli extérieure, etc.
- invocation classique inadaptée
- Utilisation de boîtes aux lettres

Message Driven Bean

- Dans le monde EJB :
  - Seuls les MDB reçoivent des messages
  - Ils ne servent qu'à ça, et donc
  - Ils peuvent implementer les modes P2P et pub/sub
  - Mais pas le mode request/reply!

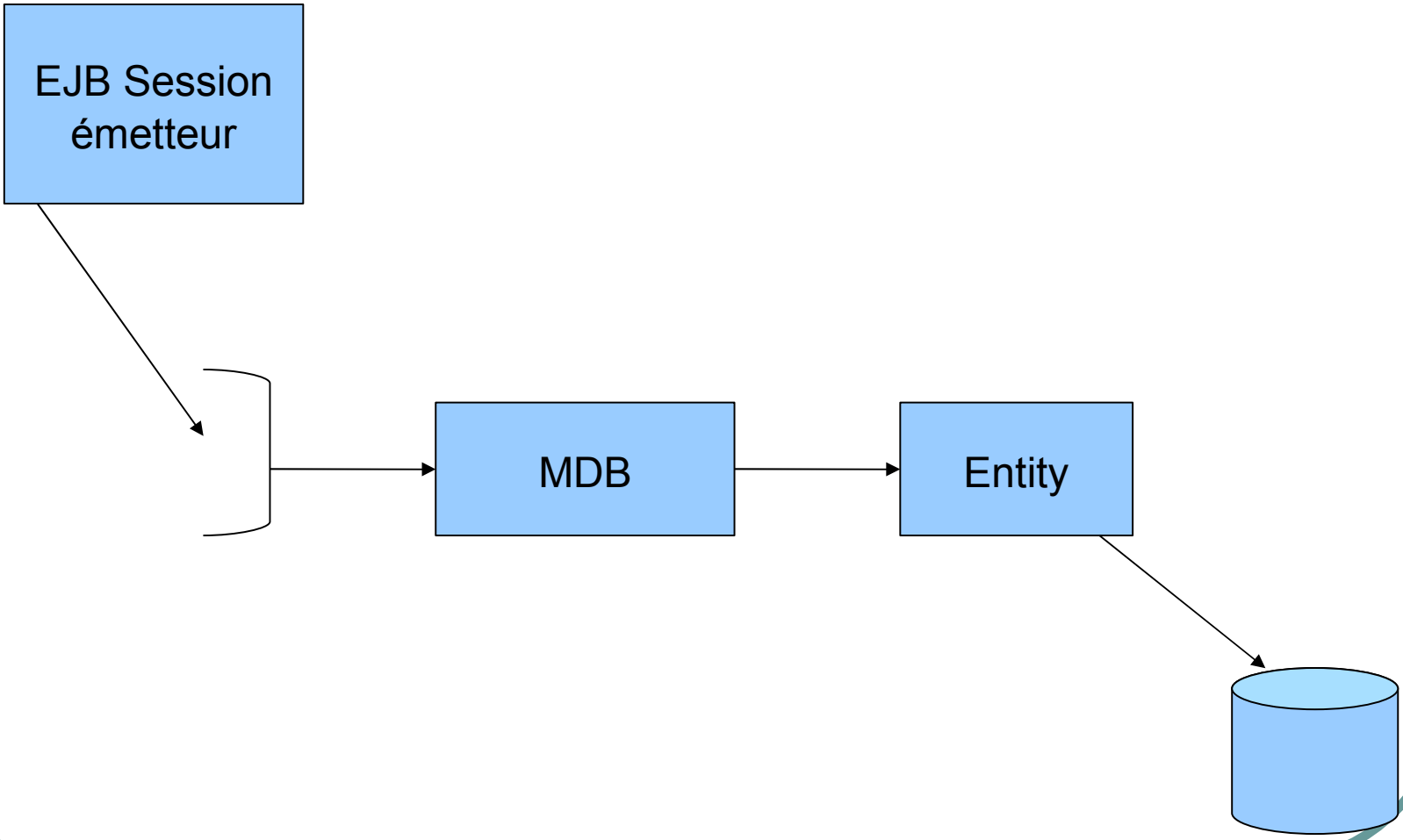
Que contient un MDB ?

- Des tags...
  - Quelle type de ressource écoute-t-il ?
- Une méthode « interface »
  - OnMessage
- D'autres méthodes ?
- Il n'a pas d'interface local ou remote...

Exemple : log d'administration

- Objectif: mémoriser toutes les tâches d'administration
- Méthode : envoi d'un message contenant le nom de l'utilisateur, la date, le type d'action.
- Le MDB se contente de stocker dans une table de log

Vue schématique



Code du MDB

```
@MessageDriven(activationConfig={
        @ActivationConfigProperty(
                propertyName= « destinationType »,
                propertyValue= « javax.jms.Queue »),
        @ActivationConfigProperty(
                propertyName= « destination »,
                propertyValue= « queue/A »)
})
public class LogBean implements javax.jms.MessageListener {
        @PersistenceContext(unitName= « maBase »)
        private EntityManager em;
```

Code du MDB (suite)

```
        public void OnMessage(Message m){
                MapMessage mm=(MapMessage) m;
                Log l=new Log()
                l.setDate(mm.getJMSTimestamp());
                l.setUserName(mm.getString(« user »);
                l.setOp(mm.getString(« op »);
                em.persist(l);
        }
}
```

Envoi de message...

- Pour l'envoi, on reste sur les bonnes vieilles méthodes...
- En utilisant tout de même les injections pour la factory et la file d'attente (ou liste diffusion)

Exemple

```
....
@Resource(mappedName= « ConnectionFactory »)
        private ConnectionFactory cf;
@Resource(mappedBy= « queue/A »)
        private Queue queue;
...
try{
        QueueConnection connect=cf.createQueueConnection();
        QueueSession s=connect.createQueueSession(true,0);
        QueueSender qs=s.createQueueSender(queue);
        ...
```

Plus d'infos sur les MDB

- ActivationConfigProperty :messageSelector
  - Filtrage des messages
- MDB et connecteurs
  - JMS n'est qu'un des protocoles utilisables
  - D'autres types de messageries peuvent être utilisées : mise en oeuvre de connecteurs
    - eg : mailer

Cycle de vie