

Java Persistence API

Objectifs

version 3.0

Objectifs

- Comprendre le rôle d'un ORM
- Comprendre le cycle de vie des objets principaux
 - EntityManagerFactory, EntityManager et Entity
- Savoir paramétrer JPA
- Savoir créer et utiliser des entités JPA
- Savoir créer des requêtes JPA
 - JPQL
 - Criteria



Chapitres

0. Objectifs
1. Accès aux données
2. Concepts de base de JPA
3. Entité JPA
4. Liaisons entre entités et non entités
5. Liaisons entre entités
6. Héritage
7. JPQL
8. Clés simples et composées
9. Criteria API

copyleft

Support de formation créé par

Franck SIMON

<http://www.franck-simon.com>



copyleft

Cette œuvre est mise à disposition sous licence
Attribution

Pas d'Utilisation Commerciale

Partage dans les Mêmes Conditions 3.0 France.

Pour voir une copie de cette licence, visitez
<http://creativecommons.org/licenses/by-nc-sa/3.0/fr/>

ou écrivez à

Creative Commons, 444 Castro Street, Suite 900,
Mountain View, California, 94041, USA.



JPA

Accès aux données

Introduction

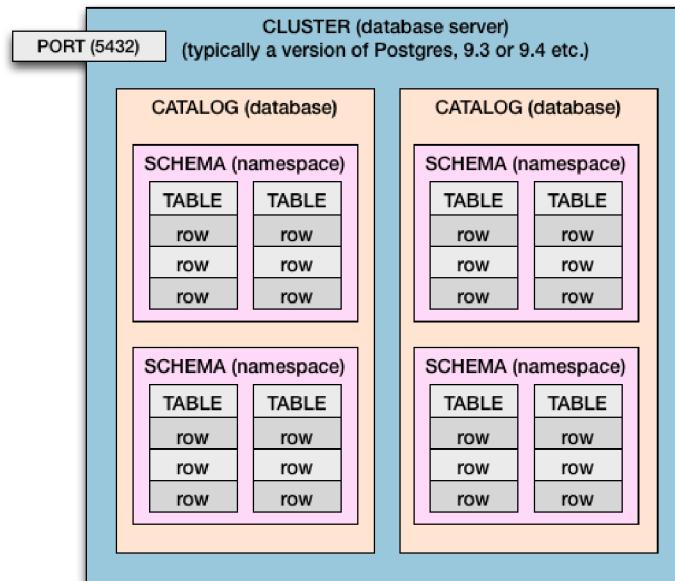
- La majeure partie des applications nécessite de sauvegarder les données qu'elles manipulent
- La base de données permet d'obtenir ce service
 - serveur de base de données
 - Oracle, SQL Server, MySQL, ...
 - base de données embarquée
 - Derby, H2DB, SQLite, ...
- Nos applications utilisent des bases de données

Introduction

- Catalogue : contient les méta-données des bases
 - implémentation différente selon les base
 - un catalogue pour toutes les bases de données (MySQL)
 - table information_schema
 - un catalogue par base (Oracle)
- Schema : description de l'organisation des données
 - tables, colonnes, relations entre tables, sécurité ...
- Dans de nombreuses bases de données le catalogue est synonyme de base de données

Introduction

- Exemple d'organisation



Introduction

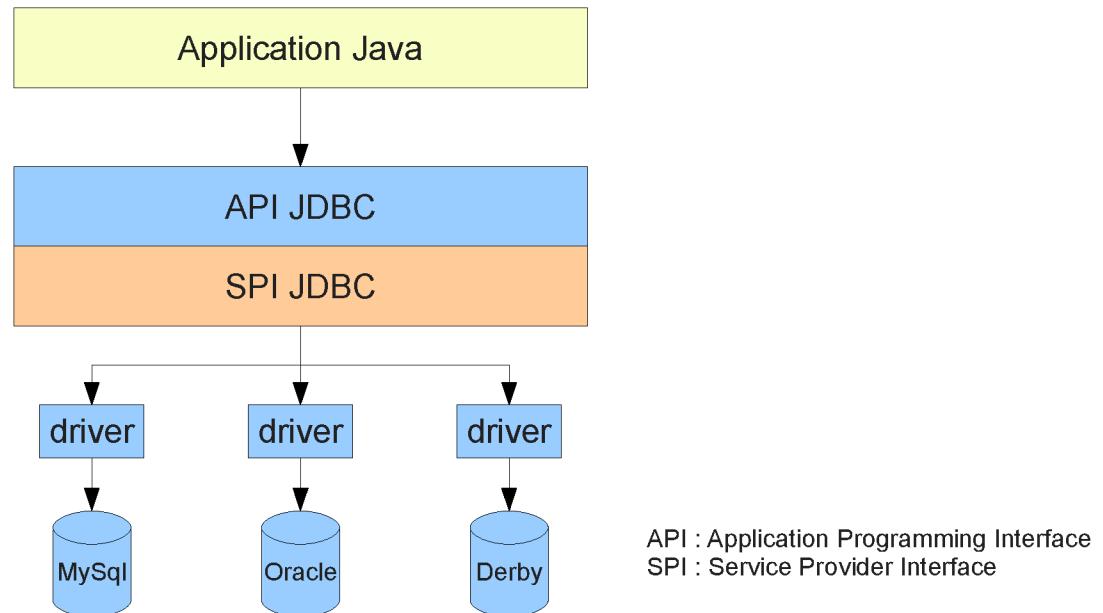
- L'interrogation d'une base de données est effectuée traditionnellement en SQL
 - Structured Query Language
 - principales commandes :
 - INSERT, UPDATE, DELETE, SELECT, ...
 - cf. <https://sql.sh/>

Introduction

- Tous les langages permettent de
 - se connecter à une base de données
 - par l'URL du serveur
 - par le système de fichier
 - envoyer des commandes SQL vers la base
 - traiter le retour de ces commandes
- En Java, JDBC permet d'utiliser une API générique
 - Java DataBase Connectivity
 - comme ODBC sous Windows

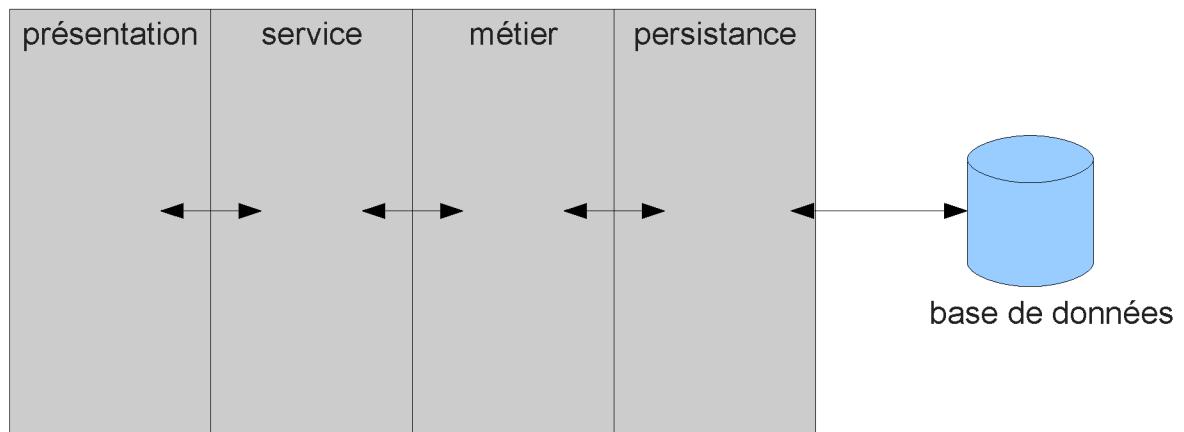
Introduction

- JDBC



Architecture

- Classiquement les applications sont découpées en N tiers



Architecture

- La couche de persistance est chargée
 - d'interagir avec la base de données
 - CRUD : Create Read Update Delete
 - de créer les objets métiers d'après les enregistrement qui sont dans les tables
 - cette couche est appelée
 - persistance
 - DAO - Data Access Object
 - repository

Architecture

- Exemple
 - une base de données avec les communes française
 - une classe Commune qui modélise une commune
 - c'est notre objet du domaine, une entité
 - une classe CommuneDAO qui est chargée d'interagir avec la base de données
 - recherche de communes par code postal, département, ...
 - dans la couche de persistance

JDBC

- JDBC fournit une couche d'abstraction vis à vis des bases de données
 - l'utilisation de la base est uniforme pour le développeur
- Cependant, JDBC possède un certain nombre de contrainte
 - écart entre le modèle objet (classes Java) et le modèle en base (tables)
 - utilisation du SQL pour interagir avec la base
 - ce qui implique de connaître la structure des tables
 - redondance de code

JDBC

- Table en base de données
 - base : user-france
 - tables : villes

| Nom de la colonne | # | Type de donnée | Non Null | Auto-Incrémantation | Clef | Défaut | Extra |
|-------------------|---|----------------|-------------------------------------|-------------------------------------|------|--------|----------------|
| 123 pk | 1 | int(11) | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | PRI | | auto_increment |
| ABC region | 2 | varchar(100) | <input checked="" type="checkbox"/> | <input type="checkbox"/> | | | |
| ABC departement | 3 | varchar(100) | <input checked="" type="checkbox"/> | <input type="checkbox"/> | | | |
| ABC nom | 4 | varchar(100) | <input checked="" type="checkbox"/> | <input type="checkbox"/> | | | |
| ABC code_postal | 5 | varchar(10) | <input checked="" type="checkbox"/> | <input type="checkbox"/> | | | |
| 123 lat | 6 | double | <input checked="" type="checkbox"/> | <input type="checkbox"/> | | | |
| 123 lng | 7 | double | <input checked="" type="checkbox"/> | <input type="checkbox"/> | | | |

JDBC

- Classe Commune - extrait

```
public class Commune {  
    private int id; // reflet de la clé primaire  
    private String nom;  
    private String codePostal;  
    private String departement;  
    private String region;  
    private double latitude;  
    private double longitude;  
  
    // getteurs et setteurs  
    // ...  
}
```

JDBC

- Connexion à la base
 - application Java SE

```
public class DAOConnexion {  
    // 4 paramètres de connexion  
    String driverName = "com.mysql.jdbc.Driver";  
    String url = "jdbc:mysql://localhost:3306/user-france?serverTimezone=UTC";  
    String user = "user";  
    String pswd = "userpw";  
  
    public DAOConnexion() throws ClassNotFoundException {  
        Class.forName(driverName);  
    }  
  
    public Connection getConnection() throws SQLException {  
        return DriverManager.getConnection(url, user, pswd);  
    }  
}
```

JDBC

- Connexion à la base de données
 - 4 éléments sont nécessaires
 - la classe driver de la base de données cible
 - l'URL de connexion à la base
 - les identifiant et mot de passe de connexion à la base

JDBC

- L'URL de connexion est constituée de trois parties
 - **protocole:sous-protocole:infos-complémentaires**
 - protocole : toujours jdbc
 - sous-protocole : dépend de la base, correspond au driver utilisé
 - infos-complémentaires : chaîne de connexion à la base, dépend de la base utilisée
 - **exemples**

`jdbc:odbc:cds_et_dvds`

`jdbc:derby://localhost:1527/D:/SERVEURS/db-derby-10.10.1.1-bin/databases/france`

`jdbc:mysql:///france?user=root&password=password`

`jdbc:mysql://localhost:3306/bovoyage`

JDBC

- Classe CommuneDAO - recherche par code postal

```
public List<Commune> getCommunesParCodePostal(String cp) throws SQLException{
    String sql = "SELECT * FROM villes WHERE code_postal LIKE ?";
    Connection cnx = daoConnexion.getConnection();
    PreparedStatement ps = cnx.prepareStatement(sql);

    // Execution de la requête
    ps.setString(1, cp+"%");
    ResultSet rs = ps.executeQuery();
    List<Commune> communes = new ArrayList<>();
    while(rs.next()) {
        Commune c = createCommune(rs);
        communes.add(c);
    }
    cnx.close();
    return communes;
}
```

méthode de mapping avec la table

JDBC

- Classe CommuneDAO - mapping avec la table
 - méthode appelée par les méthodes de recherche

```
private Commune createCommune(ResultSet rs) throws SQLException {
    Commune c = new Commune();
    c.setCodePostal(rs.getString("code_postal"));
    c.setDepartement(rs.getString("departement"));
    c.setId(rs.getInt("pk"));
    c.setLatitude(rs.getDouble("lat"));
    c.setLongitude(rs.getDouble("lng"));
    c.setNom(rs.getString("nom"));
    c.setRegion(rs.getString("region"));
    return c;
}
```

JDBC

- Inconvénients
 - beaucoup de copie/coller
 - les développeurs n'ont pas forcément une connaissance pointue des requêtes SQL
 - jointure, tables de liaisons, ...
 - utilisation de la classe Statement plutôt que PreparedStatement
 - optimisation
 - risque d'injection SQL

ORM

- Object Relational Mapping
- Fournit une passerelle entre le monde Objet et la base de donnée
 - prend en charge le mapping entre la classe et la/les tables
 - gère les jointures
 - langage de requête spécifique, orienté Objet
 - les noms de classe, propriétés sont utilisées
 - les jointures sont calculées
 - prend en charge le CRUD de base

JPA

- JPA - Java Persistence API
 - spécification ORM (Object Relational Mapping) Java
 - JPA - JSR 220
 - JPA 2.0 - JSR 317
 - JPA 2.1 - JSR 338
 - JAP 2.2 - maintenance de JSR 338
 - package : javax.persistence
 - implémentations
 - Hibernate
 - EclipseLink
 - DataNucleus
 - OpenJPA
 - ...

JPA

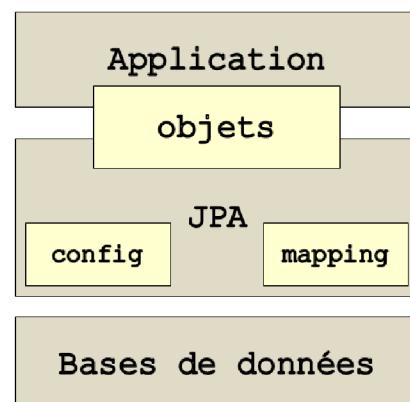
- Peut-être utilisée dans une application
 - Java SE
 - aucune implémentation fournit
 - Java EE
 - le serveur Java EE fournit une implémentation
 - généralement Hibernate
 - Tomcat étant un conteneur Servlet/JSP, aucune implémentation n'est fourni
 - idem pour undertow, jetty, ...

JPA

- Nécessite
 - la description de la connexion à la base
 - soit directement, si Java EE
 - soit sur le DataSource du serveur
 - c'est le serveur qui gère un pool de connexions à la base
 - même les conteneurs de Servlet/JSP
 - l'implémentation JPA
 - divers paramètres
 - visualisation des requêtes, création des tables, ...

JPA

- JPA implémente la couche d'accès aux données
 - retourne des objets du domaine comme réponse aux requêtes de type recherche
 - persiste les objets du domaine



- La mise en œuvre de JPA nécessite :
 - de décrire le mapping entre les classes entités et la(les) table(s) en base
 - par annotation, ou fichier XML
 - de décrire l'accès à la base
 - dans un fichier persistence.xml
 - d'utiliser les classes de base de l'API
 - EntityManagerFactory, EntityManager



JPA

Concepts de base

Architecture d'un projet JPA

- Il faut
 - créer le fichier *persistence.xml*
 - dans un répertoire META-INF
 - situé dans le répertoire des sources Java
 - attention à ne pas confondre ce répertoire avec le META-INF des applications web
 - créer les classes entités
 - mapping par annotations (ou fichier XML)
 - créer une couche DAO qui utilisera l'API JPA

Contenu du fichier *persistence.xml*

- Le nom de l'unité de persistance, qui sera utilisé par l'EntityManagerFactory
- L'emplacement des entités
 - par défaut recherche automatique
 - mais on peut préciser
 - un fichier de mapping *orm.xml*
 - une liste de classe
 - une librairie .jar
- La classe créant un EntityManagerFactory
 - optionnel si dans un serveur Java EE

Contenu du fichier *persistence.xml*

- La connexion à la base de donnée
 - directement si Java SE
 - via une DataSource si utilisé dans un conteneur
 - il faudra préciser si le conteneur gère les transactions ou non
- Des propriétés propres à l'implémentation
 - des propriétés communes à toutes les implémentations ont été définies depuis JPA 2.0
 - connexion à la base de données
 - JPA 2.1 a défini un nouvel ensemble de propriétés pour la génération des tables en base

Contenu du fichier *persistence.xml*

- La racine du fichier est l'élément <persistence>
- ATTENTION
 - les espaces de nommage ont changés entre les version 2.0 et 2.1
 - passage de *java.sun.com* à *xmlns.jcp.org*

Contenu du fichier *persistence.xml*

- Élément racine jusqu'à JPA 2.0

```
<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_2_0.xsd"
    version="2.0">
...
</persistence>
```

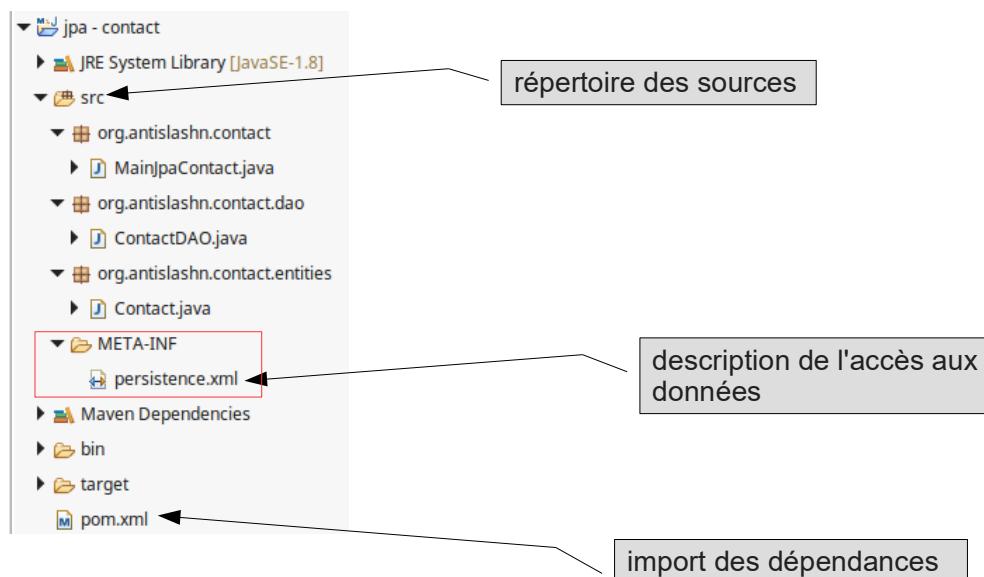
Contenu du fichier *persistence.xml*

- Élément racine à partir de JPA 2.1

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd"
    version="2.2">
...
</persistence>
```

Architecture d'un projet JPA

- Projet Java SE sous Eclipse



Architecture JPA

- Projet Java SE sous Eclipse
 - il faut fournir les dépendances adéquates
 - aucune gestion du cycle de vie de l'EntityManagerFactory et l'Entytimanager
 - le développeur doit donc les gérer
 - aucune gestion de transaction

Architecture JPA

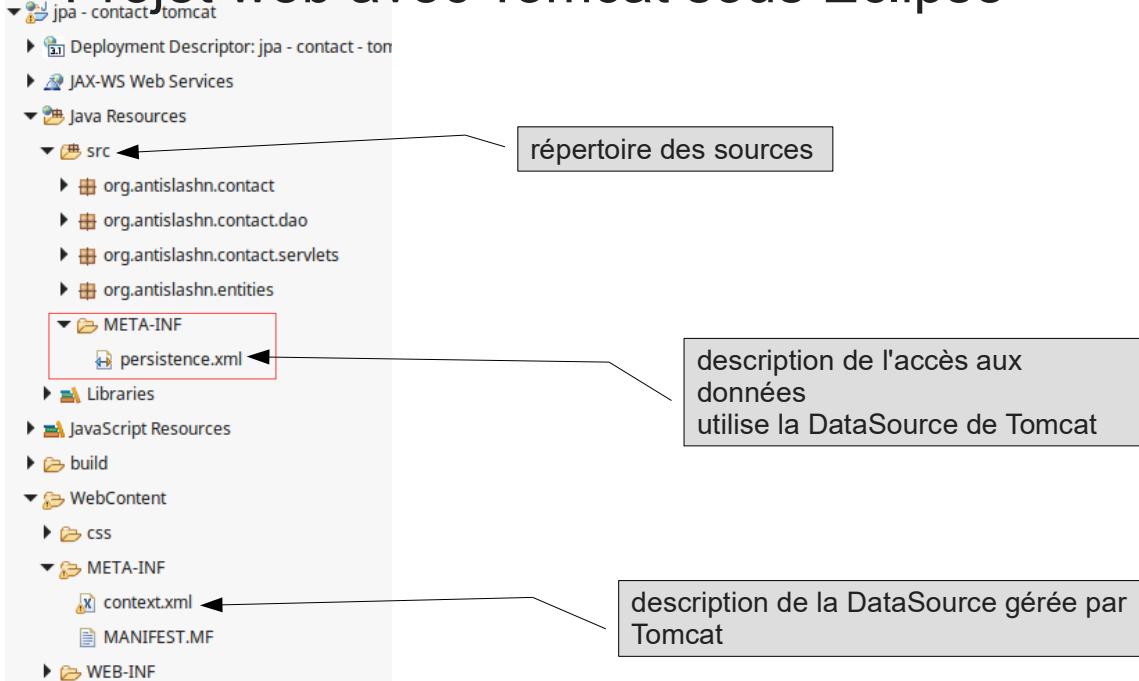
- Projet Java SE sous Eclipse
 - fichier *persistence.xml*

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
                      http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd"
  version="2.2">

  <persistence-unit name="contacts" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <properties>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
      <property name="javax.persistence.jdbc.url"
                value="jdbc:mysql://localhost:3306/user-contacts" />
      <property name="javax.persistence.jdbc.user" value="user" />
      <property name="javax.persistence.jdbc.password" value="userpw" />
      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5Dialect" />
      <property name="hibernate.hbm2ddl.auto" value="none" />
    </properties>
  </persistence-unit>
</persistence>
```

Architecture d'un projet JPA

- Projet web avec Tomcat sous Eclipse



Architecture JPA

- Projet web avec Tomcat sous Eclipse
 - Tomcat ne fournit pas les implémentations JPA
 - il faut donc lui fournir les dépendances adéquates
 - options possibles
 - copier les librairies dans le répertoire lib de Tomcat
 - copier les librairies dans le répertoire WEB-INF/lib de l'application web
 - ou utiliser Maven pour le faire
 - aucune gestion du cycle de vie de l'EntityManagerFactory et l'EntityManager
 - le développeur doit donc les gérer
 - aucune gestion de transaction

Architecture JPA

- Projet web avec Tomcat sous Eclipse

- fichier *persistence.xml*

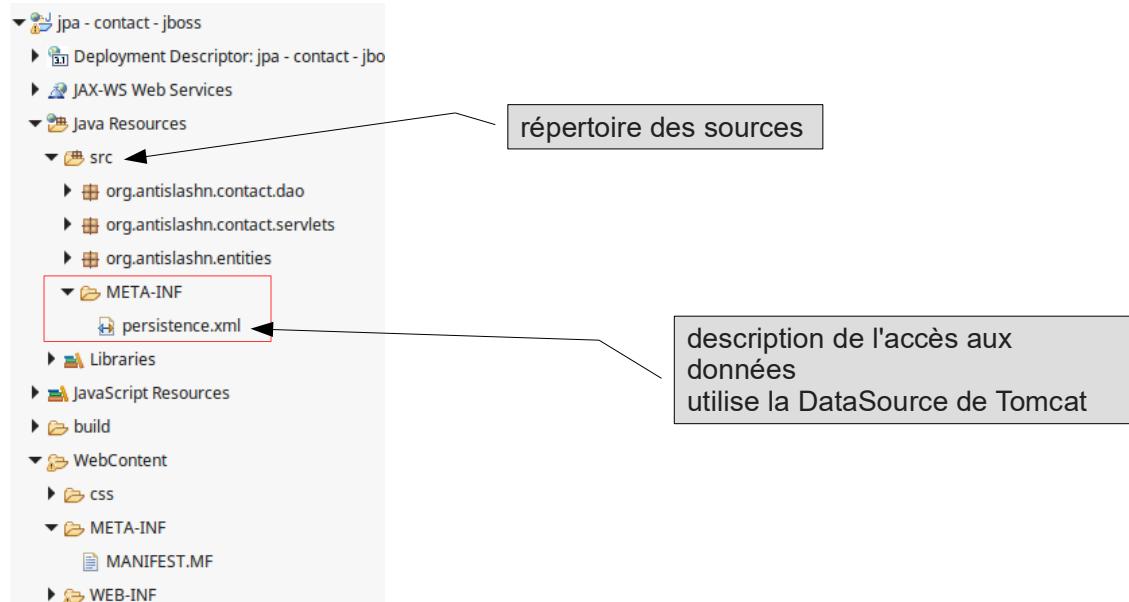
- c'est Tomcat qui gère la source de données

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

    <persistence-unit name="contacts" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <non-jta-data-source>java:comp/env/jdbc/contacts</non-jta-data-source>
        <properties>
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL5Dialect"/>
            <property name="hibernate.show_sql" value="true" />
        </properties>
    </persistence-unit>
</persistence>
```

Architecture JPA

- Projet Java EE sous Eclipse



Architecture JPA

- Le serveur Java EE
 - fournit une implémentation JPA
 - aucune librairie à ajouter
 - gère les transaction
 - gère le cycle de vie de l'EntityManagerFactory et l'EntityManager

Architecture JPA

- Projet Java EE sous Eclipse
 - fichier *persistence.xml*

```
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

    <persistence-unit name="contacts" transaction-type="JTA">
        <jta-data-source>java:/jboss/jdbc/contacts</jta-data-source>
        <properties>
            <property name="hibernate.show_sql" value="true" />
        </properties>
    </persistence-unit>
</persistence>
```

Architecture JPA

- Gestion des transactions

- par le développeur sous Java SE ou dans un conteneur Servlet/JSP

```
<persistence-unit name="contacts" transaction-type="RESOURCE_LOCAL">
```

- dans le conteneur référencement de la source de donnée par l'élément

```
<non-jta-data-source>java:comp/env/jdbc/contacts</non-jta-data-source>
```

- par le serveur Java EE

```
<persistence-unit name="contacts" transaction-type="JTA">
```

- référencement de la source de données

```
<jta-data-source>java:jboss/jdbc/contacts</jta-data-source>
```

Quelques propriétés de configuration

- Propriétés standards

- javax.persistence.jdbc.url
- javax.persistence.jdbc.driver
- javax.persistence.jdbc.user
- javax.persistence.jdbc.password
- javax.persistence.jdbc.show_sql
- javax.persistence.schema-generation.database.action
 - none, create, drop-and-create, drop

Quelques propriétés de configuration

- Propriétés Hibernate
 - hibernate.connection.url
 - hibernate.connection.driver_class
 - hibernate.connection.username
 - hibernate.connection.password
 - hibernate.hbm2ddl.auto
 - validate, update, create, create-drop
 - hibernate.dialect
 - classe de type org.hibernate.dialect.Dialect
- cf. <https://docs.jboss.org/hibernate/orm/5.0/javadocs/org/hibernate/cfg/Environment.html>

JPA Entity

Entité JPA

- Écrire la classe Java qui suit la spécification JavaBean
 - fournir un constructeur sans argument
 - chaque propriété est associée à des méthodes get/set
- Fournir une propriété représentant la clé primaire
 - c'est avec cette propriété que l'ORM maintient le lien entre l'instance et l'enregistrement en base



Entité JPA

- Annoter la classe avec `@Entity`
- Annoter les propriétés
 - l'identifiant avec `@Id`
 - l'identifiant est le reflet de la clé primaire de la table
 - par défaut toutes les propriétés sont persistées
 - `@Transient` si une propriété ne doit pas être persistée
 - certaines annotations peuvent être mises sur les propriétés ou les méthodes get/set
 - héritage JPA 1.0 : détermine le style d'injection
 - attention de ne pas mélanger les deux styles, il peut y avoir des différences de comportement en fonction des implémentations
 - préférer l'utilisation de l'annotation `@Access`

Entité JPA

- Au minimum deux annotations sont nécessaires
 - `@Entity` pour marquer la classe
 - `@Id` pour l'identifiant
- Par défaut le nom de la classe correspond au nom de la table
 - sinon utiliser l'annotation `@Table`
- Par défaut les noms des colonnes correspondent aux nom des propriétés
 - sinon utiliser l'annotation `@Column`

Entité simple

- L'entité la plus simple ne comporte
 - que des propriétés Java de type simple
 - pas d'objets embarqués (embedded)
 - pas de collections
 - pas de liens vers d'autres entités
 - pas de propriété ou collection de type entité
 - par exemple l'entité Contact liée à l'entité Adresse

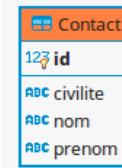
Entité simple

- Types associés automatiquement à des colonnes
 - types de base : int, long, float, double, ...
 - et leur wrapper : Integer, Long, Float, Double, ...
 - le type String
 - types dates du package java.sql
 - Date, Time, Timestamp
 - les types énumérés
 - les types Serializable
 - sont enregistrés sous forme de blob

Entité simple

- Même nommage dans la classe et la table

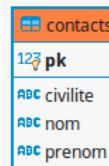
```
@Entity  
public class Contact {  
  
    @Id  
    private int id;  
    private String civilité;  
    private String nom;  
    private String prenom;  
  
    // constructeurs  
  
    // getteurs/setteurs  
}
```



Entité simple

- Noms différents
 - entre la classe et la table
 - entre la propriété *id* et le champ *pk*

```
@Entity  
@Table(name="contacts")  
public class Contact {  
  
    @Id  
    @Column(name="pk")  
    private int id;  
    private String civilité;  
    private String nom;  
    private String prenom;  
  
    // constructeurs  
  
    // getteurs/setteurs  
}
```



Injection

- L'annotation @Access précise le type d'injection
 - par les propriétés :AccessType.FIELD
 - sur les setteurs :AccessType.PROPERTY
 - peut paraître redondant avec le choix uniforme de poser les annotations sur les propriétés ou les setteurs/getteurs
 - une exception est levée s'il y a une incohérence

```
@Entity  
@Table(name="contacts")  
@Access(AccessType.FIELD)  
public class Contact {  
    ...  
}
```

```
@Entity  
@Table(name="contacts")  
@Access(AccessType.PROPERTY)  
public class Contact {  
    @Id @Column(name="pk") private int id;  
    ...  
}
```

Injection

- L'annotation @Access précise le type d'injection
 - par les propriétés :AccessType.FIELD
 - sur les setteurs :AccessType.PROPERTY
 - peut paraître redondant avec le choix uniforme de poser les annotations sur les propriétés ou les setteurs/getteurs
 - une exception est levée s'il y a une incohérence

```
@Entity  
@Table(name="contacts")  
@Access(AccessType.FIELD)  
public class Contact {  
    ...  
}
```

```
@Entity  
@Table(name="contacts")  
@Access(AccessType.PROPERTY)  
public class Contact {  
    @Id @Column(name="pk") private int id;  
    ...  
}
```

Injection

- Correct

```
@Entity  
@Table(name="contacts")  
@Access(AccessType.FIELD)  
public class Contact {  
    @Id @Column(name="pk") private int id;  
    ...  
}
```

- Erreur

```
@Entity  
@Table(name="contacts")  
@Access(AccessType.PROPERTY)  
public class Contact {  
    @Id @Column(name="pk") private int id;  
    ...  
}
```

[org.hibernate.AnnotationException](#): No identifier specified for entity: org.antislashn.contact.entities.Contact

Annotation @Table

- Permet de

- nommer la table : attribut name
 - de préciser si nécessaires le catalogue et le schéma
 - attributs catalog et schema
- préciser les index
 - attribut indexes
 - seulement si génération de la table par JPA
- préciser les contraintes d'unicité
 - attribut uniqueConstraints
 - seulement si génération de la table par JPA

Annotation @Table

- Attribut indexes

```
@Entity  
@Table(name="contacts",  
      indexes= {@Index(columnList="nom")})  
)  
public class Contact {  
    ...  
}
```

| Nom de la colonne | # | Type de donnée | Non Null | Auto-Incrémation | Clef | Défaut |
|-------------------|---|----------------|-------------------------------------|--------------------------|------|--------|
| pk | 1 | int(11) | <input checked="" type="checkbox"/> | <input type="checkbox"/> | PRI | |
| civilite | 2 | varchar(255) | <input type="checkbox"/> | <input type="checkbox"/> | | |
| nom | 3 | varchar(255) | <input type="checkbox"/> | <input type="checkbox"/> | | MUL |
| prenom | 4 | varchar(255) | <input type="checkbox"/> | <input type="checkbox"/> | | |
| surnom | 5 | varchar(255) | <input type="checkbox"/> | <input type="checkbox"/> | | |

Annotation @Table

- Attribut uniqueConstraint

```
@Entity  
@Table(name="contacts",  
      uniqueConstraints= {@UniqueConstraint(name="nom_prenom",columnNames= {"nom","prenom"})})  
)  
public class Contact {  
    ...  
}
```

| Colonnes | Nom | Propriétaire | Type |
|------------------|------------|--------------|-------------|
| Contraintes | PRIMARY | contacts | PRIMARY KEY |
| Clefs étrangères | nom_prenom | contacts | UNIQUE KEY |

Annotation @Id

- Annotation @GeneratedValue
 - stratégie de gestion de l'identifiant
 - attribut strategy
 - AUTO : stratégie choisie par le provider JPA, par rapport à la base de données
 - ATTENTION aux implémentations et versions
 - IDENTITY : stratégie de la base de données
 - pour MySQL : auto_increment
 - SEQUENCE : génération par une séquence définie par la base de données
 - utilise @SequenceGenerator
 - TABLE : utilisation d'une table dédiée
 - utilise @TableGenerator

Annotation @Column

- Attributs
 - name : nom de la colonne
 - length : taille pour les chaînes de caractères
 - unique : ajout d'une contrainte d'unicité
 - nullable : valeur NULL ou non
 - precision, scale : paramètres pour les nombres
 - columnDefinition : permet de donner en SQL le code de création d'une colonne
 - code natif
 - table : nom de la table secondaire contenant la colonne

Annotation @Column

- Exemple - extrait

```
@Id  
@Column(name="pk")  
private int id;  
@Column(length=5, nullable=false)  
private String civilité;  
@Column(length=100, nullable=false)  
private String nom;  
@Column(length=100, nullable=false)  
private String prenom;  
@Column(length=100, nullable=true)  
private String surnom;  
@Column(length=100, nullable=false, unique=true)  
private String email;
```

| Nom de la colonne | # | Type de donnée | Non Null | Auto-Incrémantation | Clef | Dé |
|-------------------|---|----------------|-------------------------------------|--------------------------|------|----|
| pk | 1 | int(11) | <input checked="" type="checkbox"/> | <input type="checkbox"/> | PRI | |
| civilité | 2 | varchar(5) | <input checked="" type="checkbox"/> | <input type="checkbox"/> | | |
| email | 3 | varchar(100) | <input checked="" type="checkbox"/> | <input type="checkbox"/> | UNI | |
| nom | 4 | varchar(100) | <input checked="" type="checkbox"/> | <input type="checkbox"/> | | |
| prenom | 5 | varchar(100) | <input checked="" type="checkbox"/> | <input type="checkbox"/> | | |
| surnom | 6 | varchar(100) | <input type="checkbox"/> | <input type="checkbox"/> | | |

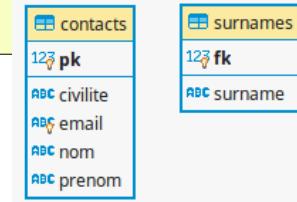
Utilisation d'une table secondaire

- La classe Contact possède une propriété surnom
 - il y a une table contacts et une table surnames
- @SecondaryTables et @SecondaryTable permettent de mettre une entité en relation avec plusieurs tables
 - déclaration de la(les) table(s) secondaire(s) au niveau de l'entité
 - au niveau de la propriété @Column fait référence à la table secondaire

Utilisation d'une table secondaire

- Exemple

```
@Entity
@Table(name="contacts")
@AccessType(AccessType.FIELD)
@SecondaryTable(name="surnames",pkJoinColumns=@PrimaryKeyJoinColumn(name="fk"))
public class Contact {
    @Id @Column(name="pk")
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    ...
    @Column(table="surnames",name="surname",length=100, nullable=true)
    private String surnom;
    ...
}
```



Énumération Java

- Une propriété peut-être liée à une énumération Java
 - exemple : la propriété `civilite`
- L'annotation `@Enumerated` permet de préciser si le stockage de l'information est
 - sous forme d'un entier, dans l'ordre de l'enum java
 - `EnumType.ORDINAL`
 - sous forme d'une chaîne de caractères
 - `EnumType.STRING`

```
public enum Civilite {
    M, Mme, Dr, Dre, Me, Pr
}
```

```
@Column(length=5, nullable=false)
@Enumerated(EnumType.STRING)
private Civilite civilite;
```

Autres annotations sur les propriétés

- `@Temporal` - type des propriétés de type `java.util.Date` et `java.util.Time`
 - `TemporalType.DATE`
 - `TemporalType.TIME`
 - `TemporalType.TIMESTAMP`
- `@Lob`
 - force la sauvegarde dans un blob SQL

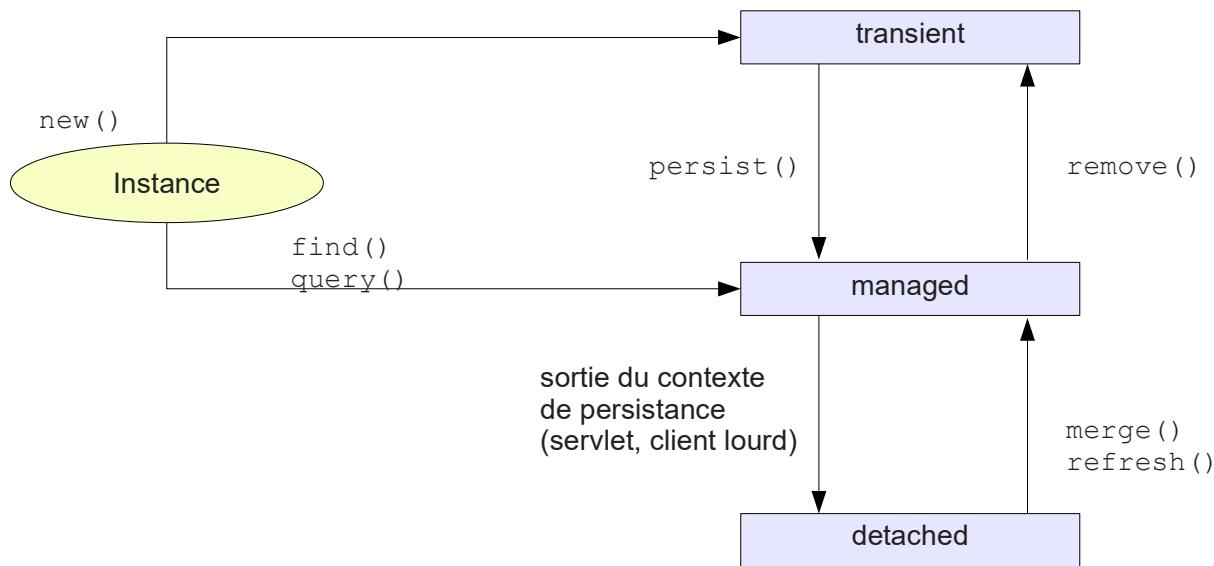
Utilisation de l'entité

- Classes d'accès aux données
 - comparaison des technologies

| Concept | JDBC | Hibernate | JPA |
|-----------------------|--------------|--------------------|----------------------|
| Ressource | Connection | SessionEntity | EntityManager |
| Fabrique de ressource | DataSource | SessionFactory | EntityManagerFactory |
| Exception | SQLException | HibernateException | PersistenceException |

- il en ressort :
 - l'`EntityManagerFactory` à une durée de vie de l'application
 - l'`EntityManager` à une durée de vie brève
 - en général, la méthode

Cycle de vie des entités



EntityManager

- L'utilisation de JPA nécessite une interaction avec un EntityManager
 - fournit les opérations de création, recherche, mise à jour et de suppression
 - une instance d'EntityManager est fournie
 - par le serveur dans un environnement Java EE
 - par injection avec l'annotation `@PersistenceContext`
 - par demande auprès de l'EntityManagerFactory dans un environnement Java SE ou dans un conteneur Servlet/JSP

EntityManager

- Dans un environnement Java EE
 - EntityManager est injecté via @PersistenceContext
 - le serveur Java EE gère
 - son cycle de vie
 - les transactions
- Dans un environnement non Java EE
 - Java SE, ou conteneur Servlet/JSP
 - le cycle de vie l'EntityManager doit être géré
 - les transactions doivent être gérées

Récupération de l'EntityManager

- Hors conteneur Java EE
 - récupération d'un EntityManager via un EntityManagerFactory
 - on précise le nom de l'unité de persistance définie dans le fichier *persistence.xml*
- L'EntityManagerFactory doit être fermé à la fin de l'application
 - il est exécuté dans un thread indépendant
 - utiliser un ApplicationListener pour les conteneurs Servlet/JSP

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("contacts");
EntityManager em = emf.createEntityManager();
```

Récupération de l'EntityManager

- Exemple avec Java SE

```
public static void main(String[] args) {  
    Contact c1 = new Contact("M", "LAGAFFE", "Gaston");  
    EntityManagerFactory emf = Persistence.createEntityManagerFactory("foo");  
    ContactDAO dao = new ContactDAO(emf);  
    dao.save(c1);  
    emf.close();  
}
```

```
public class ContactDAO {  
    private EntityManagerFactory emf;  
  
    public ContactDAO(EntityManagerFactory emf){  
        this.emf = emf;  
    }  
  
    public void save(Contact contact){  
        EntityManager em= emf.createEntityManager();  
        ...  
        em.close();  
    }  
    ...
```

Récupération de l'EntityManager

- Exemple sous Tomcat

- utilisation d'un ServletContextListener

```
@WebListener  
public class ApplicationListener implements ServletContextListener {  
    public static final String EMF = "emf";  
  
    @Override  
    public void contextInitialized(ServletContextEvent event) {  
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("foo");  
        event.getServletContext().setAttribute(EMF, emf);  
    }  
  
    @Override  
    public void contextDestroyed(ServletContextEvent event) {  
        EntityManagerFactory emf = (EntityManagerFactory) event.getServletContext().getAttribute(EMF);  
        emf.close();  
    }  
}
```

Récupération de l'EntityManager

- Exemple sous Tomcat
 - récupération du factory

```
public class AddContactServlet extends HttpServlet{  
    private static final long serialVersionUID = 1L;  
    private ContactDAO dao;  
  
    @Override  
    public void init() throws ServletException {  
        EntityManagerFactory emf = null;  
        emf = (EntityManagerFactory)this.getServletContext().getAttribute(ApplicationListener.EMF);  
        dao = new ContactDAO(emf);  
    }  
    ...  
}
```

Récupération de l'EntityManager

- Dans une serveur Java EE
 - la récupération d'un EntityManager est effectuée par injection dans un EJB

```
@PersistenceContext(unitName="jpa") private EntityManager em;
```

- il est automatiquement fermé à la sortie de la méthode

Développer avec JPA

- Toutes les opérations effectuées sur l'EntityManager doivent être effectuées dans une transaction

```
...
public void save(Contact contact)
{
    EntityManager em= emf.createEntityManager();
    EntityTransaction transaction = em.getTransaction();
    transaction.begin();
    em.persist(contact);
    transaction.commit();
    em.close();
}
...
```

EntityManager

- Opérations de base :
 - sauvegarde des entités avec persist (...)
 - récupération d'un objet par son identifiant : find (...)
 - suppression d'une entité en base : remove (...)
 - mise à jour de la base avec l'objet : merge ()
 - mise à jour de l'objet avec la base : refresh ()
 - les modifications sur les objets sont propagées vers la base lors du commit sur la transaction

Exemple avec Java SE

- L'entité est une classe Contact
 - cette même classe sera utilisée dans les autres exemples

```
@Entity
@Table(name="personnes")
@NamedQueries({
    @NamedQuery(name="allContacts",query="SELECT c FROM Contact c")
})
@Access(AccessType.FIELD)
public class Contact {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="pk")
    private int id;
    private String civilité;
    private String nom;
    private String prenom;
    ...
}
```

Exemple Java SE

- Classe DAO

```
public class ContactDAO {
    private EntityManagerFactory emf;

    public ContactDAO(EntityManagerFactory emf) {
        this.emf = emf;
    }

    // gestion du cycle de vie de l'EntityManager
    public Contact findById(int id) {
        Contact c = null;
        EntityManager em = emf.createEntityManager();
        c = em.find(Contact.class, id);
        em.close();
        return c;
    }

    // gestion des transactions
    // gestion du cycle de vie de l'EntityManager
    public void save(Contact contact) {
        EntityManager em = emf.createEntityManager();
        em.getTransaction().begin();
        em.persist(contact);
        em.getTransaction().commit();
        em.close();
    }
}
```

Exemple Java SE

- Application
 - gestion du cycle de vie de l'EntityManagerFactory

```
public class MainJpaContact {  
  
    public static void main(String[] args) {  
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("contacts");  
        ContactDAO dao = new ContactDAO(emf);  
        Contact c = dao.findById(10);  
        System.out.println(c);  
  
        // Fermeture de l'EntityManagerFactory  
        emf.close();  
    }  
  
}
```

Exemple Tomcat

- Les classes entité Contact et dao ContactDAO sont les mêmes que pour l'environnement Java SE
 - gestion du cycle de vie de l'EntityManagerFactory par un ServletContextListener

```
@WebListener  
public class ApplicationListener implements ServletContextListener {  
  
    public void contextInitialized(ServletContextEvent evt) {  
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("contacts");  
        evt.getServletContext().setAttribute(Constants.CONTRACT_DAO,new ContactDAO(emf));  
        evt.getServletContext().setAttribute(Constants.EMF,emf);  
    }  
  
    public void contextDestroyed(ServletContextEvent evt) {  
        // Fermeture de l'EntityManagerFactory  
        EntityManagerFactory emf =  
            (EntityManagerFactory) evt.getServletContext().getAttribute(Constants.EMF);  
        emf.close();  
    }  
}
```

Exemple avec Java EE

- La classe entité Contact reste la même
- Le DAO est simplifié

```
@Singleton
public class ContactDAO {
    @PersistenceContext(unitName="contacts") private EntityManager em;

    public Contact getContactById(int id) {
        return em.find(Contact.class, id);
    }

    public void remove(Contact contact) {
        Contact c1 = em.find(Contact.class, contact.getId());
        em.remove(c1);
    }

    public void save(Contact contact) {
        em.persist(contact);
    }
}
```

Un peu de JPQL

- JPQL : Java Persistence Query Language
- Langage similaire à SQL
 - mais JPQL permet une approche objet des requêtes
 - SQL est basé sur la connaissance des structures de la base
 - JPQL est basé sur la connaissance du modèle objet
- La recherche des adresses est effectuée par
 - SELECT * FROM table_adresses en SQL
 - SELECT a FROM Adresse a en JPQL

Un peu de JPQL

- Les requêtes sont créées via l'EntityManager
 - `createQuery()` pour les requêtes JPQL
 - `createNamedQuery()` pour les requêtes nommées définies par les annotations
 - `@NamedQueries({...})`
 - `@NamedQuery()`
 - `createNativeQuery()` pour les requêtes SQL
- Les requêtes peuvent comporter des paramètres

Un peu de JPQL

- Exemple de requête avec paramètres

```
public List<Video> getVideoParTitre(String titre)
{
    Query query = em.createQuery("from Produit p where p.titre like :titre");
    query.setParameter("titre", titre+"%");
    return query.getResultList();
}
```

- Les paramètres peuvent aussi être indicés

```
public List<Video> getVideoParTitre(String titre)
{
    Query query = em.createQuery("from Produit p where p.titre like ?1");
    query.setParameter(1, titre+"%");
    return query.getResultList();
}
```

Un peu de JPQL

- Utilisation de requêtes nommées
 - mise en place par annotation

```
@Entity  
@Table(name="cd_audios")  
@PrimaryKeyJoinColumn(name="ke_produit")  
@NamedQueries({  
    @NamedQuery( name="CDAudio.getParArtiste",  
                query="from CDAudio cd where cd.artiste = :artiste" )  
})  
public class CDAudio extends Produit  
{...}
```

- utilisation dans le DAO

```
public List<CDAudio> getCDAudiosParArtiste(String artiste)  
{  
    Query query = em.createNamedQuery("CDAudio.getParArtiste");  
    query.setParameter("artiste", artiste);  
    return query.getResultList();  
}
```

JPA

Liaisons entre entité et non entité

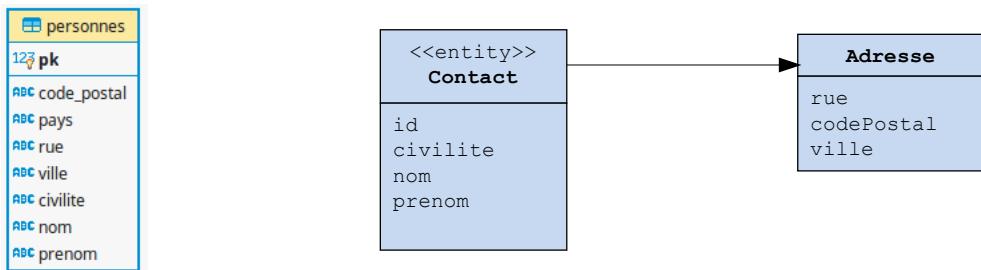
Entité et non entité

- Une entité possède un identifiant
 - reflet de la clé primaire de la table
- Notre modèle POO peut-être très différent du modèle logique de données
 - exemple :
 - une classe Contact et une classe Adresse
 - mais une seule table pour toutes ces données



Entité et non entité

- MLD versus modèle objet

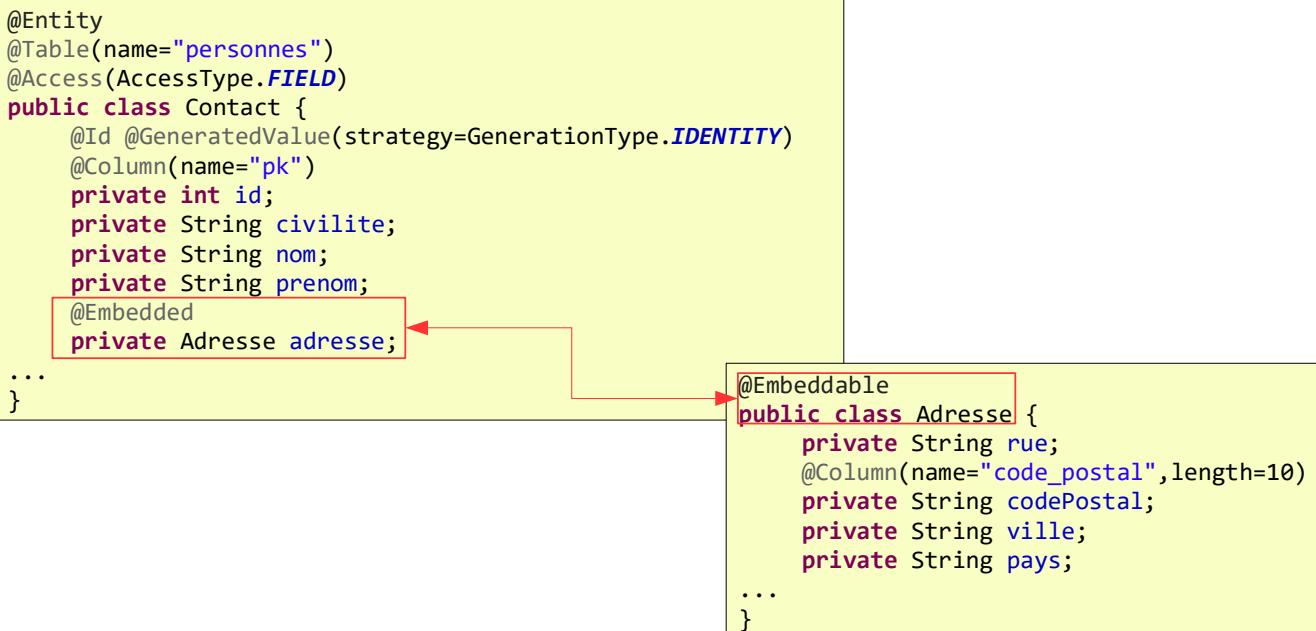


Objets embarqués

- JPA fait la différence entre les entités et les objets internes
- Les entités ont leur propres identifiant
 - la classe est annotée avec `@Entity`
 - il possède un `@Id`
- Un objet interne à une entité peut ne pas être lui-même une entité
 - c'est un objet embarqué
 - la classe est annotée par `@Embeddable`

Objets embarqués

- Classes



JPQL et objets embarqués

- Les requêtes JPQL sont effectués sur les objets embarqués avec la logique POO

```
@Entity
@Table(name="personnes")
@NamedQueries({
    @NamedQuery(name="Contact.getAllContacts",
                query="SELECT c FROM Contact c"),
    @NamedQuery(name="Contact.getContactsbyPays",
                query="SELECT c FROM Contact c WHERE c.adresse.pays=:pays")
})
@AccessType(AccessType.FIELD)
public class Contact{
    ...
}
```

JPQL et objets embarqués

- ContactDAO : exploitation de la requête

```
public class ContactDAO {  
    private EntityManagerFactory emf;  
  
    public ContactDAO(EntityManagerFactory emf) {  
        this.emf = emf;  
    }  
  
    public List<Contact> getContactsByPays(String pays){  
        EntityManager em = emf.createEntityManager();  
        List<Contact> contacts = em.createNamedQuery("Contact.getContactsbyPays", Contact.class)  
            .setParameter("pays", pays)  
            .getResultList();  
        em.close();  
        return contacts;  
    }  
    ...  
}
```

Collection de types

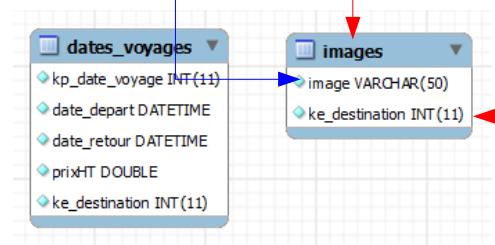
- De nombreuses tables comportent des listes de types Java
 - String, int, double, ...



- Les types Java ne peuvent pas être `@Embeddable`
- Il faut marquer la collection comme `@ElementCollection`

Collection de types

```
@Entity  
@Table(name="destinations")  
public class Destination {  
    @Id  
    @GeneratedValue(strategy=GenerationType.AUTO)  
    @Column(name="kp_destination")  
    private long id;  
    private String region;  
  
    @ElementCollection  
    @CollectionTable(name="images",joinColumns=@JoinColumn(name="ke_destination"))  
    @Column(name="image")  
    private List<String> images;  
    ...  
}
```



Collection de types

- La collection dans l'entité est décorée par
 - @ElementCollection
 - qui spécifie une collection de type de base ou d'objets embarqués
 - fetch : par défaut en mode LAZY
 - ATTENTION aux exceptions de type LazyInitializationException
 - @CollectionTable
 - qui spécifie la table où se trouve la donnée
 - @Column
 - qui spécifie, si nécessaire, le nom de la colonne dans la table spécifiée par @CollectionTable

Modes LAZY et EAGER

- Par défaut les collections sont en mode LAZY
 - la collection de valeurs n'est pas récupérée tant que l'appel du getteur n'est pas effectué...
 - MAIS si l'entité est détachée de l'EntityManager, le proxy mis en place par l'ORM n'a plus accès aux données
 - d'où cette `LazyInitializationException`
- Le mode EAGER permet de récupérer la collection lors de la requête récupérant l'entité
 - MAIS...

Modes LAZY et EAGER

- Si plusieurs listes (`List<T>`) sont récupérées en mode EAGER
 - risque de produit cartésien
 - une `MultiBagFetchException` est levée par Hibernate
 - Hibernate possède des annotations `@LazyCollection`, `@Fetch`
 - si une collection est de type `List<T>` et l'autre `Set<T>`
 - des doublons apparaissent dans la liste
 - => produit cartésien
 - le passage en `Set<T>` évite les doublons, mais pas le produit cartésien

Modes LAZY et EAGER

- Le mode de fonctionnement par défaut (LAZY) évite de ramener en mémoire des grappes d'objets
- Bonne pratique
 - ne rien mettre en mode EAGER
 - ou une seule collection dont on maîtrise le nombre d'objets
 - numéros de téléphone d'un contact
 - mais pas les listes de factures avec les listes de lignes d'articles facturé
 - travailler sur le pattern DTO - Data Transfert Object

Collection de type Map

- Le contact possède une collection d'URL de sites web
 - chaque site web est associé à un raccourci
 - Map<K, V> semble adapté
 - K est le type de la clé, clé unique, ici le raccourci
 - V est le type de la valeur, ici l'URL
- @MapKeyColumn permet de mapper la clé avec la colonne de la table
- @Column mappe la valeur avec la colonne de la table

Collection de type Map

- Dans la classe Contact

```
@Entity
@Table(name="personnes")
@Access(AccessType.FIELD)
public class Contact {
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="pk")
    private int id;
    private String civilité;
    private String nom;
    private String prenom;

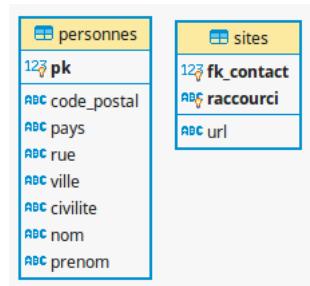
    ...

    @ElementCollection(fetch=FetchType.EAGER)
    @CollectionTable(name="sites",joinColumns=@JoinColumn(name="fk_contact"))
    @MapKeyColumn(name="raccourci")
    @Column(name="url")
    private Map<String, String> sitesWeb = new HashMap<>();

    ...
}
```

Collection de type Map

- En base



JPA Liaisons en entités

Liaisons entre entités

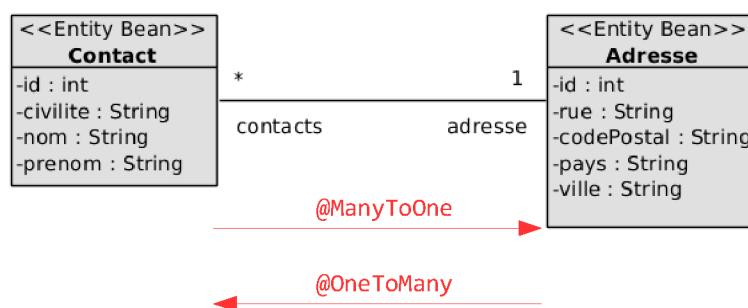
- Du point de vue de la POO deux entités peuvent être liées par une relation
 - 1:1 @OneToOne
 - 1:p @OneToMany
 - n:1 @ManyToOne
 - n:p @ManyToMany
- De plus une relation peut-être
 - unidirectionnelle
 - ou bidirectionnelle

Liaisons entre entités

- Dans les liaisons bidirectionnelles il convient de prendre en compte les cardinalités pour connaître le type de la seconde liaison
 - un contact ↔ une adresse
 - côté Contact : @OneToOne
 - côté Adresse : @OneToOne
 - un contact à une adresse, mais une adresse peut-être partagée par plusieurs contacts
 - Contact → Adresse : @ManyToOne dans Contact
 - Adresse → Contact : @OneToMany dans Adresse

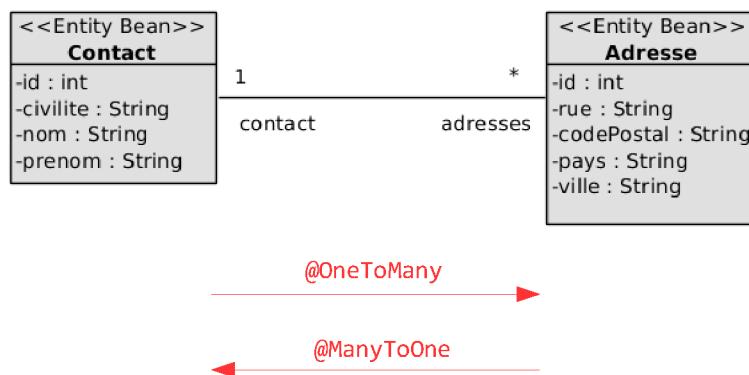
Liaisons entre entités

- Un contact à une adresse
- Une adresse peut être partagée par plusieurs contacts



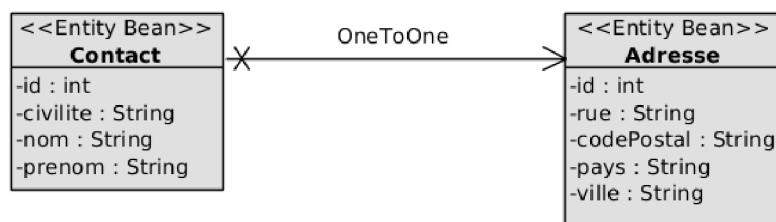
Liaisons entre entités

- Un contact a plusieurs adresses
- Une adresse est liée à un seul contact



@OneToOne unidirectionnel

- Deux classes entités
 - pas de collection



- En base deux tables
 - liaison entre les deux table
 - clé étrangère dans une ou l'autre table
 - ou table de liaison comportant les deux clés étrangères

@OneToOne unidirectionnel

- Contact **marque** son champ Adresse **avec** @OneToOne
 - **par défaut**
 - la clé étrangère de addresses est dans personnes
 - la clé étrangère est nommée *nom_champ_nom_cle*
- L'entité Adresse **n'est pas affectée**
 - c'est l'entité Contact qui est l'entité parent
 - la visibilité est de Contact **vers** Adresse

@OneToOne unidirectionnel

```
@Entity  
@Table(name="personnes")  
@Access(AccessType.FIELD)  
public class Contact {  
    @Id @GeneratedValue(strategy=GenerationType.IDENTITY)  
    @Column(name="pk_personne")  
    private int id;  
    private String civilité;  
    private String nom;  
    private String prenom;  
    @OneToOne(cascade=CascadeType.ALL)  
    private Adresse adresse;  
...  
}
```

```
@Entity  
@Table(name = "adresses")  
@Access(AccessType.FIELD)  
public class Adresse {  
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)  
    @Column(name = "pk_adresse")  
    private int id;  
    private String rue;  
    @Column(name = "code_postal", length = 10)  
    private String codePostal;  
    private String ville;  
    private String pays;  
...  
}
```

@OneToOne unidirectionnel

- Comportement par défaut : base de données

```
@OneToOne(cascade=CascadeType.ALL)
private Adresse adresse;
```

| adresses |
|-----------------|
| 123 pk_adresse |
| ABC code_postal |
| ABC pays |
| ABC rue |
| ABC ville |

| personnes |
|-----------------|
| 123 pk_personne |
| ABC civilité |
| ABC nom |
| ABC prenom |
| 123 fk_adresse |

- Pour changer le nom de la clé étrangère
 - dans l'entité Contact, utilisation de @JoinColumn

```
@OneToOne(cascade=CascadeType.ALL)
@JoinColumn(name="fk_adresse")
private Adresse adresse;
```

| adresses |
|-----------------|
| 123 pk_adresse |
| ABC code_postal |
| ABC pays |
| ABC rue |
| ABC ville |

| personnes |
|-----------------|
| 123 pk_personne |
| ABC civilité |
| ABC nom |
| ABC prenom |
| 123 fk_adresse |

@OneToOne unidirectionnel

- La table *adresses* contient la clé étrangère de *personnes*

@OneToOne unidirectionnel

- Table de jointure entre les tables *personnes* et *adresses*
 - sur l'entité Contact, description de la table de jointure
 - seul le nom de la table est obligatoire

```
@OneToOne(cascade=CascadeType.ALL),  
@JoinTable(name="personnes_adresses",  
joinColumns=@JoinColumn(name="fk_personne"),  
inverseJoinColumns=@JoinColumn(name="fk_adresse"))  
private Adresse adresse;
```



Cascade

- L'attribut `cascade` des annotations de liaison précise comment est gérée l'entité cible (*Adresse*) lorsque l'entité source (*Contact*) subit une des opérations de l'`EntityManager`
 - `detach`, `merge`, `persist`, `remove`, `refresh`
 - énumération `CascadeType`
 - `DETACH`, `MERGE`, `PERSIST`, `REMOVE`, `REFRESH`, `ALL`

Cascade

- Si un contact est lié avec une adresse
 - un appel de `contact.setAdresse(null)` doit pouvoir supprimer la donnée qui était liée
 - sinon il existerait en base des enregistrements orphelins
 - JPA 2.0 : attribut `orphanRemoval`
 - `true` ou `false`
 - sur les annotations `@OneToOne` et `@OneToMany`

@OneToOne bidirectionnel

- On peut naviguer d'une entité à l'autre
 - Contact possède une propriété de type Adresse
 - Adresse possède une propriété de type Contact
- Une des extrémités, et seulement une, est responsable de la mise à jour des colonnes de l'association
 - l'extrémité non responsable utilise l'attribut `mappedBy` sur l'annotation `@OneToOne`

@OneToOne bidirectionnel

- Entité Contact - pas de changement par rapport à la configuration unidirectionnelle

```
@OneToOne(cascade=CascadeType.ALL)
@JoinColumn(name="fk_adresse")
private Adresse adresse;
```

- Entité Adresse

- une propriété de type Contact est ajoutée
- l'attribut mappedBy référence la propriété de type Adresse dans Contact

```
@OneToOne(mappedBy="adresse")
private Contact contact;
```

@OneToOne bidirectionnel

- En base rien ne change

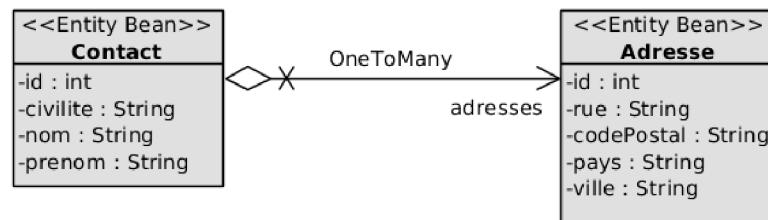


- Ne pas oublier de coder cette double association lors de la création de nouvelles instances

```
ContactDAO dao = new ContactDAO(emf);
Contact c = new Contact("M", "LAGAFFE", "Gaston");
Adresse a = new Adresse("rue de Bruxelles", "75011", "Paris");
c.setAdresse(a);
a.setContact(c);
dao.save(c);
```

@OneToMany unidirectionnel

- L'entité source possède une collection d'entités cibles
 - ici un Contact possède plusieurs Adresse
 - dans Contact la relation sera décorée par @OneToMany



@OneToMany unidirectionnel

- En base :
 - les clés étrangères des enregistrements de la table "personnes" peuvent être
 - dans la tables "adresses"
 - dans une table de jointure

@OneToMany unidirectionnel

- Entité Adresse : pas de propriété de type Contact
- Entité Contact : une collection d'Adresse
- Annotation @OneToMany seule
 - comportement Hibernate par défaut :
 - création d'une table de jonction

```
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)  
private Set<Adresse> adresses = new HashSet<>();
```



@OneToMany unidirectionnel

- `@JoinTable` : personnalisation de la table de jointure
 - `name` : nom de la table
 - `joinColumn` : définition de la colonne pour la clé étrangère de l'entité source (Contact)
 - `inverseJoinColumn` : définition de la colonne pour la clé étrangère de l'entité cible (Adresse)

@OneToMany unidirectionnel

- Contact : ajout de @JoinTable

```
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
@JoinTable(name="adresses_contacts",
    joinColumns=@JoinColumn(name="fk_personne"),
    inverseJoinColumns=@JoinColumn(name="fk_adresse"))
private Set<Adresse> adresses = new HashSet<>();
```



@OneToMany unidirectionnel

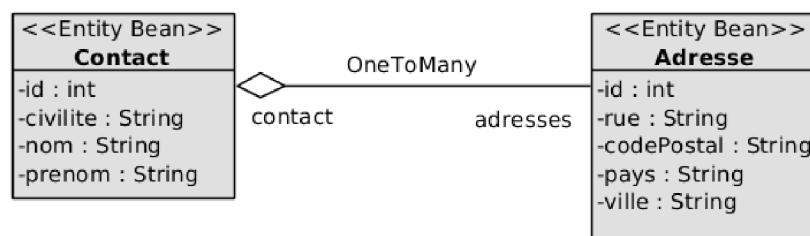
- Positionnement des clés étrangères des enregistrements de la table "personnes" dans la table "adresses"
- Dans Contact : définition de la colonne de jointure par @JoinColumn

```
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
@JoinColumn(name="fk_personne")
private Set<Adresse> adresses = new HashSet<>();
```



@OneToMany bidirectionnel

- On peut naviguer entre les deux entités
 - Contact possède une collection d'Adresse
 - Adresse possède une propriété de type Contact



@OneToMany bidirectionnel

- Ne change pas le schéma de la base
- Comme pour @OneToOne bidirectionnel, il faut ajouter dans Adresse
 - une propriété de type Contact
 - l'annotation @ManyToOne
 - Contact → Adresse : @OneToMany
 - un contact pour plusieurs adresses
 - Adresse → Contact : @ManyToOne
 - plusieurs adresses pour un contact

@OneToMany bidirectionnel

- Personnalisation de la table de liaison
 - la table de jointure ne peut pas être définie des deux côtés
 - sur les deux entités
 - il faut définir la table de jointure sur une entité, et mapper l'autre entité sur celle qui définit la table de jointure
 - l'attribut `mappedBy` n'existe pas dans `@ManyToOne`
 - il faut donc mapper `Contact` vers la définition de la table de jointure dans `Adresse`

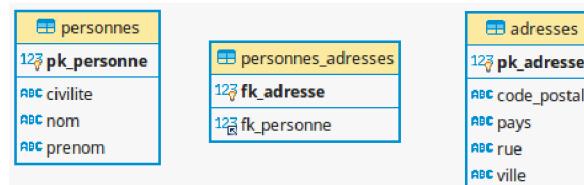
@OneToMany bidirectionnel

- Contact

```
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER, mappedBy="contact")
private Set adresses = new HashSet<>();
```

- Adresse

```
@ManyToOne(fetch=FetchType.EAGER)
@JoinTable(name="personnes_adresses",
    joinColumns=@JoinColumn(name="fk_adresse"),
    inverseJoinColumns=@JoinColumn(name="fk_personne"))
private Contact contact;
```



@OneToMany bidirectionnel

- Sans table de jointure, le résonnement est le même
 - chaque enregistrement de la table "adresses" contient la clé étrangère vers un enregistrement de la table "personnes"
 - Contact

```
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER, mappedBy="contact")
private Set<Adresse> adresses = new HashSet<>();
```

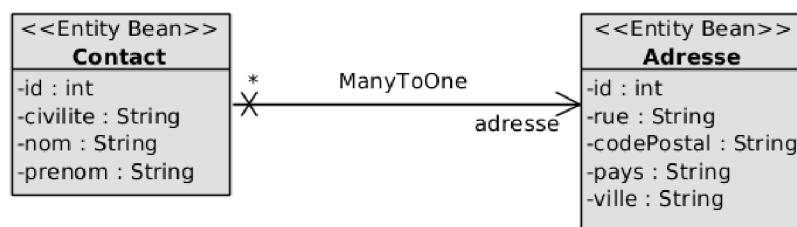
- Adresse

```
@ManyToOne(fetch=FetchType.EAGER)
@JoinColumn(name="fk_personne")
private Contact contact;
```



@ManyToOne unidirectionnel

- L'entité source possède une propriété de l'entité cible
- L'entité cible est potentiellement partagée entre plusieurs entités sources
 - un contact à une adresse
 - à une même adresse habitent plusieurs contacts



@ManyToOne unidirectionnel

- La navigabilité est de Contact vers Adresse
 - Contact à une propriété de type Adresse
 - cette propriété sera annotée par @ManyToOne
 - Adresse n'a pas de propriété de type Contact

@ManyToOne unidirectionnel

- Comportement par défaut
 - Contact

```
@ManyToOne  
private Adresse adresse;
```

- la clé étrangère de l'adresse est dans la table "personnes"



@ManyToOne unidirectionnel

- Précautions

- l'entité Adresse est partagée entre plusieurs Contact
- la sauvegarde de Adresse doit être gérée avec soin

```
Contact c1 = new Contact("M", "LAGAFFE", "Gaston");
Contact c2 = new Contact("Mme", "BLANC SEC", "Adèle");
Adresse a = new Adresse("rue de Bruxelles", "75011", "Paris");
c1.setAdresse(a);
c2.setAdresse(a);
dao.save(c1);
dao.save(c2);
```

- @ManyToOne (cascade=CascadeType.ALL), l'adresse sera persistée 2 fois
 - exception javax.persistence.PersistenceException

@ManyToOne unidirectionnel

- Le DAO doit s'arranger pour gérer les persistances
 - extrait de la classe ContactDAO

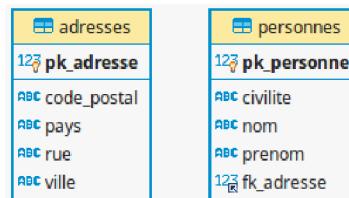
```
public void save(Contact contact) {
    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();
    this.saveOrUpdate(contact.getAdresse(),em); ←
    em.persist(contact);
    em.getTransaction().commit();
    em.close();
}

private void saveOrUpdate(Adresse adresse, EntityManager em) {
    if(adresse.getId()==0) {
        em.persist(adresse);
    }else {
        em.merge(adresse);
    }
}
```

@ManyToOne unidirectionnel

- @JoinColumn pour personnaliser le nom de la colonne contenant la clé étrangère
 - classe Contact

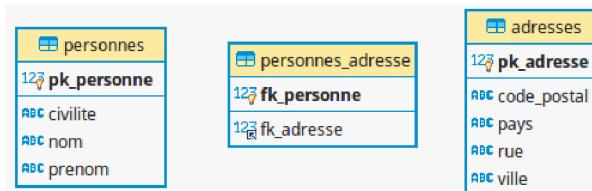
```
@ManyToOne  
@JoinColumn(name="fk_adresse")  
private Adresse adresse;
```



@ManyToOne unidirectionnel

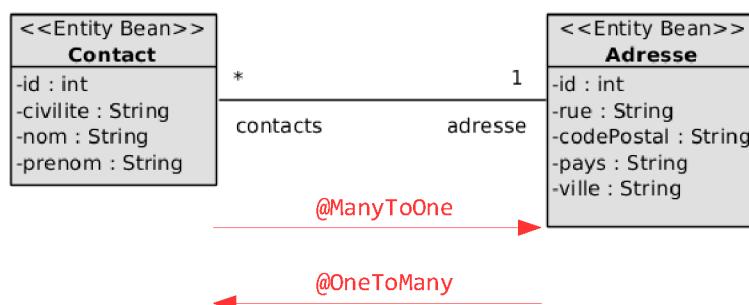
- Avec table de jointure
 - Contact contient la description de la table de jointure

```
@ManyToOne  
@JoinTable(name="personnes_adresse",  
joinColumns=@JoinColumn(name="fk_personne"),  
inverseJoinColumns=@JoinColumn(name="fk_adresse"))  
private Adresse adresse;
```



@ManyToOne bidirectionnel

- Contact **contient le** @ManyToOne
 - il contient la description de la liaison
 - table ou colonne de jointure
- Adresse **contient** @OneToMany
 - avec un mappedBy vers Contact



@ManyToOne bidirectionnel

- Contact - rien en change
 - la description de la jointure est dans cette classe
- Adresse
 - ajout de la collection de contacts
 - décoration par @OneToMany avec mappedBy vers Contact

```
@OneToMany(mappedBy="adresse")
private List<Contact> contacts = new ArrayList<>();
```

@ManyToMany unidirectionnel

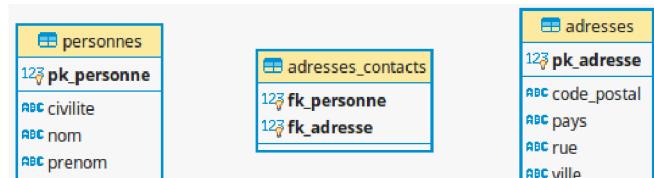
- Un contact peut avoir plusieurs adresses
- Une adresse peut-être partagée par plusieurs contacts
- Une table de jointure est obligatoire



@ManyToMany unidirectionnel

- Contact
 - porte @ManyToMany
 - décrit la jointure

```
@ManyToMany(fetch=FetchType.EAGER)
@JoinTable(name="adresses_contacts",
    joinColumns=@JoinColumn(name="fk_personne"),
    inverseJoinColumns=@JoinColumn(name="fk_adresse"))
private Set<Adresse> adresses = new HashSet<>();
```



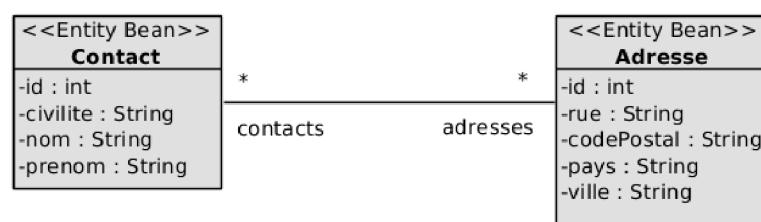
@ManyToMany unidirectionnel

- Toujours les mêmes précautions dans le DAO
 - extrait de ContactDAO

```
public void save(Contact contact) {  
    EntityManager em = emf.createEntityManager();  
    em.getTransaction().begin();  
    for(Adresse a : contact.getAdresses()) {  
        saveOrUpdate(a, em);  
    }  
    em.persist(contact);  
    em.getTransaction().commit();  
    em.close();  
}  
  
private void saveOrUpdate(Adresse adresse, EntityManager em) {  
    if(adresse.getId()==0) {  
        em.persist(adresse);  
    }else {  
        em.merge(adresse);  
    }  
}
```

@ManyToMany bidirectionnel

- Navigabilité entre les deux classes
 - via des collections



@ManyToMany bidirectionnel

- L'entité Contact ne change pas

- elle décrit la jointure

```
@ManyToMany(fetch=FetchType.EAGER)
@JoinTable(name="adresses_contacts",
joinColumns=@JoinColumn(name="fk_personne"),
inverseJoinColumns=@JoinColumn(name="fk_adresse"))
private Set<Adresse> adresses = new HashSet();
```

- Entité Adresse

- ajout de la collection de contacts
 - décoration par @ManyToMany avec un mappedBy vers la propriété adresses dans Contact

```
@OneToMany(mappedBy="adresses")
private List<Contact> contacts = new ArrayList();
```

Conclusion

- Choisir le mapping adapté
 - @OneToOne, @OneToMany, @ManyToOne, @ManyToMany
- Choisir sa structure de jointure
 - @JoinTable, @JoinColumn
- Choisir la navigabilité
 - et sur l'entité cible adapter le mapping
- Mettre en place la référence vers l'entité définissant la jointure
 - attribut mappedBy

Conclusion

- Attributs des mappings
 - tous les attributs ne sont pas représentés, cf. la javadoc

| | mappedBy | cascade | fetch | optional | orphanRemove |
|-------------|----------|---------|-------|----------|--------------|
| @OneToOne | OUI | OUI | OUI | OUI | OUI |
| @OneToMany | OUI | OUI | OUI | NON | OUI |
| @ManyToOne | NON | OUI | OUI | OUI | NON |
| @ManyToMany | OUI | OUI | OUI | NON | NON |

JPA

Mapper les héritages

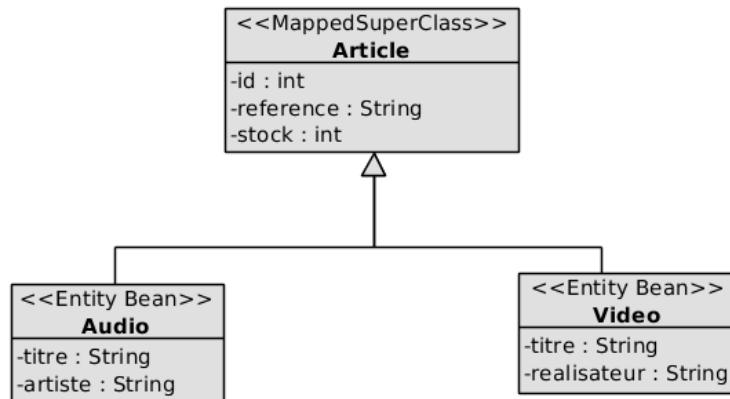
Mapper les héritage

- JPA propose
 - d'hériter d'une classe mère de propriétés communes
 - @MappedSuperClass
 - n'impacte pas la base de données
 - donc purement Java POO
 - de mettre en place des stratégies de structure de tables pour permettre de faire un héritage en base de données
 - trois stratégies qui correspondent à une représentation différente dans le modèle relationnel



@MappedSuperClass

- Partage de propriétés entre plusieurs entités
 - la classe mère n'est pas une entité
 - factorisation de propriétés devant être enregistrées en base
 - pas de table propre



@MappedSuperClass

- Classe Article

```
@MappedSuperclass
public abstract class Article
{
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private String numero;
    private int stock;
    ...
}
```

- Classe Audio

```
@Entity
@Table(name="cd_audios")
public class Audio extends Article
{
    private String titre;
    private String artiste;
    ...
}
```

@MappedSuperClass

- Les propriétés de la classe mère sont héritées dans les classes filles
 - la plupart des annotations ne sont pas héritées
 - sauf si l'annotation est marquée par la méta-annotation `@Inherited`
 - ce mécanisme permet de récupérer
 - les propriétés Java
 - les annotations JPA qui s'y rapportent

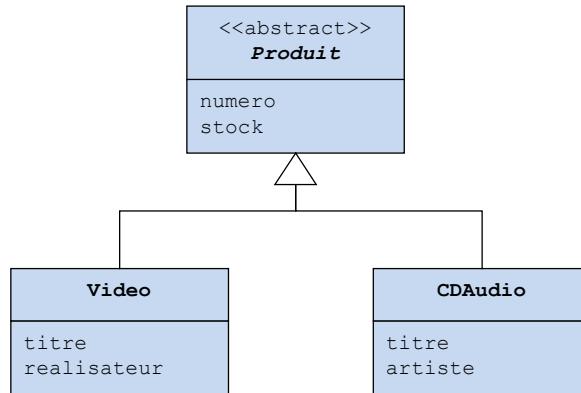
@MappedSuperClass

- En base



Mapper les héritages

- Trois stratégies pour mapper l'héritage
 - une table par classe concrète
 - une seule table pour la hiérarchie de classe
 - une table par classe fille



Une table par classe concrète

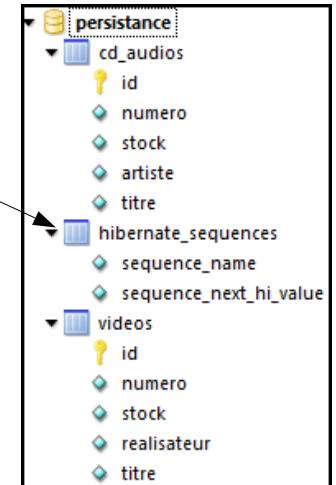
- La stratégie TABLE_PER_CLASS est mise en place dans l'annotation `@Inheritance`
- La plupart des ORM implémentent cette stratégie en utilisant des UNION SQL pour le polymorphisme
- Le type de stratégie de création de la clé primaire est liée à l'implémentation

Une table par classe concrète

```
@Entity  
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)  
public abstract class Produit  
{  
    @Id  
    @GeneratedValue(strategy=GenerationType.TABLE)  
    private int id;  
    ...
```

```
@Entity  
@Table(name="cd_audios")  
public class CDAudio extends Produit  
{  
    ...
```

```
@Entity  
@Table(name="videos")  
public class Video extends Produit  
{  
    ...
```



Une table pour la hiérarchie de classes

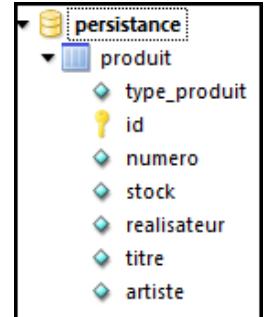
- La stratégie SINGLE_TABLE est mise en place dans l'annotation @Inheritance
- Une seule table est utilisée
 - requêtes polymorphiques plus rapides
 - colonnes "nullables"

Une table pour la hiérarchie de classes

```
@Entity  
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)  
@DiscriminatorColumn(name="type_produit")  
public abstract class Produit  
{  
    @Id  
    @GeneratedValue(strategy=GenerationType.AUTO)  
    private int id;  
    ...
```

```
@Entity  
@Table(name="cd_audios")  
public class CDAudio extends Produit  
{ ... }
```

```
@Entity  
@Table(name="videos")  
@DiscriminatorValue(value="video")  
public class Video extends Produit  
{ ... }
```



| type_produit | id | numero | stock | realisateur | titre | artiste |
|--------------|----|--------|-------|-----------------|------------------|------------|
| video | 1 | v1 | 3 | Stanley KUBRICK | Orange mécanique | NULL |
| CDAudio | 2 | cd1 | 10 | NULL | The Wall | Pink Floyd |

Une table par classe fille

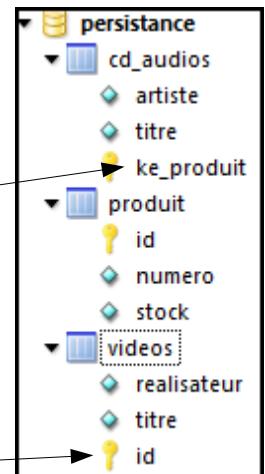
- La stratégie JOINED est mise en place dans l'annotation @Inheritance
- Une table regroupe les propriétés commune
- Les identifiants des tables de classes filles correspondent aux clés primaire de la classe mère

Une table par classe fille

```
@Entity  
@Inheritance(strategy=InheritanceType.JOINED)  
public abstract class Produit  
{  
    @Id  
    @GeneratedValue(strategy=GenerationType.AUTO)  
    private int id;  
    ...
```

```
@Entity  
@Table(name="cd_audios")  
@PrimaryKeyJoinColumn(name="ke_produit")  
public class CDAudio extends Produit  
{ ... }
```

```
@Entity  
@Table(name="videos")  
public class Video extends Produit  
{ ... }
```



JPA

Les requêtes JPQL

Les requêtes

- Plusieurs types de requêtes sont disponibles
 - requêtes natives
 - procédures stockées
 - requêtes JPQL
 - requêtes par Criteria API
- Java Persistence Query Language
 - sémantiquement très proche de SQL
 - mais orienté objet



Langage JPQL

- JPQL : Java Persistence Query Language
- Langage similaire à SQL
 - mais JPQL permet une approche objet du requêtage
 - SQL est basé sur la connaissance des structures de la base
 - JPQL est basé sur la connaissance du modèle objet
- La recherche des adresses est effectuée par
 - SELECT * FROM table_adresses en SQL
 - from Adresse en JPQL
 - attention certaines implémentations imposent un SELECT

Terminologie

- Quatre catégories de requêtes
 - sélection
 - retourne des entités avec filtrage du résultat
 - agrégation
 - groupe les résultats pour produire une donnée (somme, ...)
 - mise à jour
 - suppression

Création des requêtes

- EntityManager : méthodes principales de création de requête
 - createNativeQuery
 - createStoreProcedureQuery
 - createNamedStoreProcedureQuery
 - createQuery
 - createNamedQuery
 - certaines signature acceptent le type de l'entité retour
 - évite le cast Object → entité

Création de requêtes

- Les méthodes de Query permettent
 - d'initialiser les paramètres de requête
 - setParameter()
 - renvoie l'instance de Query : possibilité de chaîner
 - de récupérer un résultat
 - getResultList()
 - getSingleResult()
 - getResultStream()
 - d'exécuter une mise à jour ou une suppression
 - executeUpdate()

Requête native

- A éviter
 - non portable
 - non orienté objet
 - compliquée à mettre au point
 - **risque d'injection SQL**
- Une requête native renvoie
 - List<Object [] >
 - ou List<E> si le type de l'entité est précisé

Requête native

- Exemples

```
public List<Adresse> getAdresseByVille(String ville){  
    String sql = "SELECT * FROM adresses WHERE ville=?";  
    EntityManager em = emf.createEntityManager();  
    List<Adresse> adresses = em.createNativeQuery(sql, Adresse.class)  
        .setParameter(1, ville)  
        .getResultList();  
    em.close();  
    return adresses;  
}
```

```
public List<Object[]> getCodePostalAndPays(String ville){  
    String sql = "SELECT a.code_postal, a.pays FROM adresses a WHERE ville=?";  
    EntityManager em = emf.createEntityManager();  
    List<Object[]> resultat = em.createNativeQuery(sql)  
        .setParameter(1, ville)  
        .getResultList();  
    em.close();  
    return resultat;  
}
```

Requêtes JPQL

- Sélection

- syntaxe

```
SELECT <expression> FROM <from_clause>
[WHERE <where_clause>]
[ORDERED BY <ordered_clause>]
```

- La requête la plus simple est

```
SELECT c FROM Contact c
```

- retourne la liste de tous les contacts
 - notez bien l'utilisation de la classe Contact

Requêtes JPQL

- Sélection

- il est possible de récupérer un sous ensemble de l'entité

```
SELECT c.prenom, c.nom FROM Contact c
```

- retourne List<Object[]>

- expression constructeur NEW

- permet de construire des instances non entité

```
SELECT NEW org.antislashn.contact.NomPrenom(c.nom,c.prenom)
```

- FROM Contact c

- retourne List<NomPrenom>

- expression DISTINCT

Requêtes JPQL

- WHERE
 - peut comporter
 - [NOT] LIKE, [NOT] BETWEEN, [NOT] IN
 - AND, OR
 - [NOT] EXISTS
 - ALL, SOME/ANY
 - IS [NOT] NULL
 - IS [NOT] EMPTY
 - [NOT] MEMBER OF : pour les collections

Requêtes JPQL

- HAVING
 - doit porter sur l'expression de regroupement GROUP BY
 - ou une fonction de regroupement
 - AVG(), COUNT(), MAX(), MIN(), SUM()
- sous-requête
 - les clauses WHERE et HAVING peuvent contenir des sous-requêtes

```
SELECT e FROM Employe e WHERE e.salaire >=
    (SELECT e2.salaire FROM Employe e2
        WHERE e2.departement=75)
```

Requêtes JPQL

- La plupart des fonctions, clauses SQL sont utilisables avec JPQL
 - cf. : <https://sql.sh/>
- Les jointures sont simplifiées
 - mais elles peuvent aussi être explicites
 - <https://sql.sh/cours/jointures>

Exemple : pagination avec JPQL

- Objectif : parcourir la liste des adresses avec pagination
- L'API nous propose
 - `setFirstResult (int)` : positionne l'offset pour la pagination
 - `setMaxResults (int)` : initialise le nombre de résultat à récupérer

Exemple : pagination avec JPQL

- AdresseDAO
 - méthode permettant de récupérer le nombre d'adresses

```
public long count() {  
    EntityManager em = emf.createEntityManager();  
    long r = (Long) em.createQuery("SELECT COUNT(a.id) FROM Adresse a").getSingleResult();  
    em.close();  
    return r;  
}
```

- méthode gérant la pagination

```
public List<Adresse> getAdresses(int from, int max){  
    EntityManager em = emf.createEntityManager();  
    List<Adresse> adresses = em.createQuery("FROM Adresse",Adresse.class)  
        .setFirstResult(from)  
        .setMaxResults(max)  
        .getResultList();  
    em.close();  
    return adresses;  
}
```

Exemple : pagination avec JPQL

- Utilisation du DAO

```
int pageSize = 30;  
long nbResults = adresseDAO.count();  
for(int pageNumber=0 ; pageNumber*pageSize <= nbResults ; pageNumber++) {  
    System.out.println("== PAGE "+(pageNumber+1));  
    List<Adresse> adresses = adresseDAO.getAdresses(pageNumber*pageSize,pageSize);  
    adresses.forEach(System.out::println);  
}
```

Mode batch

- JPA ne fournit pas de mode batch
 - JDBC en fournit un
 - mais les driver peut ne pas l'implémenter
 - les implémentations peuvent en fournir
 - pour hibernate
 - ```
<property name="hibernate.jdbc.batch_size" value="25" />
<property name="hibernate.order_inserts" value="true" />
```
    - ou les frameworks (Spring)
- JPA définit deux niveaux de cache
  - L1 : le cache de l'EntityManager
  - L2 : le cache de l'EntityManagerFactory

# Mode batch

- Exemple de batch

```
public void save(List<Contact> contacts) {
 int batchSize = getBatchSize();
 EntityManager em = emf.createEntityManager();
 EntityTransaction tx = em.getTransaction();
 tx.begin();
 int count = 0;
 for(Contact c : contacts) {
 em.persist(c);
 count++;
 if(count%batchSize == 0) {
 em.flush();
 em.clear();
 }
 }
 tx.commit();
 em.close();
}
```

# Mode batch

- EntityManager
  - flush() : synchronise le contexte de persistance en base
  - clear() : détache les entités
    - évite les OutOfMemoryException

## Requêtes JPQL

- Deux façons de créer les requêtes
  - via createQuery()
    - la requête est sous forme de chaîne de caractères

```
String jpql = "SELECT NEW org.antislashn.contact.NomPrenom(c.nom,c.prenom) FROM Contact c";
List<NomPrenom> result = em.createQuery(jpql, NomPrenom.class).getResultList();
```

- via createNamedQuery()
  - récupération dans les requêtes déclarées

```
@NamedQueries({
 @NamedQuery(name="Contact.getAllContacts",query="SELECT c FROM Contact c")
})
```

```
List<Contact> contacts = em.createNamedQuery("Contact.getAllContacts", Contact.class).getResultList();
```

# ReqêtesJPQL

- Les requêtes nommées sont
  - compilées au démarrage de l'application
    - mises dans le statementCache
    - transformées en PreparedStatement si le driver JDBC le peut
- Avantages
  - centralisation des requêtes
  - validées au démarrage

# JPA

## Clés simples et clés composées

### Identifiant

- Toute entité possède un identifiant
- Il est plus simple de gérer un clé constituée d'une seule valeur qu'une clé composée
  - clé artificielle ou surrogate key
  - clé produite à partir d'une séquence
    - différentes stratégies
      - natives aux bases
      - ou non, et alors JPA peut alors fournir une stratégie

# Identifiant

- Propriété identifiant de type séquence
  - type Java long
  - annotée par @Id
  - production par @GeneratedValue, attribut strategy
    - GenerationType.AUTO
      - l'implémentation génère la valeur
        - hibernate crée une table hibernate-sequence
    - GenerationType.IDENTITY
      - utilise le type de colonne spécial de la base de données
        - MySQL => AUTO\_INCREMENT

# Identifiant

- Stratégie TABLE
  - utilise une table dédiée qui stocke les clés des tables
  - utilise l'annotation @TableGenerator
  - exemple
    - une table "indexes" est créée, qui contient le nom de chaque table associée à la valeur de son prochain index

```
@Id
@GeneratedValue(strategy=GenerationType.TABLE,generator="indexes")
@TableGenerator(name="indexes")
@Column(name = "pk")
private int id;
```

# Identifiant

- Stratégie SEQUENCE

- utilise la stratégie de séquence proposé par certaines bases de données
- utilise l'annotation `@SequenceTableGenerator`
  - positionné sur la classe de l'entité

```
@Entity
@Table(name = "adresses")
@Access(AccessType.FIELD)
@SequenceGenerator(name="ADR_SEQ",sequenceName="ADR_SEQ")
public class Adresse {
 @Id
 @GeneratedValue(strategy=GenerationType.SEQUENCE,generator="ADR_SEQ")
 @TableGenerator(name="indexes")
 @Column(name = "pk")
 private int id;
```

## Clé composée

- Deux stratégies de mise en place

- annotation `@IdClass` sur l'entité
  - `@IdClass` précisera sur l'entité concernée la classe représentant la clé composée
    - la classe qui modélise la clé composée est une classe Java non annotée
- utilisation d'une classe annotée par `@Embedded` comme identifiant
  - la propriété est annotée par `@Id`

# Clé composée

- Une clé composée est un POJO
  - constructeur par défaut
  - implémente Serializable
  - définit les méthodes hashCode() et equals()

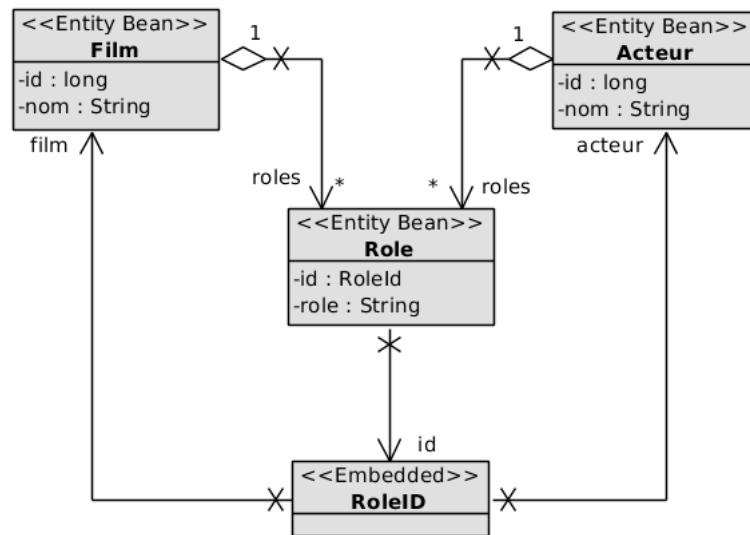
## Clé composée @Embedded

- Modélisation d'une base de données de film
  - films, acteurs, rôles joués, ...



# Clé composée @Embedded

- Diagramme de classe



# Clé composée @Embedded

- La table de rôles possède une clé composée
  - le couple (fk\_acteur,fk\_film) est la clé composée primaire
- La clé composée est modélisée sous forme d'un objet embarqué

```
@Embeddable
public class RoleId implements Serializable{
 @ManyToOne
 @JoinColumn(name="fk_acteur")
 private Acteur acteur;

 @ManyToOne
 @JoinColumn(name="fk_film")
 private Film film;

 ...
}
```

# Clé composée @Embedded

- L'entité Role possède un identifiant id de type RoleId

```
@Entity
@Table(name="roles")
@Access(AccessType.FIELD)
public class Role {
 @Id
 private RoleId id;
 private String role;

 ...
}
```

# Clé composée @Embedded

- Entités Film et Acteur

```
@Entity
@Table(name="films")
@Access(AccessType.FIELD)
public class Film {
 @Id
 @GeneratedValue(strategy=GenerationType.IDENTITY)
 @Column(name="pk_film")
 private long id;
 private String nom;
 @OneToMany(mappedBy="id.film",cascade=CascadeType.ALL)
 private Set<Role> roles = new HashSet<>();

 ...
}
```

```
@Entity
@Table(name="acteurs")
@Access(AccessType.FIELD)
public class Acteur {
 @Id
 @GeneratedValue(strategy=GenerationType.IDENTITY)
 @Column(name="pk_acteur")
 private long id;
 private String nom;
 @OneToOne(mappedBy="id.acteur",cascade=CascadeType.ALL)
 private Film film;

 ...
}
```

# Clé composée @IdClass

- Exemple

- une entité Personne utilise les propriétés nom et prenom comme clé primaire
  - très mauvaise idée...
- chaque champ est marqué avec @Id
- une classe PersonnePK représente la clé primaire
  - pas de génération automatique des valeurs

# Clé composée @IdClass

- Entité Personne

```
@Entity
@Table(name="personnes")
@Access(AccessType.FIELD)
@IdClass(PersonnePK.class)
public class Personne {
 private String civilité;
 @Id @Column(length=100)
 private String nom;
 @Id @Column(length=100)
 private String prenom;
 ...
}
```

# Clé composée @IdClass

- Classe PersonnePK
  - doit posséder hashCode() et equals()

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

 public PersonnePK() {}

 ...
}
```

# Clé composée @IdClass

- Il n'y pas de génération automatique de PersonnePK
  - cette classe est notamment utilisée lors de recherche
  - extraits du DAO

```
public void save(Personne personne) {
 EntityManager em = emf.createEntityManager();
 em.getTransaction().begin();
 em.persist(personne);
 em.getTransaction().commit();
 em.close();
}

public Personne findPersonneById(String nom, String prenom) {
 EntityManager em = emf.createEntityManager();
 PersonnePK pk = new PersonnePK(nom, prenom); ←
 Personne personne = em.find(Personne.class, pk); ←
 em.close();
 return personne;
}
...
```

# JPA Criteria API

## Criteria API

- Autre manière de créer des requêtes
  - les requêtes JPQL sont créées avec des chaînes de caractères
    - ce qui peut être complexe lorsqu'il faut créer des requêtes dynamiquement
      - concaténation de String
  - les requêtes basées sur Criteria sont créées par des appels à des méthodes
    - la requête ainsi créée est ensuite exécutée classiquement



# Criteria API

- Exemple de requête simple par Criteria
  - équivalent à SELECT d FROM Destination d

```
EntityManagerFactory entityManagerFactory =
Persistence.createEntityManagerFactory("bovoyage");
EntityManager entityManager = entityManagerFactory.createEntityManager();

CriteriaBuilder criteriaBuilder = entityManager.getCriteriaBuilder();

CriteriaQuery<Destination> criteriaQuery =
criteriaBuilder.createQuery(Destination.class);
Root<Destination> d = criteriaQuery.from(Destination.class);
criteriaQuery.select(d);

TypedQuery<Destination> query = entityManager.createQuery(criteriaQuery);

List<Destination> destinations = query.getResultList();
```

# Criteria API

- CriteriaBuilder : fabrique pour les requêtes et éléments de requêtes basés sur Criteria API
  - peut être obtenu auprès de l'EntityManagerFactory ou EntityManager
- CriteriaQuery : la requête qui devra être créée
  - obtenu via le CriteriaBuilder
- Root : représente la clause FROM de la requête

# Criteria API

- Une fois la requête Criteria créée

```
criteriaQuery.select(d);
```

- Elle peut-être utilisée via l'EntityManager

```
TypedQuery<Destination> query =
entityManager.createQuery(criteriaQuery);
List<Destination> destinations = query.getResultList();
```

# Criteria API

- Si pour des requêtes simples Criteria semble plus bavard, cette API prend toute sa puissance avec des requêtes créées dynamiquement
  - méthodes de CriteriaQuery
    - select(), where(), orderBy(), groupBy(), ...
  - utilisation des paramètres
    - ParameterExpression
  - cf. documentation

# JPA

## Principe des transactions

## Verrouillages

- Plusieurs utilisateurs peuvent accéder en même temps à une même base de données
  - lectures, écritures, suppressions
- Le verrouillage consiste à bloquer l'accès à un niveau de données pour toutes les requêtes
  - pour un temps déterminé : la requête en cours
  - le niveau de verrouillage peut-être
    - la table, la vue, l'enregistrement
    - dépend de la base de données

# Gestion de la concurrence

- Les traitements métiers nécessitent une série d'interactions avec l'utilisateur
- Ces interactions sont entrecoupées d'accès à la base de données
- Dans les applications distribuées, une transaction en base ne doit pas se dérouler sur plusieurs interactions avec l'utilisateur

## Les transactions

- Une transaction représente une série de traitements élémentaires
  - un SELECT, une série de UPDATE, un batch de mise à jour de données, ...
  - exemple : virement de 100 € d'un compte courant vers un compte épargne
    - étape 1 : débit de 100 € du compte courant
    - étape 2 : crédit de 100 € vers le compte épargne
    - que ce passe-t-il si un problème survient entre les étapes 1 et 2 ?

# Les transactions

- Une transaction doit respecter les propriétés ACID
  - **A**ttomiticité
  - **C**ohérence
  - **I**solation
  - **D**urabilité
- Exemple : processus de réservation d'un trajet de train Paris → Carnac
  - pas de train direct => deux trajets
    - Paris → Rennes et Rennes → Carnac
  - paiement du billet

# Les transactions

- **Atomicité** : la transaction doit être entièrement réalisée, sinon elle est annulée
  - c'est une unité de travail unique et indivisible
- Exemple du trajet Paris → Carnac
  - la réservation Paris → Rennes a été effectué
  - la réservation Rennes → Carnac **n'a pas pu être effectué**
  - la transaction est **abandonnée** et la réservation Paris → Rennes n'est pas enregistrée dans la base

# Les transactions

- **Cohérence** : l'état de la base est cohérente si toutes les contraintes d'intégrité sur les données sont satisfaites
  - une transaction fait passer une base de données d'un état cohérent à un nouvel état cohérent
- Exemple du trajet Paris → Carnac
  - si une défaillance a pour conséquence que le trajet Rennes → Carnac ne soit pas enregistré
  - le client possède son billet mais la réservation est incomplète en base

# Les transactions

- **Isolation** : une transaction en doit pas montrer son effet aux autres transactions avant sa validation
  - durant la transaction, les données ne peuvent pas être utilisées par une autre transaction
- Exemple du trajet Paris → Carnac
  - que ce passe-t-il si une réservation R1 prend la dernière place sur le trajet Paris → Rennes ?
  - une transaction concurrente R2 sur ce même trajet ne peut pas connaître l'état de la base avant que la réservation R1 ne soit validée
- Le standard SQL définit 4 niveaux d'isolation

# Les transactions

- **Durabilité** : une fois la transaction validée, la base reste dans un état cohérent même en cas de défaillance du système
- Exemple du trajet Paris → Carnac
  - une fois le commit effectué, aucune défaillance ne peut changer l'état de la base
  - que ce passerait-il si le client avait son billet mais qu'une défaillance d'un des trajets disparaissait ?

## Niveaux d'isolation des transactions

- Les transactions concurrentes peuvent générer des phénomènes indésirables (violations)
  - lecture sale - dirty read
    - une transaction lit des données écrites par une transaction non validée concurrente
  - lecture non reproductible - non-repeatable read
    - une transaction relit des données qu'elle a lu précédemment et ces données ont été modifiées par une autre transaction (validée depuis la lecture initiale)
  - lecture fantôme - phantom read
    - une transaction ré-exécute une requête sur une condition de recherche, et trouve un nombre de ligne différent du fait de la validation d'une autre transaction

# Dirty Read

## Transaction A

```
UPDATE voyageurs SET
age=12 WHERE id=1;
```

rollback

## Transaction B

```
SELECT * FROM voyageurs
WHERE id=1;
```

L'enregistrement de la transaction B est "sale"

# Non-Repeatable Read

## Transaction A

```
SELECT * FROM voyageurs
WHERE id=1;
```

```
SELECT * FROM voyageurs
WHERE id=1;
```

Les deux lectures renvoient des valeurs différentes pour le même enregistrement

## Transaction B

```
UPDATE voyageurs SET
age=12 WHERE id=1;
```

commit

# Phantom Read

## Transaction A

```
SELECT * FROM voyageurs
WHERE age>10 AND age<20;
```

```
SELECT * FROM voyageurs
WHERE age>10 AND age<20;
```

Les deux requêtes renvoient  
un nombre d'enregistrements  
différent

## Transaction B

```
INSERT INTO voyageurs
(age) VALUES (18);
```

commit

## Niveaux d'isolation des transactions

- SQL définit 4 niveaux d'isolation afin d'empêcher les violations précédentes

| Niveau d'isolation                                  | Lecture<br>sale | Lecture non<br>reproductible | Lecture<br>fantôme |
|-----------------------------------------------------|-----------------|------------------------------|--------------------|
| Read Uncommitted<br>Lecture de données non validées | Possible        | Possible                     | Possible           |
| Read Committed<br>Lecture de données validées       | Impossible      | Possible                     | Possible           |
| Repeatable Read<br>Lecture répétée                  | Impossible      | Impossible                   | Possible           |
| Serializable<br>Sérialisable                        | Impossible      | Impossible                   | Impossible         |

# Niveaux d'isolation des transactions

- Consultez la documentation de votre base de données pour connaître le niveau d'isolation par défaut
  - ex. : PostgreSQL : Read Committed
- Plus le niveau d'isolation est élevé, moins la performance est présente
  - à confirmer par la documentation de votre base
  - Serializable émule les exécutions des transactions en série plutôt qu'en parallèle
    - les applications doivent prendre en compte les tentatives de ré-essai en cas d'échec de la transaction

## JPA

### URL de connexion aux base de donnée

## MySQL

- Drivers
  - MySQL 5  
`com.mysql.jdbc.Driver`
  - MySQL 8  
`com.mysql.cj.jdbc.Driver`

# MySQL - URL

- `jdbc:mysql://[host][:port]/<nom_base>?[params]`
  - host : ip ou nom du serveur
    - valeur par défaut : localhost
  - port : port d'écoute du serveur
    - valeur par défaut : 3306
  - params est de la forme
    - `param=value`
    - les paires `param=value` sont séparées par le caractère ;
      - si le fichier de configuration est du XML le caractère ; est remplacé par `&`;

# MySQL - URL

- Exemples - nom de la base de donnée : contacts
  - `jdbc:mysql://localhost:3306/contacts`
    - équivalent à
      - `jdbc:mysql:///contacts`
      - `jdbc:mysql://localhost/contacts`
  - si le fuseau horaire du serveur n'est pas paramétré
    - `jdbc:mysql://localhost:3306/contacts?serverTimezone=UTC`

# Derby en mode serveur

- Derby est installé avec certains JDK, il est alors situé dans `/path/to/jdk/db`
- Le serveur est lancé en ligne de commande :  
`/path/to/derby/bin/startNetworkServer`
  - port d'écoute par défaut : 1527
- driver Derby en mode serveur  
`org.apache.derby.jdbc.ClientDriver`
  - la librairie `derbyclient.jar` du driver est située dans le répertoire `/path/to/derby/lib`

## Derby - URL

- Les bases de données Derby sont placées dans un répertoire, situé sur le système de fichier
    - pas de répertoire prédéfini
    - il faut fournir le nom de ce répertoire
- `jdbc:derby://localhost:1527/</path/to/base>`
- exemple

`jdbc:derby://localhost:1527//home/foo/db/contacts`

# Derby - URL

- Crédation de la base de données s'il elle n'existe pas
  - `jdbc:derby://localhost:1527//home/foo/db/contacts;create=true`

