**Commandes en Java - JPA**

JPA = Java Persistance API = lien entre Java et les Data Base

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Symbole | Signification | Code | Utilisation | | |
| De base |  |  |  | | |
| persistence.xml | fichier à placer dans le dossier src/main/resources/META-INF/ du projet | *voir fichier* peristence.xml |  | |  |
| POM.txt | dépendances indispensables à placer dans le fichier POM.xml du projet Maven | *voir fichier* dans le Pom.txt |  | |  |
|  |  |  |  | |  |
| Annotations |  |  | javax.persistence | | package issu de l'importation de la dépendance par Maven |
|  | **sur les classes** |  |  | |  |
| @Entity | indique un objet de type been comme un objet à persister (cad à enregistrer dans une DB) = POJO | @Entity  @Table(name="MACLASSE")  public class MaClasse {  @Id  @GeneratedValue(strategy=GenerationType.IDENTITY)  private int id;  @Column(name="MONPARAMETREA", lenght = 30, nullable = false, unique = true)  private String monParametreA;  @Column(name="MONPARAMETREB")  private String monParametreB;  @Column(name="MON\_PARAMETRE\_C")  private String monParametreC;  @Transient  private String monParametreAIgnorer;  public MaClasse() {  }  //getters and setters  } |  | |  |
| @Table | précise le nom de la table à utiliser dans la DB |  | |  |
|  |  |  | |  |
|  | **sur les variables** |  | |  |
| @Id | indique l'integer qui servira de clé primaire |  | |  |
| @GeneratedValue | précise le type de générateur que doit utiliser la DB pour générer l'id, cad la clé primaire |  | | Rq:  IDENTITY = auto-incrémenté  TABLE = utilisaton d'une table dédiée aux clés (ajout de @TableGenerator)  SEQUENCE = mécanisme + complexe (ajout de @SequenceGenerator) |
| @Column | indique le nom de la colonne à utiliser dans la table pour stocker la valeur de l'élément | options de **@Column( )**   * **name** nom de la colonne * **unique** permet de créer une contrainte d’unicité uniquement sur ce champ * **nullable** champ not null ou non (défaut : true) * **insertable** inclu dans les ordres SQL INSERT (défaut : true) * **updatable** inclu dans les ordres SQL UPDATE (défaut : true) * **length** ajuste la longueur d’un champ string (par défaut : 255) * **precision**  ajuste le nombre de chiffres d’un champ décimal stockées (à gauche et à droite du séparateur décimal) * **scale** ajuste le nombre de chiffre après le séparateur décimal (par défaut : 0) | | |
| @Transient | indique de ne pas tenir compte du paramètre lors du mapping |  | |  |
| @Temporal | indique le type SQL à utiliser pour une colonne de type date et/ou heure |  | TemporalType.DATE  TemporalType.TIME  TemporalType.TIMESTAMP | | ATTENTION : ne fonctionne qu'avec les objets **java.util.Date** |
| java.time | pour les paramètres de type java.time (Instant, LocalDate, ...) | @Column(nullable = false, columnDefinition = "TIMESTAMP WITH TIME ZONE")  private Instant creation; |  | | Rq: stocker la date avec "TIMESTAMP WITH TIME ZONE" permettra de la réutiliser facilement en l'adaptant selon la time zone de l'utilisateur |
| @Enumerated | indique un paramètre de type enumération | @Column(name="power\_source", nullable = false)  @Enumerated(EnumType.STRING)  public PowerSource powerSource; |  | |  |
|  |  |  |  | |  |
|  | **sur les variables liées à la jointure avec d'autres tables** |  |  | |  |
| @OneToOne | un élément est lié qu'à un seul autre élément de manière unique  création d'une colone contenant l'id de l'élément lié | @Entity  @Table(name="FACTURE")  public class Facture {  @Id  private int id;  @OneToOne  private Reservation reservation:  } |  | |  |
| @OneToMany  @ManyToOne | un élément pouvant être lié à plusieurs éléments  plusieurs éléments de ce type peuvent être liés à un seul élément  créé une jointure entre les tables liées par l'id de l'élément OneToMany  (cad dans l'exemple, la table CHAMBRE contiendra une colonne ID\_HOTEL) | @Entity  @Table(name="CHAMBRE")  public class Chambre {  @Id  private int id;  @Column(name="NUMERO")  private int numero;  **@ManyToOne**  **@JoinColumn(name="ID\_HOTEL")**  private Hotel **hotel**; | @Entity  @Table(name="HOTEL")  public class Hotel {  @Id  private int id;  @Column(name="NOM")  private String nom;  **@OneToMany(mappedBy="hotel")**  private Set<Chambre> chambres;  } | |  |
| @ManyToMany | un élément peut être lié à plusieurs autres, et inversement  ceci nécessite la création d'une table intermédiaire qui contiendra les deux id des éléments associés  (cad dans l'exemple, création d'une table CHAMBRE\_RESERVATION avec les colonnes : ID\_CHAMBRE et ID\_RESERVATION) | **@ManyToMany**  @JoinTable(name="CHAMBRE\_RESERVATION",  joinColumns=@JoinColumn(name="ID\_CHAMBRE", referencedColumnName="ID"),  inverseJoinColumns=@JoinColumn(name="ID\_RES", referencedColumnName="ID") )  private Set<Reservation> **reservations**;  } | @Entity  @Table(name="RESERVATION")  public class Reservation {  @Id  private int id;  **@ManyToMany(mappedBy="reservations**"**)**  private Set<Chambre> chambres;  } | |  |
|  |  |  |  | |  |
|  |  |  |  | |  |
| Inclusion | une classe dans une autre classe, mais une seule table ! |  |  | |  |
| @Embeddable  @Embedded | inclus automatiquement les variables d'une classe à un élément comme s'il s'agissait de ses propres variables  (dans l'exemple, la table PERSONNE contiendra les colonnes NOM, PRENOM, NUMERO, RUE, CODE\_POSTAL et VILLE) | @Embeddable  public class Adresse {  @Column(name="NUMERO")  private int numero;  @Column(name="RUE")  private String rue;  @Column(name="CODE\_POSTAL")  private int codePostal;  @Column(name="VILLE")  private String ville;  } | @Entity  @Table(name="PERSONNE")  public class Personne {  @Id  private int id;  @Column(name="NOM")  private String nom;  @Column(name="PRENOM")  private String prenom;  @Embedded  private Adresse adresse;  } | | ATTENTION : la classe Embeddalble   * ne doit pas contenir d'ID (l'id sera celui de la classe dans laquelle elle est incluse) * doit présenter des noms de colonnes différents de ceux de la classe qui l'inclue |
| @Embeddable  @EmbeddedId | inclusion d'une clé composite  => nécessite de passer par une classe spécifique qui gère ces clés | @Entity  @Table(name="PROJECT)  public class Project {  @EmbeddedId  private CleComposite id;  …  } | @Embeddable  class CleComposite {  private int departmentId;  private long projectId;  …  } | |  |
|  |  |  |  | |  |
|  |  |  |  | |  |
| Héritage |  |  |  | |  |
| SINGLE\_TABLE | stratégie une seule table pour toute la hiérarchie  + très simple  + pas de jointure  - bcp de colonnes vides  - NOTNULL inutilisable sur les colonnes | @Entity  **@Inheritance(**strategy = **InheritanceType.SINGLE\_TABLE)**  @DiscriminatorColumn(name="TYPE")  public class MaClasseMereOuAbstraite {  @Id  protected int id;  …  } | @Entity  @DiscriminatorValue("F1")  public class MaClasseFille extends MaClasseMereOuAbstraite {  @Column(name="MA\_VARIABLE)  private String maVariable;  } | | Rq: nécéssite la création d'une colonne qui discriminera la ligne en fonction de son type de classe  ATTENTION : les classes dérivées ne contiennent pas d'id, puisqu'il est hérité de la classe mère ! |
| TALBE\_PER\_CLASS | stratégie une table par classe concrète (classe mère et classes filles)  = ignore la notion d'héritage  + pas de colonne vide  + pas de jointure  - pas d'unicité des clés primaires sur chaque table  - duplication des colonnes | @MappedSuperclass  **@Inheritance(**strategy = **InheritanceType.TABLE\_PER\_CLASS)**  public class MaClasseMereOuAbstraite {  @Id  protected int id;  …  } | @Entity  public class MaClasseFille extends MaClasseMereOuAbstraite {  @Column(name="MA\_VARIABLE)  private String maVariable;  } | | ATTENTION : les classes dérivées ne contiennent pas d'id, puisqu'il est hérité de la classe mère ! |
| JOINED | stratégie une table par sous-classe  (une table pour les variables de la classe mère, une table pour les variables supplémentaires de la classe fille 1, une table pour les variables supplémentaires de la classe fille 2, …)  + pas de colonne vide  + NOTNULL utilisable partout  + pas de duplication des colonnes  - => jointure pour chaque requête par le SGBD (pb si bcp de données !) | @Entity  **@Inheritance(**strategy = **InheritanceType.JOINED)**  public class MaClasseMereOuAbstraite {  @Id  protected int id;  …  } | @Entity  public class MaClasseFille extends MaClasseMereOuAbstraite {  @Column(name="MA\_VARIABLE)  private String maVariable;  } | | ATTENTION : les classes dérivées ne contiennent pas d'id, puisqu'il est hérité de la classe mère ! |
|  |  |  |  | |  |
|  |  |  |  | |  |
| Evènements |  |  |  | |  |
| @PrePersist  @PreUpdate  @PreRemove | permet de définir des instructions à réaliser **avant** l'enregistrement, la MàJ ou la suppression de l'élément dans la DB | *Avec écouteur interne :*  @Entity  @Table(name="MON\_POJO")  public class MonPOJO {  @Id  private int id;  **@PrePersist**  private void **onPrePersist() {**  //vérifications à effectuer avant la persistance du POJO  **}**  …  } | *Avec un écouteur externe :*  @Entity  @EntityListener({AuditInterceptor.class, LogInterceptor.class})  public class MonPOJO {  @Id  private int id;  }  public class **AutitInterceptor {**  **@PrePersist**  private void **onPrePersist(Object o) {**  //Vérification avant la persistance du POJO  **}**  …  **}** | |  |
| @PostPersist  @PostUpdate  @PostRemove  @PostLoad | permet de définir des instructions à réaliser **après** l'enregistrement, la MàJ ou la suppression de l'élément dans la DB  ou après le chargement depuis la DB (cad la création du POJO à partir des infos issues de la DB) |  |
|  |  |  |  | |  |
|  |  |  |  | |  |
|  |  |  |  | |  |
|  |  |  |  | |  |
| Persistence |  |  |  | |  |
| EntityManagerFactory | objet permettant la création du lien avec la DB | ***Requêtes en simple lecture : Requêtes avec écriture :***  EntityManagerFactory emFactory = Persistence.createEntityManagerFactory("maPersistenceUnit-jpa");  EntityManager em = emFactory.createEntityManager(); | | | Rq: le nom maPersistenceUnit-jpa est celui indiqué dans la section **<persistence-unit name=" maPersistenceUnit-jpa" transaction-type… >** du fichier persistence.xml (src/main/resources/META-INF/) |
| EntityManager | objet permettant la gestion des requêtes, des éléments à sauvegarder, … | MonPojo pojo = em.find(MonPojo.class, 10);  TypedQuery<MonPojo> query = em.createQuery("SELECT p FROM MonPojo p WHERE p.monParametreA='maValeur'");  List<MonPojo> pojos = query.getResultList();  *//ou*  MonPojo pojo = query.getSingleResult();  TypedQuery<MonPojo> query = em.createQuery("SELECT p FROM MonPojo p WHERE p.monParametreA=**:paramTemp**");  query**.setParameter**(**"paramTemp"**, "maValeur");  List<MonPojo> pojos = query.getResultList();  em.close();  emFactory.close(); | | EntityTransaction transaction = em.getTransaction();  transaction.begin();  em.persist(monNewPojo);  em.merge(monPojoModifie);  em.remove(monPojoModifie);  transaction.commit();  *// ou*  trnasaction.rollback();  em.close();  emFactory.close(); |  |
| .find( , ) | retourne l'objet de type indiqué en paramètre 1 et d'identifiant indiqué en parametre 2  retourne null si non trouvé |  |
| .getReference( , ) | idem que précédente, mais retourne EntityNotFoundException si non trouvé |  |
| .createQuery( )  TypedQuery<T> | crée un objet permettant de faire une requête et de retourner le/la liste de résultat(s) |  |
| .persist( ) | enregistre les valeur d'un nouvel objet dans la DB (crée une nouvelle ligne) |  |
| .merge( ) | enregistre les modifications effectuées sur les valeurs d'un élément issu de la DB |  |
| .remove( ) | supprime la ligne contenant les valeurs de l'élément issu de la DB |  |
|  |  |  |
| EntityTransaction | objet permettant (en association avec l'EntityManager) de gérer l'écriture (ajout, modification, suppression) dans la DB |  |
| .begin() | initialise les modifications |  |
| .commit() | lorsque toutes les modifications sont effectuées sur les objets, elle envoie ces modifications vers la DB pour qu'elles soient réellement enregistrées |  |
| .rollback() | si ces modifications ne doivent finalement pas être prise en compte dans la DB, .rollback() annule la transaction avant modification de la DB |  |
|  |  |  | |  |  |
|  |  |  | |  |  |
| JPQL | langage proche du SQL et permettant de faire le lien entre Java (JPA) et la DB |  | |  |  |
|  | exemples de requètes JPQL à placer dans em.createQuery( )  sélectionner plusieurs colonnes  Fonctions  Jointure  INSERT, UPDATE et DELETE s'exécutent avec **query.executeUpdate()**  Cependant, il est préférable d'utiliser un POJO complet avec em.perist( ), em.merge( ), ou em.remove( ) | "SELECT p FROM Person p WHERE p.name = :name"  query.setParameter("name", "Doe");  "SELECT p FROM Person p WHERE p.age > 50"  "SELECT p.firstName FROM Person p"  List<String> firstNames = query.getResultList();  "**SELECT p.firstName, p.lastName** FROM Person p"  List<Object[]> firstAndLastNames = query.getResultList();  "SELECT MAX(p.age) FROM Person p"  "SELECT l FROM Livre l WHERE Size(l.emprunts)=0"  "SELECT l FROM Livre l WHERE l.emprunts IS EMPTY"  "SELECT p FROM Person p JOIN p.adress a WHERE a.city = 'Lyon'"  List<Person> persons = query.getResultList();  "UPDATE Livre l SET l.titre = :newTitre WHERE l.titre = :oldTitre"  query.setParameter("newTitre", "Germinal");  query.setParameter("oldTitre", "Le nouveau Germinal');  query.executeUpdate(); | |  |  |
|  |  |  | |  |  |
|  |  |  | |  |  |
|  |  |  | |  |  |