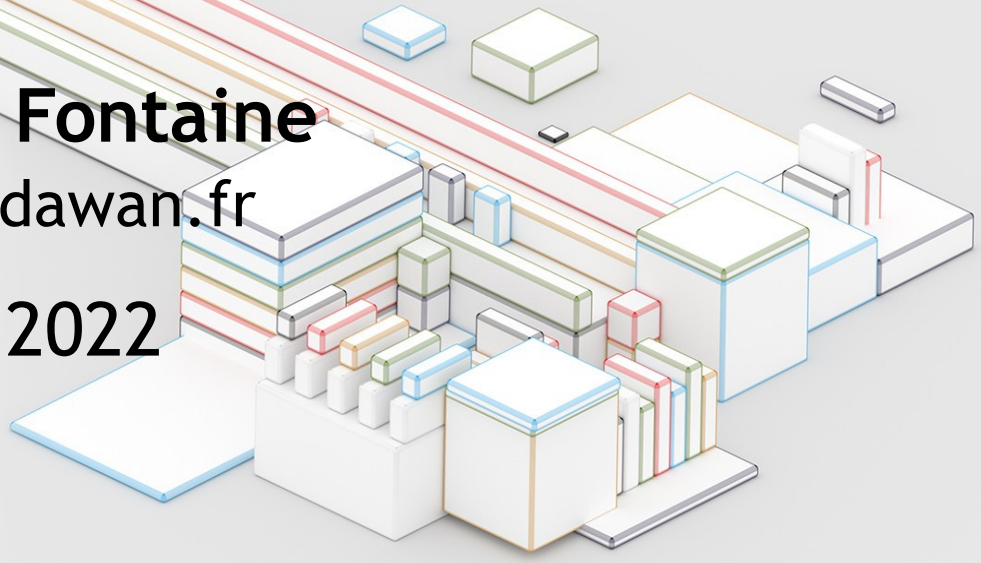
JPA / Hibernate

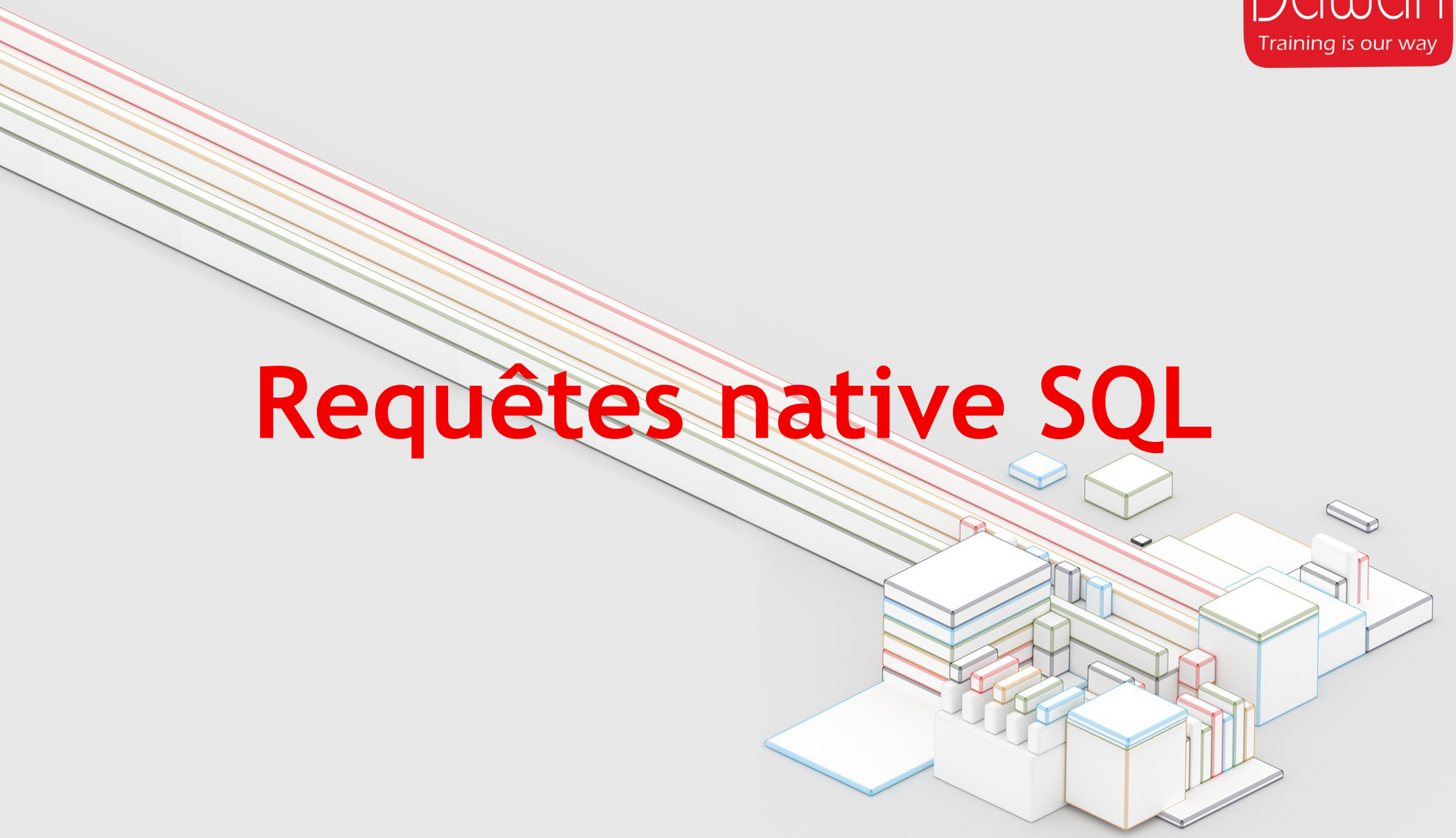
Christophe Fontaine
cfontaine@dawan.fr

21/02/2022



Dawan
Training is our way

Requêtes native SQL



Requêtes natives

- Le JPQL ne permet pas de faire tout ce que l'on peut faire en SQL, pour cela il est possible de définir également des requêtes en SQL natif en JPA
- On peut utiliser la méthode **createNativeQuery**() d'EntityManager
- On peut également définir des requêtes nommées natives avec les annotations :
  @NamedNativeQueries et **@NamedNativeQuery**

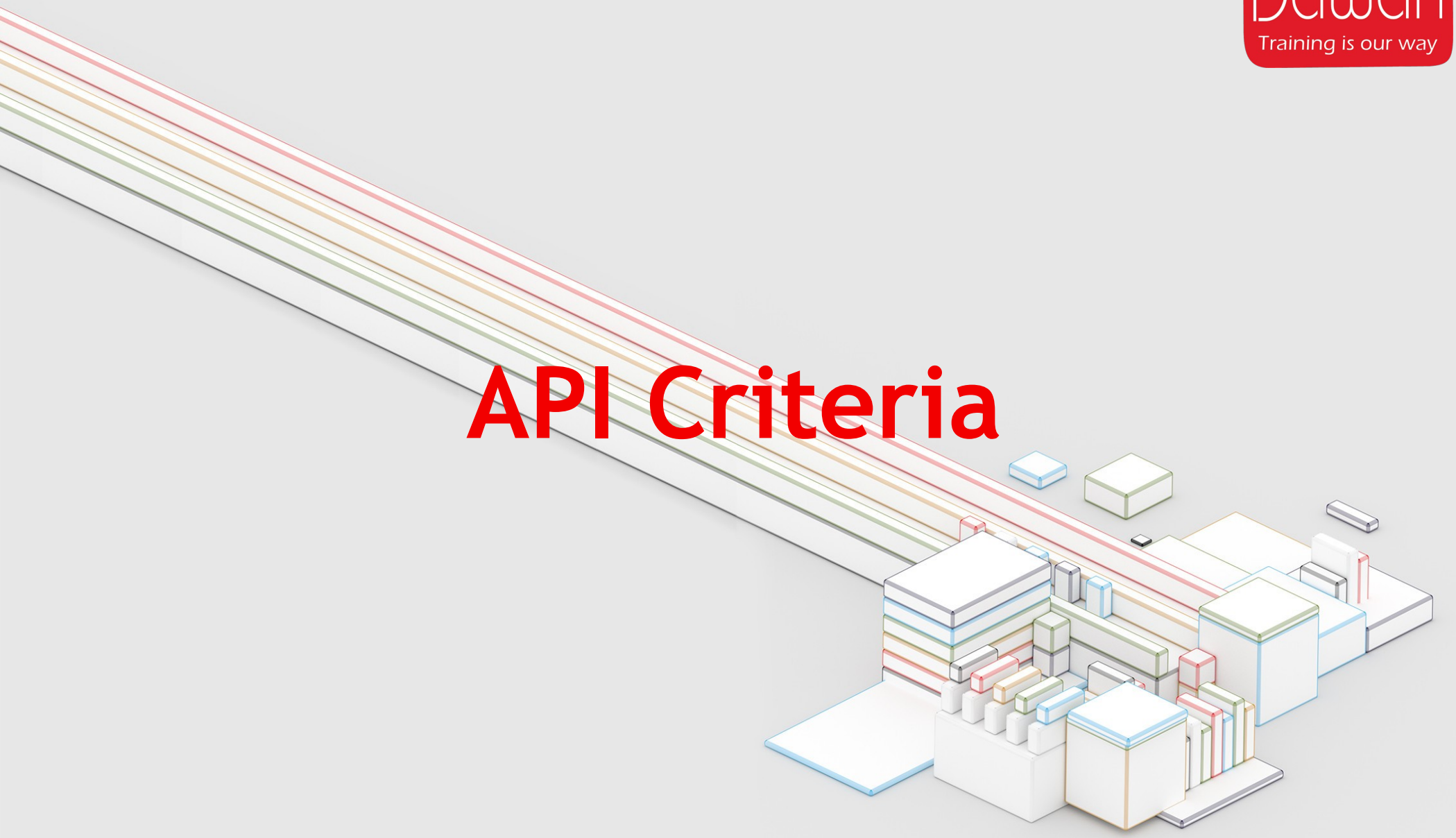
Inconvénient : attacher le code JPA à une base de données particulière

Requêtes natives

```
@Entity
@Table(name = "individu")
public class Individu {
  @Id
  @Column(name = "individuId")
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  @Column(length = 30, nullable = false)
  private String nom;
  @Column(length = 30, nullable = false)
  private String prenom;
  //  ...
}
int ageMax = 25;
List<Individu> individus = null;
individus =  ,entityManager
              .createNativeQuery( "select * from
               individu where age <= ?", Individu.class)
              .setParameter(1, ageMax)
              .getResultList();
```

Dawan
Training is our way

API Criteria



API Criteria

- API introduite par JPA 2.0
- Tout ce qui peut être fait avec JPQL peut l'être avec l'API criteria
- Les requêtes JPQL sont des chaîne de caractères qui peuvent contenir des erreurs (par exemple le nom d'une classe qui n'existe pas)
- L'avantage de l'API criteria est que les requêtes peuvent être vérifiées à la compilation
- L'inconvénient est que le code est un peu plus complexe à écrire et moins lisible

MetaModel

- **MetaModel** → Une liste de classe représentant la structure des données de l'unité de persistance
- Chaque classe décrit une entité avec ses propriétés et ses relations
- 2 types de MetaModel :
  - **dynamique** : généré à l'exécution du code
  - **statique** : généré dans le code source
    Les classes utilisées pour définir le MetaModel :

    nomDeEntite_

    fr.dawan.jpa.Personne → fr.dawan.jpa.Personne_

    Personne.nom → Persone_.nom

CriteriaBuilder

- Pour construire une requête, on utilise un objet **CriteriaBuilder**
- On l'obtient avec la méthode **getCriteriaBuilder**() de l'EntityManager
- On peut réaliser 3 opérations qui correspondent à 3 classes que l'on obtient de **CriteriaBuilder**
  - select→ **CriteriaQuery**
  - update→**CriteriaUpdate**
  - delete→ **CriteriaDelete**

```
CriteriaBuilder cb=em.getCriteriaBuilder();
CriteriaDelete<Livre> cd=cb.createCriteriaDelete(Livre.class);
```

CriteriaQuery

Pour construire une requête on doit, dans l'ordre :

1. **définir le type de retour (obligatoire)**
   - pour une entité → `createQuery(`*`classeEntité`*`)`
   - pour une sélection particulière non typé
     ↳ `createQuery(`*`classeEntité`*`)`
   - pour une sélection particulière typé tupple
     ↳ `createTuppleQuery()`
2. **paramétrage du périmètre (obligatoire)**
   ↳ méthode `from()`
3. **paramétrage de restriction** → méthode `where()`

CriteriaQuery

4. **paramétrage du regroupement**
   ↳ méthode `having()` et `groupBy`
5. **paramétrage du retour**
   - pour un objet → méthode `select()`
   - pour une liste d'objet → méthode `multiselect()`
   - Il doit être prise en compte par le périmètre :

```
CriteriaQuery<Livre> cq=cb.createQuery(Livre.class);
Root<Livre> l=cq.from(Livre.class);
```

   - pour une entité Livre → `cq.select(p);`
   - pour sélectionner l' attribut titre de Livre
     ↳ `cq.select(p.get(Livre_.titre)`
   - pour sélectionner les attributs titre de Livre

↳`cq.multiselect(p.get(Livre_.titre),p.get(Livre_.titre))`

CriteriaQuery

6. **Paramétrage de trie** → méthode `orderBy()`

7. **Préparation de la requête** → méthode `createQuery` de l'EntityManger

8. **Exécution de la requête** → identique à JPQL

From

- Les entités à utiliser sont des objets **Root<entité>**
- On fait référence à une entité dans requête avec la méthode **from()**

```
Root<Livre> p=cq.from(Livre.class);
```

- L'objet **Root** va être utilisé pour continuer à compléter la requête
- Pour faire référence à plusieurs entités, on appelle plusieurs fois **from()**

```
Root<Livre> p=cq.from(Livre.class);
Root<Auteur> pa=cq.from(Auteur.class);
```

Jointure

- On utilise la méthode **`join()`** de l'entité **`Root`** à l'origine de le jointure avec en paramètre l'attribut concerné par la jointure

```
Join<Livre,Auteur> j= p.join(Auteur_.livre);
```

- Pour une relation :
  - OnetoOne et ManytoOne → `Join`
  - OnetoMany et ManytoMany → `CollectionJoin`, `ListJoin`, `SetJoin`, `MapJoin`
- On peut définir le type de jointure avec : `JoinType.INNER` (par défaut), `JoinType.LEFT`, `JoinType.RIGHT` (pas toujours supporté par toutes les implémentations de JPA)

Where

- Les conditions de restriction doivent être fait en une fois, sinon seule la dernière sera prise en compte
- On peut appeler la méthode `where()` avec
  - **Expression<Boolean>** → test sur un attribut boolean

```
Root<Telephone> t=cq.from(Telephone.class);
Expression<Boolean>attTelephone=
                              t.get(Telephone_.portable);
cq.where(attTelephone);
```

  - **Predicat** → test

    On obtient les prédicats de CriteriaBuilder

```
Predicate prTitre=cb.like(p.get(Livre_.titre,D%)
```

Where

- **Plusieurs Predicat** → plusieurs test séparer par la condition AND
- **Predicate[ ]** → tableau de predicate
- **Sans paramètre** → efface les conditions déjà existante

GroupBy, Having, OrderBY

- **Group By**→ `groupBy()`
  ↳ utilise des Expression<>

```
Expression<String> g1=p.get(Livre_.titre);
Expression<String> g2=p.get(Livre_.anne);
cp.groupBy(g1) ; ou cp.groupBy(g1,g2);
cp.groupBy(g2);
```

- **Having** → `having()` idem `where`
- L'objet **`Order`** sert à définir L'ordre de tri. On l'obtient avec les méthodes **`asc()`** ou **`desc()`** de CriteriaBuilder

Expression, Path, Paramètre

- **Expression** → peut-être une valeur sur :
  - Une donné littéral (String, Boolean)
  - Une variable (ROOT,PATH)
  - Le résultat d'une fonction
- **Path**→ une expression qui représente un attribut

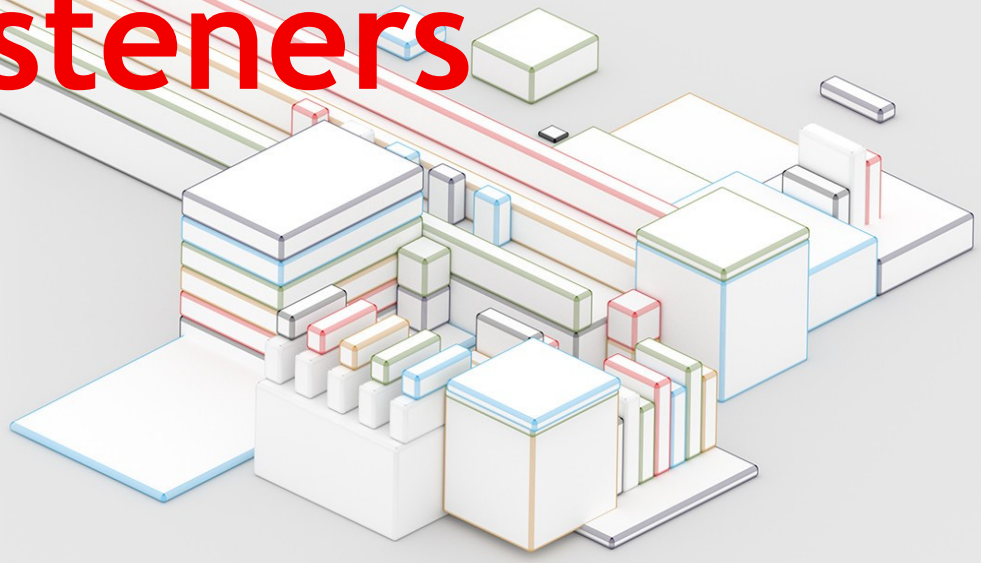
```
Path<String> attTitre= p.get(Livre_.titre);
```

- **Paramètre**→On peut placer des paramètres dans la requête. Il faut utiliser l'objet **`ParameterExpression<>`**

```
ParameterExpression<String> param=cb.parameter(String.class);
eq.where(cb.equal(p.get(Personne_nom,param));
cq.select(p);
em.creatQuery(cq).setParamter(param, "    ").getResultList();
```

Dawan
Training is our way

Intercepteurs, event-listeners



Intercepteurs

- Hibernate fournit des intercepteurs (callbacks) au niveau de l'entité permettant d'effectuer des traitements
- Interfaces à implémenter :
  - **Lifecycle** : traitement sur la sauvegarde, mise à jour, suppression ou le chargement d'un objet (propre à l'entité)
    Peut être utilisée pour gérer la cascade au lieu du mapping

    On peut utiliser des annotations JPA au lieu de cette interface :
    `@PrePersist`, **`@PostPersist`**, **`@PostUpdate`**, ...

Intercepteurs

- **Interceptor** : callback de la session à l'application pour introspecter les propriétés d'une entité
  A déclarer dans les propriétés d'Hibernate :

```
<property name="hibernate.ejb.interceptor"
value="nompackage.NomInterceptor" />
```

  ou utiliser les mêmes annotations JPA + @EntityListeners sur l'entité

Intercepteurs
JPA Callbacks

Définir des méthodes dans l'entité concernée :

- **`@PrePersist`** → avant qu'une nouvelle entité soit persisté (ajouté à l'EntityManager)
- **`@PostPersist`** → après le stockage d'une nouvelle entité dans la base de donnée (pendant un commit ou un flush)
- **`@PostLoad`** → après qu'une entité ait été extraite de la base de données
- **`@PreUpdate`** → lorsqu'une entité est identifiée comme modifiée par l'EntityManager

Intercepteurs
JPA Callbacks

- **`@PostUpdate`** → après la mise à jour d'une entité dans la base de données (pendant un commit ou un flush)
- **`@PreRemove`** → lorsqu'une entité est marquée pour être supprimée dans l'EntityManager
- **`@PostRemove`** → après la suppression d'une entité de la base de données (pendant un commit ou un flush)

Intercepteurs
Interceptor

- On peut implémenter l'interface **`Interceptor`** ou mieux, hériter de **`EmptyInterceptor`**

```
public class AuditInterceptor extends EmptyInterceptor {
    private EntityManager em;
    public AuditInterceptor() {
        System.out.println("AuditInterceptor constructed");
        EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("PU_2");
        em = emf.createEntityManager();
    }

    @Override
    public boolean onSave(Object entity, Serializable id,
          Object[] state, String[] propertyNames, Type[] types) {
        System.out.println("onSave called");
        em.getTransaction().begin();
        AuditLog a = new AuditLog();
        a.setEntry("object with ID: " + id + " saved");

       //…
```

Intercepteurs
Interceptor

```
        //…
        em.persist(a);
        em.getTransaction().commit();
        return false;
    }

    @Override
    public void afterTransactionCompletion(Transaction tx) {
        System.out.println("afterTransactionCompletion called");
        em.getTransaction().begin();
        AuditLog a = new AuditLog();
        a.setEntry("transaction: " + tx + " completed");
        em.persist(a);
        em.getTransaction().commit();
    }
}
```

Listeners

- Méthodes de callbacks à appliquer sur une entité :

```
public class MyListener {
    @PrePersist void onPrePersist(Object o) {}
    @PostPersist void onPostPersist(Object o) {}
    @PostLoad void onPostLoad(Object o) {}
    @PreUpdate void onPreUpdate(Object o) {}
    @PostUpdate void onPostUpdate(Object o) {}
    @PreRemove void onPreRemove(Object o) {}
    @PostRemove void onPostRemove(Object o) {}
}
```

Listeners

- Application sur l'entité d'un ou plusieurs listeners on peut gérer également l'héritage :

```
@Entity @EntityListeners(MyListener.class)
public class MyEntityWithListener {
}

@Entity @EntityListeners({MyListener1.class,
MyListener2.class})
public class MyEntityWithTwoListeners {
}
```

- On peut exclure des listeners hérités d'une super classe :

```
@Entity @ExcludeSuperclassListeners
public class EntityWithNoListener extends
EntityWithListener {
}
```

Listeners par défaut

- On peut définir des listeners par défaut (pas d'annotation, fichier META-INF/orm.xml) :

```
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
 http://java.sun.com/xml/ns/persistence/orm_1_0.xsd" version="1.0">
  <persistence-unit-metadata>
    <persistence-unit-defaults>
      <entity-listeners>
        <entity-listener class="fr.dawan.JpaHibernateTraining.
                                tools.MyDefaultListener" />
      </entity-listeners>
    </persistence-unit-defaults>
  </persistence-unit-metadata>
</entity-mappings>
```

Listeners par défaut

- On peut exclure les default listeners pour une entité :

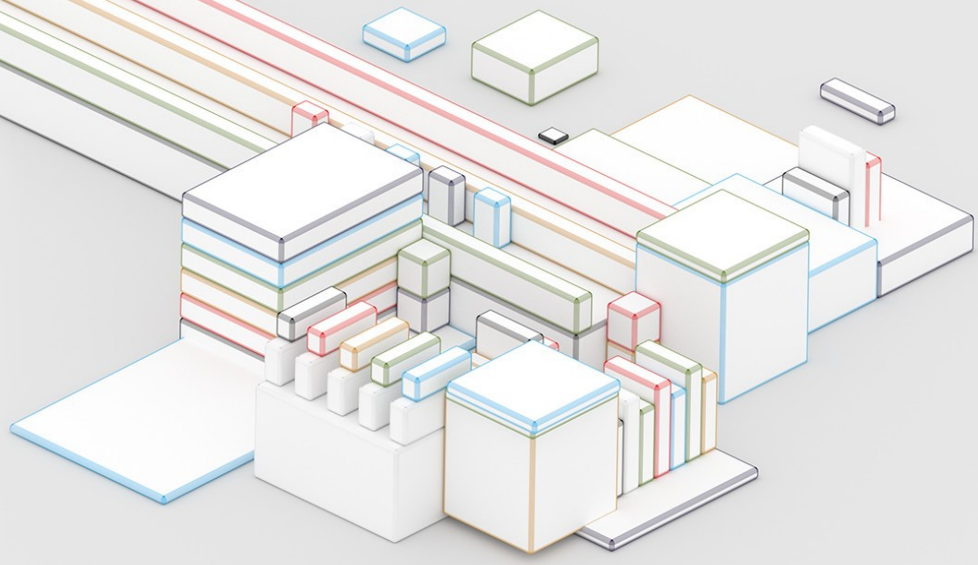
```
@Entity @ExcludeDefaultListeners
public class NoDefaultListenersForThisEntity {

}

@Entity
public class NoDefaultListenersForThisEntityEither extends
                        NoDefaultListenersForThisEntity {

}
```

Gestion des caches

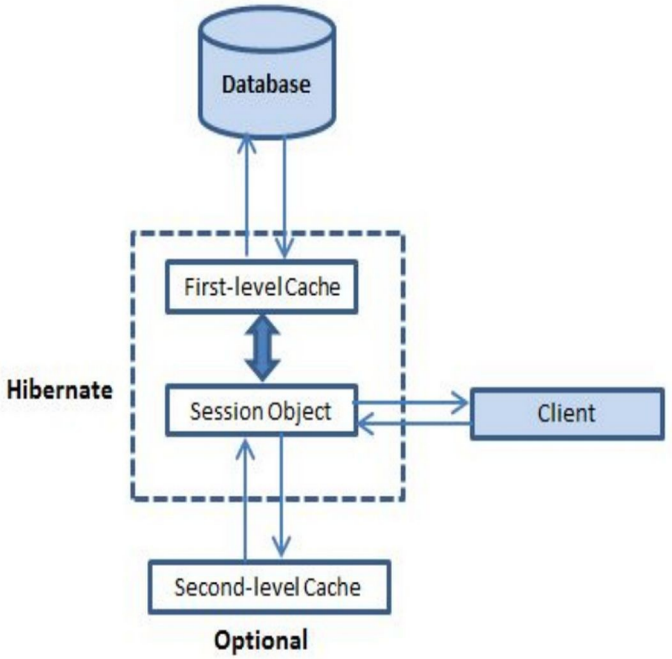


Cache de niveau 1

hibernate possède 2 niveaux de cache

Cache de niveau 1 :

- Chaque fois que vous passez un objet à **`persist()`**, **`merge()`** ou que vous récupérez un objet avec **`find()`**, cet objet est ajouté au cache interne de la session (EntityManager)
- Quand **`flush()`** est ensuite appelée, l'état de cet objet va être synchronisé avec la base de données



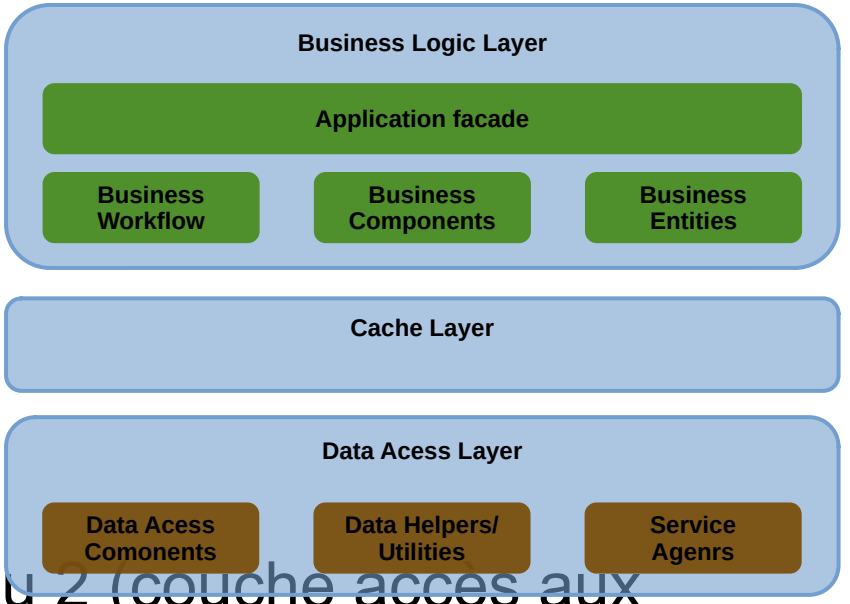
Si vous ne voulez pas que cette synchronisation se produit ou si vous traitez un grand nombre d'objets et la nécessité de gérer efficacement la mémoire :

- La méthode **`detach()`** peut être utilisée pour supprimer l'objet et ses collections dépendantes du cache de premier niveau
- La méthode **`clear()`** permet de vider complètement le cache de la session

Cache de niveau 2 Introduction

- La mise en cache des entités est une technique très importante pour améliorer les performances de l'application
  Généralement, on introduit une couche de mise en cache dans une architecture multi-couches **avant la couche d'accès aux données**

  Parfois, on utilise les composants web côté présentation pour mettre en cache les entités :
  - Session
  - Application (servletContext)



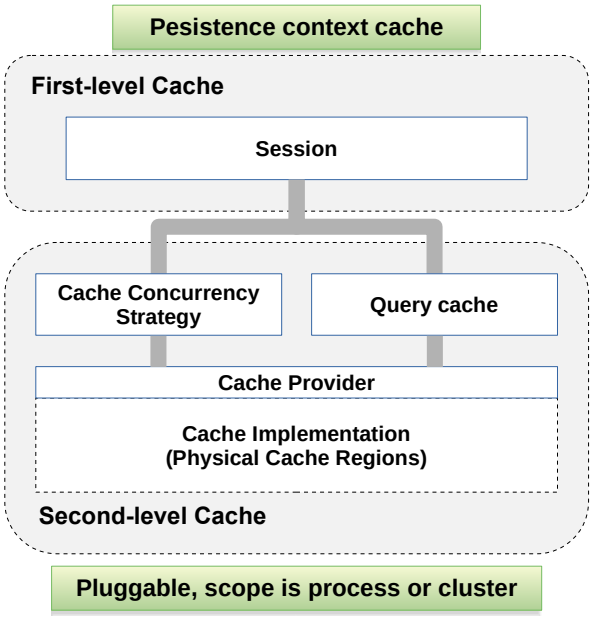
  Hibernate fournit un cache de niveau 2 (couche accès aux données) qui permet de s'abstraire de l'utilisation des composants
  de la couche présentation ou métier

Cache de niveau 2 Configuration

```
<property name="cache.use_second_level_cache">true</property>
<property name="cache.provider_class">...</property>
<property name="cache.use_query_cache">true</property>
<property name="prepare_sql">true</property>
```

- De multiples cache providers sont disponible

https://docs.jboss.org/hibernate/stable/core.old/reference/fr/html/performance-cache.html



Cache de niveau 2
Caching des entités

- Pour spécifier la mise en cache d'une entité ou d'une collection, on ajoute l'annotation **`@Cache`** dans le mapping de la classe ou de la collection ou une conf. Xml :
- L'attribut **`usage`** spécifie la stratégie de gestion de la concurrence
  - **`read only`** : cache en lecture seule, pas de modification d'instances
  - persistantes (manière la plus simple et la plus performante)
  - **`read/write`** : si l'application doit mettre a jour des données On doit s'assurer que la transaction est terminée et que la session est fermee (strict)
  - **`nonstrict read/write`** : si l'application doit occasionnellement mettre a jour des données (multiples transactions simultanées)

Cache de niveau 2
Caching de requêtes

- Les résultats d'une requête peuvent être mis en cache

  Utile uniquement pour les requêtes exécutées fréquemment avec les mêmes paramètres

- Configuration : `cache.use_query_cache = true`
- Utilisation : `Query.setCacheable(true)`

```
List<Book> booksList2 = session.createQuery("FROM Book b")
.setCacheable(true).setCacheRegion("ShortTermCacheRegion")
.list();
// …
```

- Forcer le rafraîchissement :

  `sessionFactory.evictQueries(regionName)`