Tout ce qui est sur fond rose est à remplir par le collaborateur

Collaborateur		**Date d'évaluation**	

Consignes :
Lisez bien toutes les questions avant de commencer.
Adoptez une stratégie intelligente pour optimiser votre score final.

1. Que se passe-t-il lorsqu'on appelle la méthode flush d'une session Hibernate ?

/ 0,5

Hibernate synchronise l'état de sa session avec la base de données au moment du flush.

2. Quels soucis peut-on avoir sur une Entity si hashcode et Equals ne remplissent pas leur contrat ?

/ 0,5

Le contrat entre hashcode et equals est : "**deux entités** qui sont **égales** ont le **même hashcode** mais **deux entités** avec le **même hashcode ne** sont **pas forcément égales**".

Si hashcode et égale ne vérifient pas ce contrat, on pourrait avoir deux entités qui sont égales mais qui n'ont pas le même hashcode, ce qui entraînerait qu'on **ne trouve pas l'entité dans une hashmap**. En effet, le hashcode étant mauvais, on arriverait dans le **mauvais bucket** et l'entité ne serait pas dedans.

3. Etant donné le mapping suivant :

```
@Entity
public class Pays{
// ...
  @OneToMany(mappedBy="pays")
  private List<Ville> villes;
// ...
}

@Entity
public class Ville{
// ...
```

```
  @ManyToOne
  private Pays pays;
// ...
}
```

Le code suivant avec un Pays (france) et une Ville (paris) non nulls et correctement initiés :

```
entityManager.persist(france);
entityManager.persist(paris);
france.getVilles().add(paris);
transaction.commit();
```

Expliquer l'output SQL suivant et ce qu'il manque et pourquoi ?

```
insert into Pays (name, id) values ('France', 1)
insert into Ville (name, pays_id, population, id) values ('Paris', NULL, 2000000, 2)
```

/ 0,5

Mapped-by dans l'annotation @OneToMany délègue la gestion de la relation au champ pays de la classe Ville. Ainsi, les opérations faites sur le champ villes de la classe Pays sont sans effet d'un point de vue JPA/Hibernate.

4. À quoi faut-il faire attention lorsqu'on gère une relation bidirectionnelle et quelles sont les bonnes pratiques ?

/ 0,5

Il faut faire attention à plusieurs choses :
1 - gérer les 2 côtés de la relation c'est à dire valoriser le champ des 2 côtés
2 - bien utiliser le côté "fort" de la relation c'est à dire celui où il n'y a pas le mapped-by
3 - éviter les désynchronisation entre ce qu'il y a côté persistance et ce qu'il y a en mémoire côté Java

pour cela on utilisera des méthodes de type addXXX() et removeYYY() avec

Class Pays (oneToMany)

```
addCity(City city){
    //Suppression des anciens liens
    Pays oldPays = city.getPays();
    if(city.getPays()!=null){
        oldPays.removeCity(city);

     }
    city.setPays(this);
    cities.add(city);
}
```

```
removeCity(City city){
    city.setPays(null);
    cities.remove(city);
}
```

5. Pourquoi ne doit-on JAMAIS faire cela :

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "secteurSequence")
@SequenceGenerator(name = "secteurSequence", sequenceName = "SEQ_SECTEUR", allocationSize = 1)
private Long id;
```

?

/ 0,5

Positionner allocationSize à 1 revient à demander à chaque entityManager.persist() d'aller chercher la nouvelle valeur pour la séquence. Ainsi, on multiplie les roundtrips vers la DB pour faire SELECT SEQ.nextVal from DUAL; sur Oracle par exemple.
La valeur par défaut est de 50 autant la garder voire mieux utiliser un optimiseur de séquence de type pooled-lo.

6. Pourquoi est-il conseillé de mettre @Transactional(readOnly=true) quand on fait de la lecture seule ?

/ 0,5

Si tu mets la transaction en readOnly, il n'y a plus de Flush et du coup cela désactive le mécanisme de "dirty checking".

Le dirty checking est une sorte d'observateur sur les propriétés des entités chargées dans l'EntityManager.
On considère le nombre de dirty checks N = $\sum_{k=1}^{n} p(k)$ avec n nombre d'entités et p(k) propriétés d'une entité donnée.
=> Cela consomme énormément de CPU et de mémoire surtout si les entités sont nombreuses et volumineuses.

Avec l'API de JPA on peut le faire manuellement :

```
List<Post> posts = entityManager.createQuery(
"select p from Post p", Post.class)
.setHint(QueryHints.HINT_READONLY, true)
.getResultList();
```

7. Qu'est-ce que la démarcation transactionnelle, comment fonctionne-t-elle avec @Transactional ?

/ 0,5

Une transaction est définie par un début et une fin qui peut être soit une validation des modifications (***commit***), soit une annulation des modifications effectuées (***rollback***). On parle de **démarcation transactionnelle** pour désigner la portion de code qui doit s'exécuter dans le cadre d'une transaction.

Avec le "@Transactionnal", on va utiliser **l'AOP** pour ouvrir notre transaction avant la méthode sur laquelle s'applique l'annotation et ensuite commit ou rollback cette dernière à la fin de la méthode en fonction de si une exception a été levé dans la méthode.

8. Qu'est-ce que le OpenSessionInView pattern ? Qu'implique-t-il en termes de performances ?

/ 1

L'article suivant explique exactement l'OpenSessionInView [The Open Session In View Anti-Pattern - Vlad Mihalcea](#).

Pour résumer, c'est un anti-pattern, c'est-à-dire qu'il est une réponse commune à un problème récurrent qui n'est pas efficace et qui est contre-productif.

Le principe est de pallier aux problèmes de "LazyInitializationException" levés par hibernate lorsque la session n'est plus active et que l'on a besoin de fetch les données qui étaient LAZY.

Le principe du pattern est d'ouvrir une session lorsqu'on va accéder à notre endpoint, au lieu de laisser la couche métier gérer le contexte de persistance. Le contexte de persistance est ouvert dans un filtre et reste ouvert pour que la couche de VIEW puisse accéder aux données. On n'a plus de souci au niveau du fetch de données.

En contrepartie, on peut avoir des problèmes de performances. Par exemple, la connexion à la base de données reste ouverte pendant toute la phase de rendering, ce qui augmente le nombre de connexions ouvertes en même temps et surtout la congestion sur le pool de connexions.

9. Quand peut-on être confronté au N+1 problem ? Quels sont les impacts ? Et comment le résout-on ?

/ 1

Avec les fetch de type LAZY si on itère sur les entitées liées.

Pour pallier le problème on peut faire "**join fetch**" dans les requêtes JPQL. Il est possible de faire la même chose avec Criteria.

Attention, le N+1 ne vient pas systématiquement à chaque fois cf :

```
Secteur secteur = entityManager.find(Secteur.class, 1l);
System.out.println(secteur);
secteur.getLogements().forEach(l -> System.out.println(l));

produit ⇒

Hibernate: select secteur0_.id as id1_1_0_, secteur0_.nom as nom2_1_0_ from Secteur secteur0_ where secteur0_.id=?

com.takima.formation.hibernate.examples.model.Secteur@5eabff6b

Hibernate: select logements0_.secteur_id as secteur_7_0_0_, logements0_.id as id2_0_0_, logements0_.id as id2_0_1_,
logements0_.loyer as loyer3_0_1_, logements0_.nombrePiece as nombrepi4_0_1_, logements0_.secteur_id as
secteur_7_0_1_, logements0_.superficieTerrain as superfic5_0_1_, logements0_.charges as charges6_0_1_,
logements0_.DTYPE as dtype1_0_1_ from Logement logements0_ where logements0_.secteur_id=?

Logement{id=5, loyer=1800, nombrePiece=3}Appartement{charges=95}
Logement{id=6, loyer=900, nombrePiece=3}Appartement{charges=100}
```

On n'a ici que 2 selects :-D

Explications détaillées ici

10. Pourquoi une entité ne doit pas être **final** ?

/ 0,5

D'après l'article suivant, vous comprendrez rapidement pourquoi on ne doit pas définir d'entité en final : Hibernate ORM 5.4.30.Final User Guide (jboss.org). Pour résumer, une des fonctionnalités centrale d'hibernate est le Lazy loading, la possibilité de charger les données plus tard ou pas d'une entité. Cependant, si l'entité est finale, on va être obligé de la charger.

11. Expliquer ce qu'est l'AOP, et les concepts associés

/ 1

L'AOP (**Aspect Oriented Programmation)** est un **paradigme de programmation** qui vise à accroître la modularité en permettant la séparation du code métier (qui nous intéresse) du code transverse (transactions, logging, etc…). Elle permet d'ajouter un comportement sans toucher au code actuel. Ça évite d'alourdir le code.

On parlera d'**advice** pour le code que l'on veut ajouter à notre fonction et on définit grâce à

des **pointcuts** l'endroit où l'advice devra être ajouté (avec des springs, on utilise des annotations).

Par exemple, on veut ajouter un log d'entrée et de sortie à toute fonction qui commence par set ou encore ouvrir une transaction avant une méthode et la fermer à la fin.

12. Donner un cas d'utilisation pratique du cache d'Hibernate (L1 ou/et L2)

/ 0,5

On peut mettre dans le cache L2 des données pas trop nombreuses et ne changeant que rarement et utilisées régulièrement (ex: liste des départements Français).

13. Qu'est-ce que Spring ? Quels sont les modules de Spring que vous avez déjà utilisés ?

/ 0,5

Spring est un framework Java permettant de développer des applications web. C'est une alternative plus légère à JEE.

Spring utilise principalement deux concepts : l'inversion de dépendance (IoC) et l'AOP (aspect oriented programming).

Modules utilisés : (faire la liste en fonction)

- Spring Security
- Spring JPA
- Spring MVC
- …

Voir https://springtutorials.com/introduction-to-spring-modules/

14. Considérons le mapping suivant :

```
@Entity(name = "Person")
public static class Person {

      @Id
      private Long id;

      @OneToMany(cascade = CascadeType.ALL)
      private List<Phone> phones = new ArrayList<>();

      //Getters and setters are omitted for brevity
}
```

Quelle classe Hibernate utilise en interne pour gérer la collection ?

/ 1

Les **BAGS**

Il est en effet troublant de se dire que ce n'est pas une List mais en interne hibernate utilise ses propres formats.
Pour avoir une vraie List il faut préciser le mapping des identifiants avec @OrderBy ou @OrderColumn
Hibernate ORM 5.4.30.Final User Guide (jboss.org)

15. Pourquoi vaut-il mieux utiliser les wrappers que les types primitifs pour les identifiants en JPA / Hibernate ?

/ 0,5

Providing identifiers

16. Considérons l'entité suivante :

```
@Entity
public class Visiteur {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String nom;

  private String prenom;
```

et la configuration suivante :
`hibernate.jdbc.batch_size = 30`
Que peut-on en conclure quant à l'optimisation des performances ?

/ 0,5

Cela ne sert à rien de configurer une batch_size avec des générateurs de type IDENTITY puisque Hibernate / JPA désactivent les batch pour ce type de générateurs.
En conclusion changer cette valeur qui doit être comprise entre 10 et 30 dans la plupart des cas n'apporte rien en termes d'améliorations de performances.

17. Quels sont les moyens d'éviter les roundtrips (allers/retours) avec la base de données avec Hibernate pour du batch INSERT ou UPDATE?

/ 1

Au départ

1. Support de batch par défaut chaque INSERT et UPDATE est effectué séparément (beaucoup de requêtes)

2. mettre en place des insertions en mode batch avec
   - hibernate.jdbc.batch_size : 30 (par exemple)

X Problème cela ne concerne que les INSERT et non les UPDATE
X JDBC ne fait par défaut du batch que sur une table donnée

3. cela ne suffit pas il faut aussi ordonner les opérations avec
   - hibernate.order_inserts : true
   - hibernate.order_updates : true

✓En triant les inserts on fait d'abord une table (pleins d'opérations en 1 aller/retour) puis une autre table et ainsi de suite et enfin les UPDATE qui ne sont pas batchés.

Lien vers les explications détaillées et le case qui illustre bien et impressionne

18. Quelles sont les valeurs possibles pour hbm2ddl.auto ? Quelles sont les bonnes pratiques ou alternatives en fonction de l'environnement ?

/ 0,5

Cela dépend de si on est en hibernate natif ou en JPA.

Pour JPA paramètre **`javax.persistence.schema-generation.database.action`** :

- none
- create
- drop
- drop-and-create

Pour un environnement de développement ou de tests d'intégration automatique on peut tolérer drop-and-create.

Pour les autres environnements seul NONE est conseillé cela concerne :
- qualif
- preprod
- prod

Il est fortement recommandé d'utiliser un outil externe comme Liquibase ou Flyway.
Plus d'informations ici

19. À quoi sert l'annotation `@NaturalId` dans hibernate ?

/ 0,5

@NaturalId
Sert à identifier la clé "naturelle" ou fonctionnelle d'une entité.

Permet avec hibernate natif de faire des recherches sur cette clé

```
Book book = entityManager
      .unwrap(Session.class)
      .bySimpleNaturalId( Book.class )
      .load( "978-9730228236" );
```

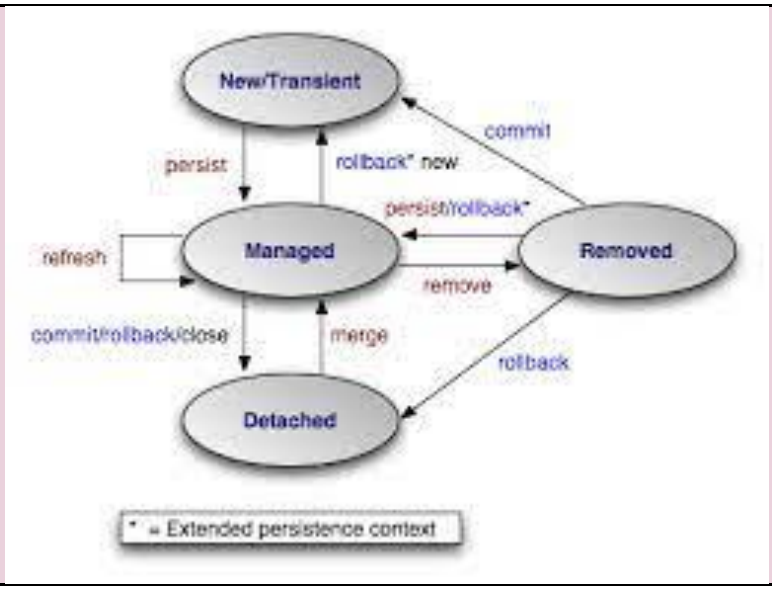
20. Qu'est-ce que le 'Dirty Checking' dans hibernate ? Quels problèmes peut-il poser ?

/ 0,5

dirty checking

21. Dessiner le cycle de vie d'une entité dans Hibernate ?

/ 0,5



22. Quand et pourquoi utiliser Criteria plutôt que JPQL ?

/ 0,5

Quand le nombre de paramètres de la requête est variable :
1 - Criteria est fait pour cela c'est un design pattern décorateur pour changer le comportement de la requête au runtime
2 - Cela permet d'éviter les concaténations de String malheureuses
3 - Cela permet de créer des bouts de requêtes et de les réutiliser
4 - L'API criteria de JPA est Typesafe ce qui ne gâche rien
5 - Pour faire des bulk opérations en UPDATE, DELETE

23. Quel est l'intérêt des NamedQueries ?

/ 0,5

Contrairement aux requêtes JPQL ou HQL entityManager.createQuery("blabla", MonEntite.class); les named queries ont un nom @NamedQuery(name="mon-joli-nom", query="blabla")

ainsi elles peuvent être utilisées et ré-utilisées à plusieurs endroits sans duplication de code :-)

On peut même utiliser des hints par exemple read-only pour désactiver le dirty check :

```
@NamedQuery(
    name = "get_read_only_person_by_name",
    query = "select p from Person p where name = :name",
    hints = {
        @QueryHint(
```

```
                name = "org.hibernate.readOnly",
                value = "true"
            )
        }
)
```

Détails

24. À quoi sert @Transactional(propagation = Propagation.REQUIRES_NEW)) ? Pourquoi l'utiliser ?

/ 0,5

Ouvre une nouvelle transaction même si une est déjà présente en suspendant la transaction éventuellement existante.

Explications ici.

25. Comment faire de l'injection par Nom dans Spring ?

/ 0,5

Soit avec l'annotation @Qualifier("bean-id"), soit avec l'annotation @Bean en précisant une valeur.

26. Qu'est-ce que CDI ?

/ 0,5

Standard de JEE pour faire de l'injection de dépendances.

Au lieu d'utiliser les annotations Spring @Autowired on doit utiliser @Inject

https://www.baeldung.com/java-ee-cdi

27. Comment gérer une clé primaire composite avec hibernate?

/ 0,5

- avec une clé Embeddable et un @EmbeddedId dans le mapping
- une classe à part et un @IdClass
- (possibilité pour les associations voir la doc)

Composite Identifiers

28. À quoi sert l'annotation @Lob dans Hibernate ? Quelles sont les limites ?

/ 0,5

Permet d'utiliser des Large Object binary de type :

- BLOB ie binaires
- CLOB ie chaînes de caractères

Attention le management de la mémoire pour les LOB est un peu complexe. En effet, leur utilisation optimise l'occupation de l'espace pour la table dans le fichier lié sans pré-réserver un espace trop grand car l'objet est sauvé "ailleurs" en fonction du SGBD et de ne mettre qu'un pointeur dans la table.

En contrepartie, supprimer l'enregistrement ne supprime que le pointeur. Ainsi, il faut de temps à autre "nettoyer" les LOB non référencés :
ALTER TABLE <nom table> SHRINK SPACE;

29. À quoi sert l'annotation @Primary dans Spring ?

/ 0,5

Permet de signaler à Spring un Bean à utiliser en priorité pendant l'injection s'il y a plusieurs candidats.

30. Qu'est-ce que le ChainedTransactionHandler dans Spring et quand l'utiliser ?

/ 0,5

Le ChainedTransactionHandler permet de créer un transaction manager pour plusieurs bases de données.

Il devient utile lorsqu'on a plusieurs bases de données que l'on veut pouvoir fetch dans une même transaction.

31. Comment faire des projections dans Hibernate et à quoi cela sert-il ?

/ 0,5

De manière générales en SQL les projections sont les fonctions que l'on met dans la clause SELECT :

- distinct
- avg
- min
- max
- ....

Généralement les projections permettent de réduire le volume de données échangé entre la base et le client.

En hibernate, on peut utiliser tous les types de projections que ce soit en JPQL ou avec l'API Criteria.

Il y a même une possibilité de récupérer des POJO directement via un mécanisme appelé "INSTANCIATION DYNAMIQUE" en JPQL cela s'appelle "CONSTRUCTOR EXPRESSION"

```
CallStatistics callStatistics = entityManager.createQuery(
     "select new org.hibernate.userguide.hql.CallStatistics(" +
     "    count(c), " +
     "    sum(c.duration), " +
     "    min(c.duration), " +
     "    max(c.duration), " +
     "    avg(c.duration)" +
     ")  " +
     "from Call c ", CallStatistics.class )
.getSingleResult();
```

Où CallStatistics est un DTO qui n'est pas mappé.

Lien vers la référence.

32. Que permet l'annotation @RestController par rapport à @Controller ?

/ 0,5

Permet de ne pas mettre @ResponseBody sur les méthodes qui renvoient de la donnée GET de plus un controller peut retourner une view mais pas un RestController.

33. À quoi sert l'annotation @Version ? Quel pattern d'architecture permet-elle ?

/ 0,5

Sert à incrémenter les versions d'une instance d'entité.

C'est utilisé par l'optimistic locking qui sert de verrou transactionnel pour gérer la concurrence.

Optimistic Locking

34. Qu'est-ce qu'une StatelessSession à quoi cela sert-il ?

/ 0,5

Stateless Session

C'est une session sans state. Les entités récupérées sont dans un état non persisté.

35. Quand utiliser entityManager.getReference plutôt que find ?

/ 0,5

Pour insérer un objet avec sa relation. Si on connaît la relation cela permet d'économiser le select qui aurait été fait pour obtenir l'entité liée à l'état managed.
Plus d'explications ici