

Cochez la/les bonnes réponses.

Dans une page HTML, on veut écrire un lien "Ajout" qui, lorsqu'on clique dessus, passe un paramètre "opération" ayant pour valeur "ajout". Quelle est alors la balise HTML ?

- Ajout
- Ajout
- Ajout

- La balise <a> permet de créer des liens hypertextes, et l'attribut href spécifie l'URL vers laquelle le lien doit pointer. Pour passer un paramètre "opération" avec la valeur "ajout" lorsque l'utilisateur clique sur le lien, on peut inclure le paramètre dans l'URL à l'aide de la syntaxe "nom=valeur". Dans ce cas, l'URL serait "Servlet?opération=ajout".
- Parmi les choix proposés, la seule bâise HTML qui inclut correctement le paramètre dans l'URL est Ajout. Les deux autres choix ne sont pas corrects car ils utilisent une syntaxe invalide pour inclure le paramètre dans l'URL.

Dans une page HTML, on a une balise Ajout

- Cela signifie que :
- Il s'agit d'un passage de paramètre avec la méthode HTTP POST
 - Il s'agit d'un passage de paramètre avec la méthode HTTP GET

- La servlet appelée lorsque l'on clique sur le lien est Servlet

- Dans le langage HTML, le symbole "?" est utilisé pour séparer l'URL de base d'une liste de paramètres de requête. Les paramètres de requête sont utilisés pour transmettre des informations supplémentaires à la page de destination lorsque l'utilisateur clique sur un lien hypertexte ou soumet un formulaire Web.

- Le symbole "?" est suivi du premier paramètre de requête, qui est représenté par un nom de paramètre, suivie du signe égal "=", suivie de la valeur du paramètre. Si la page de destination doit inclure plusieurs paramètres, les paramètres supplémentaires sont ajoutés en utilisant le signe "&".

- Par exemple, dans l'URL "<https://www.example.com/search?query=html&category=web>", le symbole "?" est utilisé pour séparer l'URL de base "<https://www.example.com/search>" des paramètres de requête "query=html" et "category=web". La page de destination peut alors utiliser ces informations pour effectuer une recherche sur le mot clé "html" dans la catégorie "web".

- La méthode HTTP GET est utilisée pour demander des ressources à un serveur Web en spécifiant des paramètres dans l'URL de la requête. Dans cet exemple, le paramètre "action" avec la valeur "Servlet" est ajouté à l'URL en utilisant le symbole "?" pour le séparer de l'URL de base. Lorsque l'utilisateur clique sur le lien, le navigateur envoie une requête HTTP GET à l'URL spécifiée, en incluant les paramètres dans l'URL.
- En revanche, la méthode HTTP POST est utilisée pour soumettre des données à un serveur Web pour traitement, généralement à partir d'un formulaire HTML. Les paramètres sont inclus dans le corps de la requête POST plutôt que dans l'URL.

- La réponse "Dans la JSP pour récupérer les propriétés (comme l'émetteur ou la date) de la requête request" est incorrecte car la JSP est utilisée pour générer la réponse HTML qui est

Dans une page HTML, on a un formulaire :

```
<form method="POST" action="GET">
    login: <input type="text" name="login"><br>
    password : <input type="text" name="motdepasse"> <br>
    <input type="submit" value="OK">
    <input type="hidden" name="password" value="true"></form>
```

Cela signifie que :

- Il s'agit d'un passage de paramètre avec la méthode HTTP POST
 - Il s'agit d'un passage de paramètre avec la méthode HTTP GET
 - Deux chaînes de caractères saisies seront passées dans des paramètres login et motdepasse
 - Il faut soumettre le formulaire si l'utilisateur tape OK au clavier
 - La saisie du paramètre password doit être cachée lors de la saisie
- Quand un formulaire HTML avec la méthode POST est validé, les paramètres sont passés
- Dans l'URL présente dans l'en-tête de la requête sous forme URL?param1=value1¶m2=value2...
 - Dans le corps de la requête sous la forme param1=value 1¶m2= value2 ...
 - Dans le corps de la requête en format XML
 - Dans le corps de la requête en format JSON
 - La bonne réponse est : Les paramètres ne sont pas passés dans l'URL, mais dans le corps de la requête sous la forme param1=value1¶m2=value2...
 - Lorsqu'un formulaire HTML est soumis avec la méthode POST, les paramètres ne sont pas passés dans l'URL présente dans l'en-tête de la requête. Au lieu de cela, les paramètres sont inclus dans le corps de la requête HTTP. Les paramètres sont codés dans un format spécifique appelé "application/x-www-form-urlencoded", qui encode les caractères spéciaux tels que les espaces et les caractères d'ampersand (&), en séquences de caractères spécifiques.
 - Les réponses "Dans l'URL présente dans l'en-tête de la requête" et "Dans le corps de la requête en format XML" sont incorrectes car elles ne décrivent pas la manière dont les paramètres sont envoyés lorsque la méthode POST est utilisée. La réponse "Dans le corps de la requête en format JSON" est également incorrecte car cela dépend de la manière dont le formulaire est implémenté et si une conversion en JSON est effectuée.

Dans un schéma MVC, la ligne de code suivante est utilisée

Properties properties = (Properties)request.getAttribute("properties");
Properties properties = (Properties)request.getAttribute("properties");

- Dans la servlet pour récupérer les propriétés (comme l'émetteur ou la date) de la requête request
- Dans la JSP pour récupérer les propriétés (comme l'émetteur ou la date) de la requête request
- La ligne de code donnée utilise la méthode getAttribute() de l'objet HttpServletRequest pour récupérer un attribut nommé "properties" et le stocke dans un objet de type Properties. Cela signifie que la ligne de code est utilisée dans la servlet pour récupérer des propriétés de la requête HTTP entrante et stockées dans un objet Properties
- La réponse "Dans la JSP pour récupérer les propriétés (comme l'émetteur ou la date) de la requête request" est incorrecte car la JSP est utilisée pour générer la réponse HTML qui est

- envoyée au navigateur de l'utilisateur. La JSP peut accéder aux attributs de la requête, mais elle ne devrait pas être utilisée pour récupérer des propriétés de la requête HTTP.
- La réponse "Dans la JSP pour récupérer un objet passé en paramètre par la serviet" est également incorrecte car la ligne de code donnée utilise la méthode `getAttribut()` pour récupérer un attribut de la requête HTTP et non pour récupérer un objet passé en paramètre par la serviet. La méthode `getAttribut()` est utilisée pour récupérer des attributs de la requête HTTP qui ont été définis dans la serviet avant que la requête ne soit transmise à la JSP.

Dans la Facade, la variable EntityManager em;

-
-
-
-
-

Dans la déclaration de la classe Facade, @ Singleton

-
-
-
-
-

- La Facade est une couche intermédiaire entre les couches de présentation et d'accès aux données (DAO), elle sert à cacher la complexité des opérations de persistance et à fournir une interface simple et cohérente pour l'accès aux données.
- La méthode "persist" est utilisée pour créer une nouvelle entité persistante dans la base de données à partir d'un objet Java. Cette méthode associe l'objet à la session de persistance et prépare une instruction SQL INSERT qui sera exécutée lors de la prochaine synchronisation avec la base de données.
- Les autres réponses sont incorrectes car :
`em.store(p)` n'est pas une méthode valide de l'EntityManager.
L'annotation @Entity est utilisée pour marquer une classe comme une entité persistante, elle ne permet pas la création d'un objet persistant dans la base de données.

Devant la déclaration de la classe Facade, @ Singleton

-
-
-
-
-

-
-
-
-
-



Retourne une Collection<Personne>

- En effet, la méthode `createQuery` est utilisée pour créer une requête en utilisant le langage de requête JPQL (Java Persistence Query Language) qui est un langage orienté objet, pour interroger les données stockées dans la base de données via JPA (Java Persistence API). La requête JPQL créée ici demande à la base de données de retourner toutes les instances de la classe Personne qui ont un âge supérieur à 20.
- Cette requête est ensuite exécutée par la méthode `getResultList()`, qui renvoie une liste des résultats de la requête.

Dans une serviet, pour avoir une référence à la Facade, on peut

- - Déclarer une variable précédée de l'annotation @Facade
 - Déclarer une variable précédée de l'annotation @EJB : L'annotation @EJB permet de faire injecter la référence à la Facade par le container EJB.
 - Déclarer une variable et instancier la classe Facade
- Dans la ligne de code suivante :
- ```
request.getRequestDispatcher(URL).forward(request, response);
```
- URL peut être une page HTML
  - URL peut être une page JSP
  - La ligne peut être suivie d'autres lignes renvoyant du contenu supplémentaire.
  - Génère une serviet dont l'exécution génère la page renvoyée
  - URL peut être une page JSP ou une autre ressource à laquelle on veut rediriger la requête. La méthode `getRequestDispatcher` récupère le contrôle de la requête et le transfère à la ressource désignée par l'URL fournie en paramètre. La méthode `forward` transfère le contrôle de la requête à la ressource cible en préservant la requête originale, qui sera gérée par la ressource cible.
  - La ligne ne peut pas être suivie d'autres lignes renvoyant du contenu supplémentaire, car forward transfère la requête au contrôle d'une autre ressource, donc le contenu doit être généré par cette ressource et non par la serviet courante.
  - La ligne ne génère pas une serviet dont l'exécution génère la page renvoyée, car elle ne fait que transférer le contrôle de la requête à une autre ressource.

Dans un schéma MVC, lors du passage d'un paramètre name entre une page HTML et une Serviet, on récupère la valeur du paramètre avec :

- String name = request.getParameter("name");
  - String name = request.getAttribute("name");
  - String name = request.getAttribute("name");
  - La méthode `getAttribute`, quant à elle, est utilisée pour récupérer des attributs stockés dans l'objet request par d'autres composants (comme une JSP ou une Serviet). Elle ne sert pas à récupérer directement les paramètres passés dans l'URL.
  - Enfin, la méthode `getName` n'existe pas dans l'objet `HttpServletRequest`.
- Si un entity bean est retourné par la Facade à la Servlet
- On dit que le bean est détaché
  - Le bean est relié de la BD
  - Si le bean est repassé par la serviet à la Facade, le bean peut synchroniser ses beans avec EntityManager();

Un bean est dit détaché lorsqu'il a été extrait de la session de persistance, généralement après une opération de lecture ou de récupération de données depuis la base de données, et n'est plus géré par le contexte de persistance. Cela signifie que toutes les modifications apportées à ce bean ne seront pas automatiquement synchronisées avec la base de données. Si un bean détaché doit être mis à jour, il doit être ré-attaché au contexte de persistance avant que les modifications ne puissent être effectuées en utilisant la méthode merge() de l'EntityManager.

Dans un schéma MVC, le contrôleur effectue

- Des appels à la Facade
  - Des appels à des entity
  - Des appels à des JSP

- Dans un schéma MVC, le contrôleur est le composant qui gère les interactions entre la vue et le modèle. Il est responsable de la gestion des requêtes et des réponses HTTP et de l'appel aux composants appropriés pour traiter ces requêtes.

- Le contrôleur effectue principalement des appels à la Facade, qui est le composant du modèle qui encapsule la logique de persistance des entités dans la base de données. Le contrôleur utilise les méthodes exposées par la Facade pour effectuer des opérations de lecture, d'écriture, de mise à jour et de suppression sur les entités.

- Le contrôleur peut également effectuer des appels à des JSP pour générer la vue en fonction des données récupérées du modèle. Cependant, il ne fait pas d'appels directs aux entités, car cela violerait le principe de séparation des préoccupations du modèle MVC. Le contrôleur utilise plutôt la Facade pour interagir avec les entités de manière encapsulée et sans connaître les détails de mise en œuvre de la persistance.

Dans un schéma MVC une ISP

- Effectué des appels à la BD pour récupérer les données à afficher
  - Reçoit en paramètre de la servlet les données à afficher
  - Génère la page renournée au client
  - Est appellée depuis la Faceade

### Problème (13 points)

ԵՐԱԾՈՒՅՈՒՆ:

- possède une clé primaire : int id;
  - possède 1 champs : String nom;
  - L'entity Entreprise:
    - possède une clé primaire : int id;
    - possède 1 champs: String numero;
  - L'entity Chantier:
    - possède une clé primaire : int id;
    - possède 1 champs : String adresse;

Une entreprise peut avoir plusieurs chantiers et un chantier est réalisé par une seule entreprise

Un chantier peut nécessiter plusieurs ouvriers et un ouvrier ne travaille que sur un seul chantier.

Ces relations sont toutes bi-directionnelles.

On ne donnera pas les settter/getters ni les imports

Une relation unidirectionnelle signifie qu'un objet (par exemple, l'entité A) a une référence vers un autre objet (par exemple, l'entité B), mais l'entité B n'a pas de référence vers l'entité A. En d'autres termes, l'objet A sait que l'objet B existe et peut y accéder, mais l'objet B ne sait pas que l'objet A existe. En revanche, une relation bidirectionnelle signifie que deux objets (par exemple, les entités A et B) ont des références les uns vers les autres. Dans ce cas, l'entité A peut accéder à l'entité B et vice versa.

Question 1 : Donner l'implantation des 2000 unités (3 points)

```
@Entity
public class Ouvrier {
 @Id
 @GeneratedValue
 int id;
 String nom;
```

```
@JoinColumn(name="chantier_id")
@ManyToOne
Chantier chantier;
```

```
@Entity
public class Chantier {
 @Id
 @GeneratedValue(value = GenerationType.IDENTITY)
 int id;
 String address;
```

```
@OneToMany(mappedBy=chantier)
Listsurveillants;
```

```
@JoinColumn(name="enterprise_id")
@ManyToOne
Enterprise enterprise;
```

```
@Entity
public class Enterprise{
 @Id
 @GeneratedValue
```

```

int id;
String numero;

@OneToMany(mappedBy="entreprise")
List chantiers;
}

- La propriété @JoinColumn est utilisée pour spécifier la colonne utilisée pour la jointure dans la table de l'entité propriétaire d'une relation @ManyToOne ou @OneToOne.
- Dans l'exemple donné, la relation entre l'entité Ouvrier et Chantier est une relation ManyToOne bidirectionnelle. L'annotation @JoinColumn(name="chantier_id") est utilisée pour spécifier que la colonne chantier_id de la table Ouvrier est utilisée pour stocker la clé étrangère qui référence la clé primaire de l'entité Chantier.
- En d'autres termes, cette annotation permet de lier la clé étrangère de la table Ouvrier à la colonne correspondante de la table Chantier.
- Il est important de noter que @JoinColumn n'est utilisé que dans l'entité propriétaire de la relation (ici, Ouvrier), et non dans l'entité inverse (ici, Chantier), car la relation est bidirectionnelle et que l'entité Chantier possède déjà une annotation @OneToMany(mappedBy="chantier") pour spécifier l'association inverse.
- on ne peut pas utiliser mappedBy=ouvriers car ouvriers est le nom de l'attribut dans la classe Ouvrier qui fait référence à la classe Chantier avec l'annotation @ManyToOne. Le paramètre mappedBy doit faire référence au nom de l'attribut dans la classe Chantier qui contient la relation @OneToMany. Dans ce cas, l'attribut dans la classe Chantier est ouvriers, donc on utilise mappedBy=chantier pour spécifier la relation.

}

```

Question 2 : donnez une possible instanciation en BD (précisez les PrimaryKeys et ForeignKeys de chaque table) (2 points)

```

Table Ouvrier: | id | nom | chantier_id |
Primary key: id
Foreign key: chantier_id référence la clé primaire id de la table Chantier

```

```

Table Entreprise: | id | numero |
Primary key: id
Table Chantier: | id | adresse | entreprise_id |
Primary key: id
Foreign key: entreprise_id référence la clé primaire id de la table Entreprise

```

Question 3 : Donner le code des deux méthodes de la Facade suivantes (4 points)

```

void ajouter_ouvrier(String nom, int id_chantier);
// ajouter un nouvel ouvrier à un chantier existant
void changer_entreprise(int id_chantier, int id_entreprise);
// change l'entreprise opérant sur un chantier

```

Question 4 : on veut maintenant qu'un chantier puisse être réalisé par plusieurs entreprises et qu'un ouvrier puisse travailler sur plusieurs chantiers  
Les relations de l'entity Chantier sont les suivantes:

```

@Entity
public class Chantier {

```

```

@ManyToMany(mappedBy="chantiers")
Collection<Entreprise> entreprises;
@ManyToMany(mappedBy="chantiers")
Collection<Ouvrier> ouvriers;
}

Donner la définition des deux autres entity et donner la nouvelle version de la méthode
ajouter_ouvrier de la Question 3 (2 points)
}

L'annotation @JoinTable est utilisée pour définir la table de jointure à utiliser pour la relation
many-to-many entre les entités Entreprise et Chantier. Dans ce cas, la table de jointure s'appelle
"chantier_entreprise".
L'utilisation de la clause @JoinTable est nécessaire pour les relations many-to-many, car dans ce
type de relation, une table de jointure est nécessaire pour représenter la relation entre les entités.
joinColumns spécifie la colonne de la table de jointure qui contiendra l'ID de l'entreprise, tandis que
inverseJoinColumns spécifie la colonne de la table de jointure qui contiendra l'ID du chantier.

@Entity
public class Entreprise {
 @Id
 private int id;
 private String numero;
 @ManyToMany
 @JoinTable(name="chantier_entreprise",
 joinColumns=@JoinColumn(name="entreprise_id"),
 inverseJoinColumns=@JoinColumn(name="chantier_id"))
 private Collection<Chantier> chantiers;
}

private void ajouter_ouvrier(String nom, int id_chantier) {
 Ouvrier o = new Ouvrier();
 o.setNom(nom);
 o.setChantier(em.find(Chantier.class, id_chantier));
 Chantier c = em.find(Chantier.class, id_chantier);
 for (Entreprise e : c.getEnterprises()) {
 o.getEnterprises().add(e);
 e.getOuvriers().add(o);
 em.merge(e);
 }
 em.persist(o);
}

Dans la boucle for, pour chaque entreprise, on ajoute l'ouvrier à sa liste d'ouvriers en utilisant la
méthode e.getOuvriers().add(o), puis on ajoute l'entreprise à la liste des entreprises de l'ouvrier en
utilisant la méthode o.getEnterprises().add(e).

Question 5 : On veut afficher la liste des entreprises dans une JSP. Donner l'implantation de
la méthode de la Facade permettant de récupérer cette liste (1 point)

public Collection<Entreprise> getEnterprises() {
 TypedQuery<Entreprise> query = em.createQuery("SELECT e FROM Entreprise e",
 Entreprise.class);
 return query.getResultList();
}

Cette méthode utilise l'interface TypedQuery pour créer une requête JPA permettant de récupérer
toutes les instances de l'entité Entreprise. Cette requête est définie sous forme d'une chaîne de
caractères représentant une requête JPQL ("SELECT e FROM Entreprise e").
```

La méthode `createQuery()` prend en paramètres cette requête ainsi que la classe d'entité attendue en résultat (ici, `Entreprise.class`), et renvoie un objet `TypedQuery<Entreprise>`.  
 Ensuite, la méthode invoque la méthode `getResults()` sur l'objet `TypedQuery` pour récupérer la liste des instances d'entreprises correspondant à la requête JQL.  
 En résumé, cette méthode permet de récupérer toutes les entreprises stockées en base de données en utilisant une requête JQL et l'interface `TypedQuery`.

Question 6: Si dans la JSP, on veut pouvoir afficher pour chaque entreprise la liste de ses chantiers, que faut-il ajouter pour être sur que les données soient accessibles dans la JSP ? (1 point)

Solution 1:

```
public void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
 Collection<Entreprise> entreprises = facade.getEnterprises();
 request.setAttribute("entreprises", entreprises);
 request.getRequestDispatcher("ma_jsp.jsp").forward(request, response);
}
```

Solution 2:  
 Il faut ajouter une boucle dans le code JSP qui permet de parcourir la liste des entreprises et pour chaque entreprise, afficher sa liste de chantiers.  
 Voici un exemple de code JSP qui permet d'afficher la liste des chantiers pour chaque entreprise :

```
<%-- boucle pour parcourir la liste des entreprises --%>
<c:forEach var="entreprise" items="#{entreprises}">
 <h2>${entreprise.numero}</h2>

 <%-- boucle pour parcourir la liste des chantiers de l'entreprise --%>
 <c:forEach var="chantier" items="#{entreprise.chantiers}">
 ${chantier.nom}
 </c:forEach>

</c:forEach>
```

Examen 2018

La méthode `createQuery()` prend en paramètres cette requête ainsi que la classe d'entité attendue en résultat (ici, `Entreprise.class`), et renvoie un objet `TypedQuery<Entreprise>`.  
 Lire l'ensemble des énoncés avant de commencer à répondre. La clarté, la précision et la concision des réponses, ainsi que leur présentation matérielle, seront des éléments importants d'appreciation. Donnez essentiellement les programmes Java demandés avec les annotations. Dans vos programmes, vous n'avez pas à programmer les imports et le traitement des exceptions.

On se propose d'implanter une application Web permettant de gérer une bibliothèque de livres. Cette application est structurée suivant le modèle MVC / JEE vu en cours.

L'application est composée de 3 entity beans.

L'entity Livre:

- possède une clé primaire : int id;
- possède 1 champs : String titre;
- possède une référence à son auteur (entity Auteur);
- possède une liste de références aux exemplaires du livre (entity Exemplaire)

L'entity Auteur:

- possède une clé primaire : int id;
- possède 1 champs : String nom;

L'entity Exemplaire:

- possède une clé primaire : int id;
- possède 1 champs : Boolean disponible;
- possède une référence au livre dont il est un exemplaire (entity Livre)

Un livre a plusieurs exemplaires dans la bibliothèque. Un livre a un auteur. Les clés primaires sont générées automatiquement par la base de données.

La Facade fournit (notamment) les méthodes suivantes :

```
Collection<Livre> ListerLivres();
// retourne la liste des livres
boolean VerifierDispo(int idLivre);
// vérifie si un exemplaire est disponible / idLivre est l'id du livre
Collection<Livres> ListerLivres(int idAuteur);
// retourne la liste des livres d'un auteur / idAuteur est l'id de l'auteur
```

Question 1 (3 points)

Implantez ces 3 entity beans (en Java + annotations).  
 Décrivez l'implantation probable de ce schéma en base de données.

```
@Entity
public class Livre {
 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 private int id;
 @Column
 private String titre;
```

```

@ManyToOne(name = "auteur_id")
@JoinColumn(name = "auteur_id")
private Auteur auteur;

@OneToMany(mappedBy = "livre")
private List<Exemplaire> exemplaires;

// Getters and setters
}

@Entity
public class Auteur {
 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 private int id;
 @Column
 private String nom;
 // Getters and setters
}

@Entity
public class Exemplaire {
 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 private int id;
 @Column
 private Boolean disponible;
 @ManyToOne(name = "livre_id")
 @JoinColumn(name = "livre_id")
 private Livre livre;
 // Getters and setters
}

```

### Question 3 (2 points)

L'implantation de la méthode `ListerLivres (int idAuteur)` de la Facade peut être optimisée en modifiant le schéma des entity beans. Proposez une telle modification et donnez la nouvelle implantation de la méthode `ListerLivres(int idAuteur)`.

Actuellement, pour récupérer tous les livres d'un auteur, on doit faire une requête sur la table `Livre` et filtrer les résultats en fonction de l'id de l'auteur. Cela peut être coûteux en termes de performance si la table `Livre` est très grande.  
Pour optimiser cela, on peut ajouter une relation bidirectionnelle `OneToMany` entre `Auteur` et `Livre`, en ajoutant une liste de `Livres` à la classe `Auteur`. De cette façon, on peut récupérer directement tous les livres d'un auteur à partir de la classe `Auteur`, sans avoir à passer par la table `Livre`.

Voici comment on peut modifier les classes pour cela :

```

@Entity
public class Auteur {
 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 private int id;
 @Column
 private String nom;
 @OneToMany(mappedBy = "auteur")
 private List<Livre> livres;
 // Getters and setters
}

Question 2 (3 points)
Donnez une implantation de la Facade (singleton) avec ses 3 méthodes.

@Singleton
public class Facade {
 @PersistenceContext
 private EntityManager em;

 public Collection<Livre> listerLivres() {
 TypedQuery<Livre> query = em.createQuery("SELECT l FROM Livre l",
 Livre.class);
 }
}

```

```

@Entity
public class Livre {
 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 private int id;
 @Column
 private String titre;
 @ManyToOne(name = "auteur_id")
 private Auteur auteur;
}

@OneToMany(mappedBy = "livre")
private List<Exemplaire> exemplaires;
}

// Getters and setters
}

@Entity
public class Auteur {
 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 private int id;
 @Column
 private String nom;
 @ManyToMany(mappedBy = "auteurs")
 private List<Livre> livres;
}

// Getters and setters
}

Maintenant, pour récupérer tous les livres d'un auteur, on peut simplement récupérer l'objet Auteur correspondant à l'id de l'auteur et accéder à sa liste de Livres.
Voici la nouvelle implantation de la méthode ListerLivres(int idAuteur) :
public Collection<Livre> listerLivres(int idAuteur) {
 Auteur auteur = entityManager.find(Auteur.class, idAuteur);
 return auteur.getLivres();
}

Dans cette nouvelle implémentation, on utilise la requête JPQL pour récupérer l'objet Auteur correspondant à l'id de l'auteur, en joignant la liste de ses Livres. Ensuite, on peut simplement retourner la liste de Livres correspondant à cet Auteur.

Question 4 (1 point)
On veut qu'un livre puisse avoir plusieurs auteurs. Modifiez le schéma d'entity dans ce sens (donnez les nouveaux entity Livre et Auteur).

Pour permettre à un livre d'avoir plusieurs auteurs, il faut utiliser une relation ManyToMany entre les entités Livre et Auteur. Voici les nouvelles classes d'entités modifiées pour refléter cette relation :

@Entity
public class Livre {
 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 private int id;
 @Column
 private String titre;
}

// Ajout de l'exemplaire à la liste des exemplaires du livre

```

```

livre.getExemplaires().add(exemplaire);
}

// Mise à jour de la base de données
entityManager.persist(exemplaire);
entityManager.merge(livre);
}
}

Question 6 (4 points)
A partir du schéma précédent, supposons qu'une page HTML permet de saisir un nom de
personne.
L'application doit retourner la liste des livres (liste les titres des livres) dont la personne est
auteur. On se limite à cette séquence (lister les livres d'un auteur).
Donnez la page HTML, le code de la servlet, les modifications apportées à la Facade et la
JSP présentant le résultat.

Page HTML

<!DOCTYPE html>
<html>
<head>
<title>Liste des livres d'un auteur</title>
</head>
<body>

<form method="GET" action="LivresAuteurServlet">
<label for="nom">Nom de l'auteur :</label>
<input type="text" name="nom" id="nom" required>
<button type="submit">Rechercher</button>
</form>
</body>
</html>

Code Servlet

@WebServlet("/LivresAuteurServlet")
public class LivresAuteurServlet extends HttpServlet {

private static final long serialVersionUID = 1L;
private Facade facade;

@Override
public void init() throws ServletException {
super.init();
facade = new Facade();
}

Question 6 (4 points)
A partir du schéma précédent, supposons qu'une page HTML permet de saisir un nom de
personne.
L'application doit retourner la liste des livres (liste les titres des livres) dont la personne est
auteur. On se limite à cette séquence (lister les livres d'un auteur).
Donnez la page HTML, le code de la servlet, les modifications apportées à la Facade et la
JSP présentant le résultat.

Page HTML

<!DOCTYPE html>
<html>
<head>
<title>Liste des livres d'un auteur</title>
</head>
<body>
<h1>Liste des livres d'un auteur</h1>

<c:if test="${not empty titresLivres}">

<c:forEach items="${titresLivres}" var="titre">
${titre}
</c:forEach>

<c:if test="${empty titresLivres}">
<p>Aucun livre trouvé pour cet auteur.</p>
</c:if>
</c:if>

```

```

Retour à l'accueil
</body>
</html>

Question 7(2 points)
On veut maintenant que la séquence précédente affiche pour chaque livre de l'auteur: le
titre du livre et les noms de tous les auteurs.
Modifiez la JSP pour prendre en compte cette modification.
Comment être sûr que les auteurs référencés depuis ces livres seront accessibles dans la
JSP. Expliquez le problème et la solution.

Modification de la JSP

<!DOCTYPE html>
<html>
<head>
<title>Liste des livres d'un auteur</title>
</head>
<body>
<h1>Liste des livres d'un auteur</h1>
<c:if test="${not empty livres}">

<c:forEach items="${livresAuteur}" var="livre">
${livre.titre}
<c:forEach items="${livre.auteurs}" var="auteur">
${auteur.nom}
</c:forEach>
</c:forEach>

<c:if test="${empty livres}">
<p>Aucun livre trouvé pour cet auteur.</p>
</c:if>

```

Pour que les auteurs référencés depuis ces livres soient accessibles dans la JSP, il faut charger les auteurs en même temps que les livres. En effet, si l'on ne charge que les livres dans la méthode de la Facade, les auteurs ne seront pas accessibles dans la JSP. Pour résoudre ce problème, il faut modifier la méthode de la Facade pour qu'elle charge également les auteurs en même temps que les livres, comme suit :

```

public List<Livre> listerLivresAuteur(String nomAuteur) {
 Query query = entityManager.createQuery("SELECT l FROM Livre l JOIN l.auteurs a
WHERE a.nom = :nomAuteur");
 query.setParameter("nomAuteur", nomAuteur);
 List<Livre> livres = query.getResultList();
 for (Livre livre : livres) {
 livre.getAuteurs().size(); // force le chargement des auteurs pour chaque livre
 }
 return livres;
}

```

Dans la méthode listerLivresAuteur de la Facade, on récupère d'abord la liste des livres associés à l'auteur donné, mais les auteurs associés à chaque livre ne sont pas encore chargés (ils sont en quelque sorte en mode "lazy loading"). En exécutant livre.getAuteurs().size(), on force le chargement de la liste des auteurs pour chaque livre. Cela a pour effet de charger les auteurs en mémoire pour chaque livre, de sorte qu'ils seront disponibles dans la JSP lorsque l'on souhaite afficher le titre du livre et les noms de tous les auteurs.

Question 8 (3 points)  
Expliquez en moins de 10 lignes ce qu'apportent (par rapport à une solution à base de servlet et JSP) des frameworks comme AngularJS ou JQuery.

Les frameworks comme AngularJS et JQuery permettent d'améliorer l'expérience utilisateur en offrant des fonctionnalités avancées telles que des mises à jour dynamiques de l'interface utilisateur sans recharge de la page, des effets d'animation, des interactions avec des API distantes, une gestion simplifiée des événements et de nombreuses autres fonctionnalités qui ne sont pas disponibles dans une solution basée uniquement sur Servlets et JSP. Ces frameworks facilitent également le développement et la maintenance de l'application grâce à une architecture plus modulaire et une meilleure séparation des préoccupations.

Cependant, le problème avec cette approche est que la classe Livre contient une liste d'objets de type Auteur, qui ne sont pas directement accessibles dans la JSP. Il faut donc trouver un moyen de les rendre accessibles.

- **Code métier:** (souvent appelée back-end) correspond au Modèle aussi appelé code métier, car cela implante la logique de l'application indépendamment du fait que ce soit une application Web ou pas. Cette partie backend est divisée entre Facade et Data. La Facade correspond au code de l'application, qui fournit une interface utilisée par le Contrôleur. La partie Data correspond à la gestion des données que l'on conserve. Cette partie Data repose en général sur une base de données.
- **Cgi** est un protocole qui permet à un serveur web de communiquer avec des programmes exécutables sur un serveur. En d'autres termes, CGI est une interface standard qui permet à un serveur web de générer des pages web dynamiques en exécutant des scripts côté serveur. Les scripts CGI sont écrits dans différents langages de programmation tels que Perl, Python, Ruby, C++, etc. L'utilité de CGI est qu'il permet de créer des pages web dynamiques et de fournir des fonctionnalités interactives telles que les formulaires en ligne, les calculs dynamiques, les moteurs de recherche, etc. Cependant, CGI est considéré comme relativement lent et gourmand en ressources, ce qui a conduit à l'émergence de technologies plus modernes telles que PHP et Java Servlets.
- **PHP :** PHP est un langage de script côté serveur qui est utilisé pour développer des applications web dynamiques. PHP est une technologie open-source et gratuite qui est très populaire pour développer des sites web, des applications e-commerce, des forums, des blogs, etc. PHP est facile à apprendre et à utiliser, et il dispose d'une grande communauté de développeurs qui fournissent des bibliothèques de fonctions et de modules supplémentaires. L'utilité de PHP est qu'il permet de créer des pages web dynamiques rapidement et facilement, avec une grande variété de fonctionnalités telles que la manipulation des bases de données, les sessions d'utilisateurs, les formulaires, la gestion des fichiers, etc.
- **Applet :** Un Applet est une petite application logicielle qui s'exécute dans un navigateur web. Les applets étaient populaires dans les années 1990 et au début des années 2000, mais leur utilisation a diminué depuis l'introduction de HTML5 et de JavaScript. L'utilisation des applets était qu'il permettait de fournir des fonctionnalités riches et interactives aux utilisateurs sans nécessiter de plugins tiers. Les applets étaient écrits en Java et pouvaient interagir avec le navigateur pour afficher des graphiques, jouer des sons, etc. Cependant, les applets ont souffert de problèmes de sécurité et de performances, et ils sont désormais remplacés par des technologies plus modernes telles que HTML5, CSS3 et JavaScript.