Workshop
Persistance avec JPA

Création d'une couche de persistance avec l'API JPA

Objectif :
Mettre en place une couche de persistance pour la gestion du catalogue d'une librairie (en ligne ?)

Durée : 3h

Nous nous situons dans le cadre d'une « application » devant gérer le catalogue d'une librairie.

Ce document se décompose de la façon suivante : Après l'introduction, vous trouverez les consignes pour préparer votre environnement. Votre environnement prêt, il faudra lire le document *JPA_presentationSolution* qui présente l'architecture de l'application au travers de diagrammes commentés. Vous pourrez passer ensuite aux 6 exercices à réaliser pour finaliser une ébauche de gestion de catalogue.

1. PREPARATION DE L'ENVIRONNEMENT DE TRAVAIL

a) Création de la base de données

Dans le dossier ***bdd***, vous trouverez les ressources relatives à la base de données librairie (fichier librairie.sql, modèle entités-relations en jpg et au format MySQL Workbench).
Le fichier librairie.sql permet de créer non seulement la structure de la base mais aussi d'insérer un jeu de données.

- *Comment créer la base **librairie** en vous aidant de l'outil MySQL WorkBench :*

Ouvrez la connexion à votre SGBD
Dans le menu *File* cliquez sur *Open SQL Script…*
Sélectionnez le fichier librairie.sql
Cliquez sur le bouton *Exécuter* (éclair jaune)

b) Ouverture du squelette de l'application Maven

Le dossier ***bookStore***, situé dans le dossier *appli*, est le dossier racine du squelette de l'application bookStore.
Ouvrez le projet BookStore depuis NetBeans :
Open Project…> Sélectionnez bookStore > Puis cliquez sur *Open Project*

Ce squelette Maven bookStore-1.0-SNAPSHOT d'application web Java EE contient :

- Des vues JSF stockées sous Web Pages.
- Le paquetage *com.bookstore.model* contenant les beans CDI permettant de lier les vues (et donc l'utilisateur) avec la logique métier implémentée par des session beans. Ces beans représentent donc le point d'entrée dans le traitement applicatif.
- Le paquetage *com.bookstore.bll.catalogmgmt* : le session bean CatalogManager exposé au travers de l'interface locale CatalogManagerService est implémenté. Il n'y a donc pas besoin d'ajouter de code dans sa classe. Ce session bean fait office de façade de services.

Les méthodes des services (classes PublisherManagerServiceBean, BookManagerServiceBean et CatalogManagerServiceBean) ne sont pas implémentées. C'est dans ces beans que vous devrez implémenter les opérations de persistance.

- Le paquetage *com.bookstore.business.persistence.catalog* contient les classes du domaine. Aucune annotation JPA n'est présente pour les classes du domaine. Ceux ne sont donc pas pour l'instant des entités au sens JPA.
- Le fichier persistence.xml qui doit être configuré.
- Des tests JUnit (paquetage *com.bookstore.test* sous le dossier Test Packages) pour valider la progression dans le workshop. Il y a des tests qui permettent la validation de la modélisation JPA et des tests pour la validation des opérations de persistance. Payara n'a pas besoin d'être démarré pour exécuter les tests.

En résumé vous allez devoir configurer les parties relatives au mapping O/R JPA et coder les opérations de persistance.

c) Création de la source de données dans Payara

Vous devez créer un pool de connexions nommé **libraryPool** associé à une source de données nommée **jdbc/libraryDB.**
Pour la création d'une source de données et du pool de connexions associées référez-vous au document *DataSourceDansPayara*. On considère que le SGBD est MySQL 8 et le connecteur en version 8 aussi.
Voici quelques rappels pour configurer correctement votre source de données :

Vérifiez que vous avez placé le connecteur MySQL dans le dossier <*racine_install_payara*>\glassfish\lib.
Nom de la source de données **jdbc/libraryDB**
Nom du pool de connexions : **libraryPool.**
Type de ressource : **javax .sql.DataSource.**
Type de SGBD (database vendor) est **MySQL 8** (si vous utilisez l'interface web Payara).
Nom de la classe implémentant DataSource : **com.mysql.cj.jdbc.MysqlDataSource.**
URL de connexion : **jdbc:mysql://localhost:3306/librairie**
Si vous utilisez NetBeans pour créer le pool de connexion alors l'URL contiendra la propriété zeroDateTimeBehavior=convertToNull que vous pouvez supprimer.

Propriétés à ajouter pour configurer le pool de connexion

- User : compte mySQL valide (ex :**root**)
- Password : mdp du compte valide (ex : **dbpwd**)
- useSSL : **false**
- serverTimezone : **CET**

Si vous utilisez NetBeans n'oubliez pas que le fichier glassfish-resources.xml doit être sous setup et que l'extension .xml doit être explicite.

Voici un extrait du contenu de glassfish-resources.xml si vous avez utilisé NetBeans pour créer la source de données en suivant les rappels ci-dessus :

```
<jdbc-resource enabled="true" jndi-name="jdbc/libraryDB" object-type="user" pool-name="libraryPool">
<description/>
  </jdbc-resource>
  <jdbc-connection-pool … datasource-classname="com.mysql.cj.jdbc.MysqlDataSource" … name="libraryPool" … res-type="javax.sql.DataSource" …>
<property name="URL" value="jdbc:mysql://localhost:3306/librairie?zeroDateTimeBehavior=convertToNull"/>
    <property name="User" value="root"/>
    <property name="Password" value="dbpwd"/>
    <property name="useSSL" value="false"/>
    <property name="serverTimezone" value="CET"/>
  </jdbc-connection-pool>
```

Dans le pom.xml (sous le nœud NetBeans Project Files) les tests sont « activés » :

```
<properties>
        <endorsed.dir>${project.build.directory}/endorsed</endorsed.dir>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.test.skip>false</maven.test.skip>
  </properties>
```

Le déploiement de l'application échouera du fait de l'échec des tests.
Si vous avez configuré votre source de données via NetBeans, il faut cependant exécuter l'application pour que le pool de connexions et la source de données soient déployés sur le serveur. Remplacez donc false par **true** pour esquiver l'étape de test. Le déploiement effectué, réactivez les tests.

d) Création de l'unité de persistance bsPU

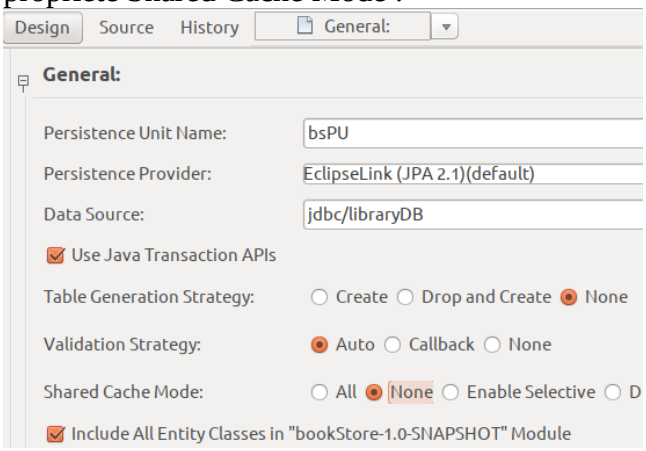
Pour information, le fichier persistence.xml est localisé dans le projet Maven sous Other Sources/src/main/resources/META-INF.
Pour l'instant le fichier de persistence.xml contient uniquement l'unité de persistance testPU utilisée pour faire des tests unitaires dans un environnement Java Standard. Vous allez créer une nouvelle unité de persistance bsPU qui utilisera la source de données jdbc/libraryDB configurée et gérée dans Payara.

- *Création de l'unité de persistance **bsPU** sous NetBeans:*

Positionnez-vous sur votre projet.
Faites un clic <droite> pour étendre *new* > *Persistence Unit…*
Si vous ne trouvez pas Persistence Unit…dans le menu <droite> NetBeans, cliquez sur Other… > Sélectionnez la catégorie Persistence > Choisissez Persistence Unit dans la liste de droite.
Nommez l'unité de persistance **bsPU**
Vérifiez que le fournisseur de persistance est bien EclipseLink
Sélectionnez la source de données **jdbc/libraryDB**
Laissez cochée la case Use *Java Transaction APIs*
Positionnez la stratégie de génération de tables sur *None*

Validez la création (cliquez sur *finish*). L'unité de persistence est ajoutée au fichier persistence.xml.

Désactivez le cache L2 partagé. Vous pouvez le faire depuis l'onglet Design du fichier persistence.xml. Pour cela déroulez le nœud General sous bsPU et cochez None pour la propriété Shared Cache Mode :



Désormais persistence.xml contient une unité de persistance de « test » et une unité de persistance « opérationnelle »
Si vous observez votre fichier persistence.xml en mode Source, Vous devriez avoir la configuration suivante pour bsPU :

```
<persistence-unit name="bsPU" transaction-type="JTA">
   <jta-data-source>jdbc/libraryDB</jta-data-source>
   <exclude-unlisted-classes>false</exclude-unlisted-classes>
   <shared-cache-mode>NONE</shared-cache-mode>
   <properties/>
 </persistence-unit>
```

Si vous voulez suivre le déroulement de l'exécution JPA via les logs, ajoutez au fichier persistence.xml la propriété :
eclipselink.logging.level Avec la valeur FINEST ou FINE ou FINER selon le niveau de journalisation (d'information) que vous désirez.
Pour cela, sélectionnez en mode Design le nœud Properties de l'unité de persistance bsPU > cliquez sur Add… > sélectionnez dans la liste déroulante eclipselink.logging.level > Sélectionnez la valeur (FINEST par exemple) > cliquez sur OK.
Vous devriez avoir l'élément suivant dans persistence.xml :

```
<properties>
    <property name="eclipselink.logging.level" value="FINEST"/>
</properties>
```

L'unité de persistance testPU utilisée dans les tests unitaires est déjà configurée dans le projet. testPU utilise pour la connexion à la base le compte *root* associé au mot de passe *dbpwd*. Si le mot de passe de votre compte *root* est différent, il faut que vous remplaciez *dbpwd* par **votre mot de passe**.
Extrait de testPU :

```
<!--unité de persistance pour les tests d'opérations de persistance-->
 <persistence-unit name="testPU" transaction-type="RESOURCE_LOCAL">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
```

```
…
<properties>
  <property name="javax.persistence.jdbc.url"
            value=" jdbc:mysql://localhost:3306/librairie?serverTimezone=CET"/>
  <property name="javax.persistence.jdbc.password" value="votre mot de passe root"/>
  <property name="javax.persistence.jdbc.driver" value=" com.mysql.cj.jdbc.Driver"/>
  <property name="javax.persistence.jdbc.user" value="root"/>
  <property name="eclipselink.canonicalmodel.subpackage" value="test"/>
</properties>
</persistence-unit>
```

Maintenant que votre environnement est prêt, prenez le temps d'étudier le document présentant la solution attendue (***JPA_presentationSolution***).

EXERCICES

1. **Avant de coder - Dresser la liste des annotations JPA à utiliser**

   Comparez le diagramme de classe représentant le modèle objet et le modèle Entités-Relations MySQL, et dresser une première liste des annotations JPA à utiliser.
   Il est conseillé de travailler sur ce point en binôme ou en trinôme. Ce travail ne doit pas durer plus de 15 minutes.
   Vous pouvez par exemple utiliser le tableau suivant :

Annotations sur les classes	
Annotations pour configurer les champs id	
Annotation pour mapper un champ avec une colonne	
Annotations pour modéliser les relations	
Annotations pour mapper les relations objets avec les associations du MCD	
Annotations pour les objets embarqués	

Pour chaque exercice il faut que vous preniez le temps de vous arrêter sur l'implémentation des méthodes du session bean CatalogManager afin d'appréhender dans sa globalité le fonctionnement de JPA.

2. **Exercice 1 - Implémentation de l'entité Publisher**

   Pour valider la mise en place du modèle JPA et sa mise en correspondance avec la base librairie, vous allez vous appuyer sur des tests écrits avec le framework JUnit. Les tests de la classe **JPAModelingTest** testent la validité de la configuration de votre mapping O/R JPA, c'est-à-dire qu'ils valident si vous annotez bien les classes du domaine et leurs attributs. Les de test de **CrudOperationTest** sont des tests complémentaires pour s'assurer de la validité votre implémentation JPA. C'est donc une double vérification.

Pour que les tests soit fonctionnels annotés les **variables d'instances (champs)** des classes et non le getter.

Nous allons détailler cette étape au pas à pas. Les prochaines étapes seront moins détaillées.

2.1. Réalisez une première ébauche

Vous allez annoter la classe Publisher pour que celle-ci soit configurée en tant qu'entité JPA mappée avec la table *editeurs*
Concernant les attributs de la classe, vous vous limitez au mapping de :

- L'attribut *name* de *Publisher* avec la colonne *RAISON_SOCIALE* de la table *editeurs*.
- L'attribut *id* de *Publisher* avec la colonne *ID_EDITEUR* de la table *editeurs*.

Vous ne considérez donc pas les attributs *address et books*

Remarque : Concernant la génération automatique des clés primaires on utilise la contrainte d'identité. La clé primaire est donc générée automatiquement lors des insertions de lignes dans la base. Ce n'est pas le mode par défaut de MySQL qui utilise la génération par séquence. Il faut donc configurer les entités pour qu'elles utilisent l'identité comme mode de génération de clé primaire.
Il faut donc utiliser la stratégie *GenerationType.IDENTITY*

Lancez les tests de JPAModelingTest. Vous pouvez continuer, si le test **checkPublisherV1** passe avec succès : votre modélisation est valide. Si le test échoue, aidez-vous du message d'erreur affiché.
Pour lancer le test, cliquez du droite sur le fichier de test (ici JPAModelingTest.java) et sélectionnez l'action Test File. Vous pouvez aussi lancer l'ensemble des tests en cliquant du droite sur le projet bookstore-1.0-SNAPSHOT puis sur Test.

2.2. Modélisez la classe embarquée

Annotez la classe Address ainsi que ses attributs pour que celle-ci soit considérée comme une classe embarquée.
En dehors de *Address.zp* qui correspond à *editeurs.CODE_POSTAL*, les noms des autres attributs de Address permettent de déduire facilement avec quelles colonnes ils doivent être mappés.
Annotez l'attribut (variable d'instance) address pour que JPA la considère comme une référence à un objet embarqué.
Annotez le champ books (private List<Book> books) de l'entité Publisher avec @javax.persistence.Transient. Cette annotation indique à JPA que le champ books n'est pas à prendre en compte dans les opérations CRUD. Dans notre cas cela évite qu'une erreur soit générée lors du lancement du test CrupOperationTest.**checkPublisherRetrieving** (cf. juste au-dessous). Sans cette annotation @Transient, le fournisseur de persistance génère une requête de sélection incluant le champ books. Or ce champ n'existe-pas en base.

Lancez les tests de JPAModelingTest puis ceux de CrupOperationTest. Vous pouvez continuer, si les tests JPAModelingTest**.checkPublisherV2** et CrupOperationTest.**checkPublisherRetrieving** passent avec succès (en plus du test checkPublisherV1) : votre modélisation est valide. Vous progressez. Si l'un des tests échoue, aidez-vous du message d'erreur affiché.

Remarque : checkPublisherRetrieving n'est qu'un test de contrôle pour s'assurer que l'entité JPA Publisher et la classe embarquée Address sont correctement annotées. Ce test recherche l'entité d'id 40 et teste que le code postal récupéré est le bon. Tous les tests de CrudOperationTest sont des tests similaires de recherche dont le but est un double contrôle de votre modélisation. Les tests CrudOperationTest utilisent testPU. Si l'exécution de ces tests se solde par des erreurs (à ne pas confondre avec un échec ou un succès), vérifiez que le mot de passe dans testPU est bien le mot de passe de votre compte root.

2.3. Injectez les entity managers

Dans les classes des 3 session beans CategoryManagerServiceBean, BookManagerServiceBean et PublisherManagerServiceBean, positionnez l'annotation nécessaire pour injecter un entity manager dans la variable d'instance de type EntityManager.

Lancez le test EntityManagerInjectionTest. Si le test réussi, votre injection d'entity manager est correctement configurée.

2.4. Implémentez la persistance d'un éditeur

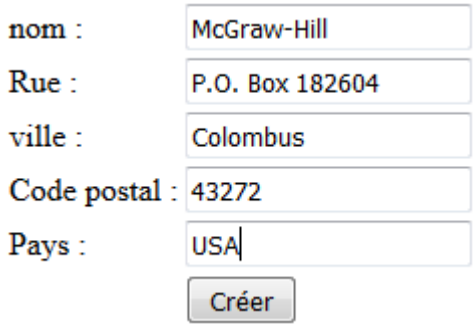
A partir de cet exercice, vous allez entre-autre exécuter l'application pour effectuer des tests fonctionnels.
En plus de l'examen de la méthode CatalogManager.createPublisher, regardez la source de la vue JSF publisherCreation.xhtml et le code du bean CDI PublisherBean impliqués pour mieux appréhender l'implémentation de la fonctionnalité et accroitre votre connaissance du framework JSF.

Dans la méthode *PublisherManagerServiceBean.savePublisher*, implémentez la persistance de l'entité Publisher passée en argument de la méthode.

Modifiez dans pom.xml la propriété *maven.test.skip* sur **true** pour que les tests soient esquivés lors de l'exécution de l'application. Sans ça l'exécution échouera du fait que tous les tests ne passent pas.

```
<maven.test.skip>true</maven.test.skip>
```

Lancez le projet bookStore : **Run** sur le module Maven *bookStore*.
Cliquez sur Créer un Editeur > Renseignez les champs de l'éditeur > cliquez sur Créer :



Si l'insertion a réussi, la vue publisher est affichée au client :

id : 94
nom : McGraw-Hill
Rue : P.O. Box 182604
ville : Colombus
Code postal : 43272
Pays : USA
Retour Accueil

Vérifiez aussi dans la base que l'entité créée est bien sauvegardée en base.

Si vous ne voyez aucun résultat, lancez un build du projet avant de le réexécuter. Par défaut, lorsque vous enregistrez une modification dans le projet, l'application est redéployée sur le serveur, si celui-ci est démarré. Par contre si le serveur est éteint, l'enregistrement d'une modification n'entraine bien évidemment pas de redéploiement. Si après avoir modifié un projet, vous lancez (Run) ce projet avec un serveur éteint, NetBeans démarre le serveur. Si le serveur héberge déjà une version de votre application, c'est cette version hébergée qui entre en service et non la version modifiée. D'où le build préalable parfois nécessaire.

3. Exercice 2 – Implémentez la relation un-plusieurs bidirectionnelle entre Publisher et Book

Avant de passer à l'implémentation JPA de la relation, vous allez finaliser la configuration basique des classes du domaine en entité.

3.1. Configurez les entités Category et Book

Annotez les classes Category et Book pour le transformer en entités. Pour l'instant vous ne tenez toujours pas compte des propriétés modélisant les relations entre entité (vous ne modélisez pas les relations JPA et leur mapping avec la base).
Autrement-dit vous faites le même travail que celui effectué pour l'entité Publisher dans les parties 1.1 et 1.2
La seule nouveauté est le mapping de *Book.date* de type java.util.Date avec *livres.DATE_PARUTION*.
Le type temporel de mapping est **javax.persistence.TemporalType.DATE**.
Notez que le champ Category.description et la colonne *categories.DESCRIPTION* ont le même nom.
Réactivez l'exécution des tests :

```
maven.test.skip>false</maven.test.skip>
```

Lancez les tests de JPAModelingTest. Votre modélisation est correcte si les tests checkBookV1 et checkCategoryV1 (en plus de checkPublisherV1&V2) se terminent avec succès.

3.2. Modélisez la relation un-plusieurs entre Publisher et Book

Supprimez l'annotation @Transient décorant la variable Publisher.books.
Positionnez les annotations pour modéliser la relation entre l'éditeur et ses livres sur les champs adéquats de Publisher et Book. Ne vous occupez pas pour l'instant du mapping entre cette relation orientée objet et la relation entre les tables sous-jacentes.
Rappelons que le propriétaire de la relation est l'entité côté « Plusieurs » donc Book.

L'annotation configurant la relation (plusieurs) Book - (un) Publisher doit spécifier *CascadeType.MERGE*. Ainsi si le livre est rattaché au contexte de persistance, l'éditeur associé sera lui aussi automatiquement rattaché au contexte de persistance.

Exécutez les tests JPAModelingTest. Votre modélisation de la relation entre Publisher et correctement implémentée si le test **checkBidirectionalOneToManyBetweenPubBook** réussit (bien évidemment les tests amonts devraient aussi avoir réussi).

3.3. Mappez la relation un-plusieurs entre Publisher et Book avec l'association sous-jacente

Placez l'annotation pour mettre en correspondance la relation entre Publisher et Book avec l'association entre les tables sous-jacentes à ces 2 entités. Vous devez donc mapper la relation avec la colonne *clé étrangère* (colonne de jointure) *ID_EDITEUR* de la table *livres*.

Lancez les tests de JPAModelingTest & de CrudOperationTest. Vous avez complété la configuration de la relation bidirectionnelle un-plusieurs entre Publisher et Book si les tests JPAModelingTest.**checkJpaBookPubRelation**
et CrudOperationTest.**checkPubFromBookRetrieving** s'exécutent avec succès. Bien évidemment tous les tests préalables devraient aussi réussir (on ne le précisera plus à l'avenir).

3.4. Persistez la relation entre un livre et un éditeur

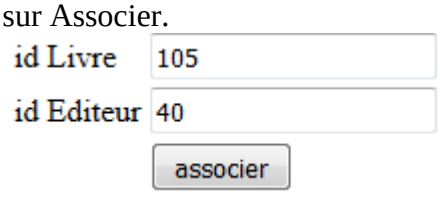
Implémentez les méthodes *PublisherManagerServiceBean.findPublisherById(Long publisherId)* et *BookManagerServiceBean.findBookById(Long bookId)* pour qu'elles retournent respectivement les entités Publisher et Book correspondant aux id passés en paramètre.

Implémentez la méthode *BookManagerServiceBean.updateBook(Book book)* pour que le livre passé en paramètre soit rattaché au contexte de persistance.
Rappelez-vous que les modifications d'entités sont répercutées sur la base si ces entités sont managées (attachées au contexte de persistance).
La modification de l'éditeur sera répercutée en base car vous avez spécifié au préalable que les opérations « MERGE » sur un livre devaient être cascadées à l'éditeur associée.

Modifiez dans pom.xml la propriété *maven.test.skip* sur **true** pour que les tests soient esquivés.

```
<maven.test.skip>true</maven.test.skip>
```

Lancez le projet > cliquez sur Associer un livre à un Editeur > Saisissez une identité de livre et d'éditeur valides. Par exemple, 105 (Livre EJB in Action) et 40 (Editeur Manning) > cliquez sur Associer.



Si l'opération a réussi, l'écran suivant s'affiche :

EJB3 in action

plonger dans le monde EJB3

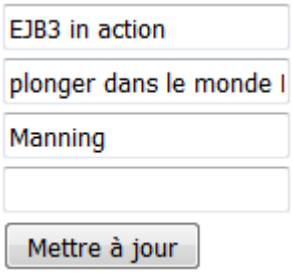
Manning

Editer

Retour Accueil

Vérifier dans la base que la relation est bien persistée.

Pour tester que les opérations de mise à jour sont bien répercutées en cascade de Book vers Publisher cliquez sur Editer. Le dernier champ de saisie correspond à la rue de l'adresse de l'éditeur (cf. vue bookEdition.xhtml). Si vous avez associé *EJB in Action* avec *Manning*, ce champ de saisie est vide car la rue pour Manning n'est pas renseignée (cf. table *editeurs*) :



Modifiez/renseignez le dernier champ avec un nom de rue (ex : Sound View Court 3B) > cliquez sur Mettre à jour.
Si la mise à jour en cascade a réussi, vous devriez avoir l'écran suivant :

EJB3 in action

plonger dans le monde EJB3

Manning

Sound View Court 3B

Editer

Retour Accueil

Vérifiez en base que l'adresse de l'éditeur est bien mise à jour.

4. Exercice 3 – Implémentez la relation plusieurs-plusieurs bidirectionnelle entre Category et Book

Maintenant que vous êtes rodés, nous allons accélérer un peu.

4.1. Modéliser et mapper la relation entre Category et Book

Annotez la variable d'instance Category.parentCategory avec @Transient pour qu'elle soit ignorée par JPA lors des opérations de persistance.
Annotez les variables d'instance **Category.books** et **Book.categories** pour configurer la relation many to many. Category sera l'entité propriétaire de la relation. Par conséquent Category.books représente le côté propriétaire de la relation. Pour rappel, le mapping fait intervenir la table de jointure *appartient*. La table *appartient* a 2 colonnes clés étrangères *ID_CATEGORIE* & *ID_LIVRE* qui composent la contrainte de clé primaire.

Réactivez l'exécution des tests :

```
maven.test.skip>false</maven.test.skip>
```

Validez avec les tests JPAModelingTest**.checkJpaCatBookRelation** et CrudOperationTest**.checkNbOfBooksFromCategoryRetrieving**

4.2. Persistez un nouveau livre associé à une catégorie existante

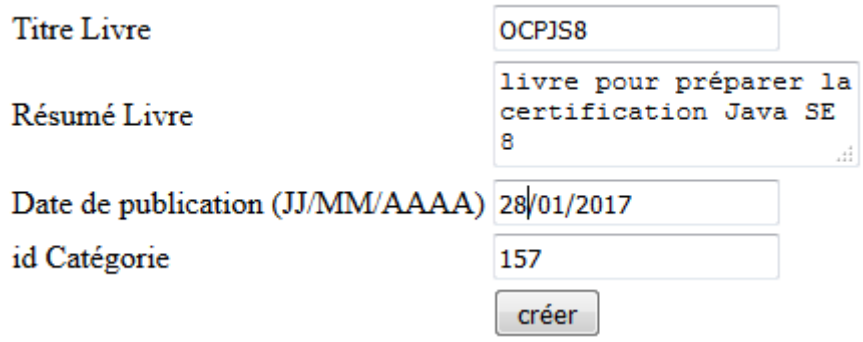
Il faut implémenter la récupération d'une catégorie en fonction d'une identité et persister le livre auquel on aura associé la catégorie.
Il vous faut donc implémenter les méthodes :
CategoryManagerServiceBean.findCategoryById & *BookManagerServiceBean.saveBook*.

Pensez à examiner la méthode *Book createBook(String title, String summary, Date date, Long categoryId)* de CatalogManagerServiceBean.

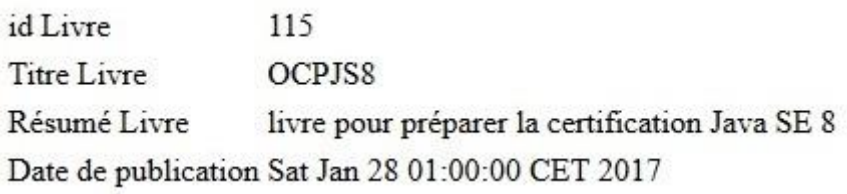
Modifiez dans pom.xml la propriété *maven.test.skip* sur **true** pour que les tests soient esquivés.

```
<maven.test.skip>true</maven.test.skip>
```

Dans la base, repérez une identité de catégorie valide. Par exemple 157 (Catégorie Java).
Lancez le projet > cliquez sur Créer un livre > Saisissez les données d'un nouveau livre en spécifiant l'identité d'une catégorie existant repérée précédemment > cliquez sur Créer

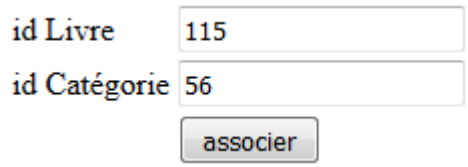


Si l'opération a réussi, la vue book.xhtml affichera les informations suivantes :

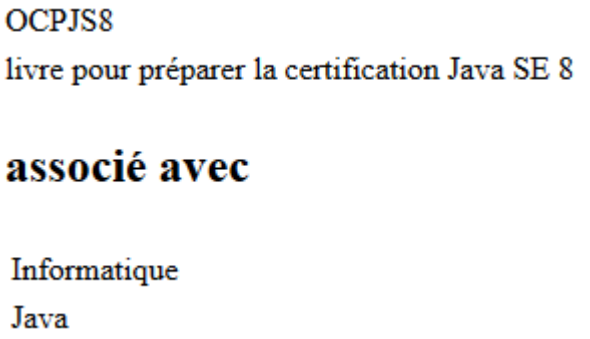


Vérifiez que le livre est bien persisté en base ainsi que sa relation avec la catégorie.

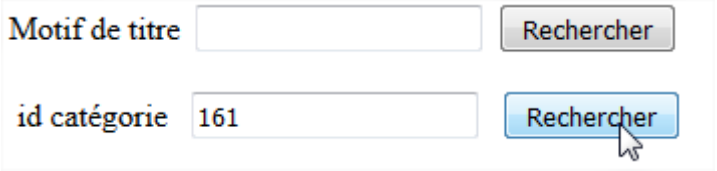
Vous pouvez aussi tester l'association d'un livre avec une catégorie via la fonctionnalité Associer un livre à une catégorie.
Dans le formulaire d'association (vue bookCategory.xhtml) saisissez une identité de livre valide et une identité de catégorie valide. Par exemple le livre 115 précédent et la catégorie Informatique 56. Cliquez sur Associer pour valider.



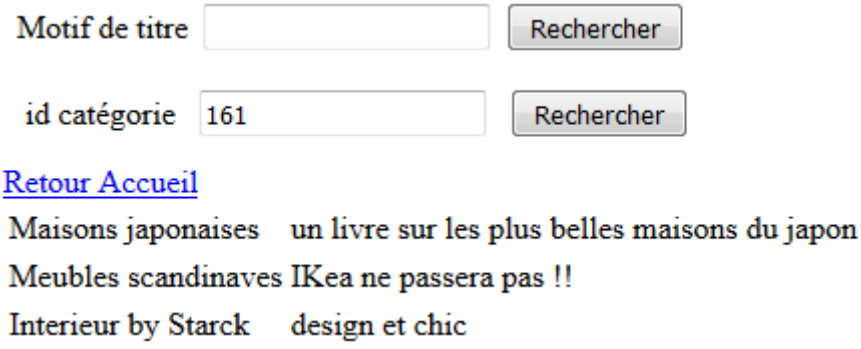
Si l'association a réussi JSF naviguera vers la vue bookCategories.xhtml dans laquelle sont affichées entre autre les catégories auxquelles le livre est associé :



Vous pouvez aussi tester l'association entre livres et catégories, en utilisant la fonctionnalité Rechercher des livres.
Dans le champ de saisie **id Catégorie**, saisissez un id de catégorie ayant une correspondance dans la table *categories* (par exemple, 161) > cliquez sur le bouton Rechercher associé :



La réponse va afficher la liste des livres associés à la catégorie sélectionnée, si la catégorie contient des livres bien évidemment :



5. Exercice 4 – Implémentez la relation réflexive unidirectionnelle Plusieurs-une Category.

Rappel : Les catégories peuvent avoir une catégorie parente. Une catégorie racine n'a évidemment pas de parent.

5.1. Annotez la relation entre une catégorie et sa catégorie parent.

Supprimez l'annotation @Transient décorant la variable Category.parentCategory.
Plusieurs catégories peuvent avoir la même catégorie parent.
Au niveau de la table categories cette association est modélisée par la colonne clé étangère *CAT_ID_CATEGORIE* qui référence la colonne clé primaire ID_CATEGORIE de cette même table.

Réactivez l'exécution des tests :

```
maven.test.skip>false</maven.test.skip>
```

Validez avec les tests JPAModelingTest.**checkCategoryReflexiveRelation** et CrudOperationTest.**checkParentCatFromChildRetrieving.**

Si tous les tests de JPAModelingTest et CrudOperationTest s'exécutent avec succès, vous avez finalisé votre mapping JPA du domaine.

5.2. Gérer les catégories

Implémentez la méthode *CategoryManagerServiceBean.saveCategory(Category category)* qui sauvegarde en base une nouvelle sous-catégorie et retourne son identité (id).
Or l'identité est générée lors de l'insertion dans la base. Par défaut, l'insertion en base a lieu lorsque la transaction active se termine avec succès. La transaction se termine lorsque la méthode *CatalogManagerServiceBean.createCategory* invoquée par le client (bean CategoryBean) a fini de s'exécuter. Ce type de comportement par défaut permet d'optimiser les ordres SQL générés et les Aller / Retours avec la base, et donc les performances de l'application.
Cependant ce comportement nous pose problème car il faut pouvoir insérer en base et rafraîchir l'entité avant la fin de la transaction pour pouvoir récupérer la valeur d'id générée.
La méthode saveCategory doit donc :
Persister l'entité Category passée en paramètre
Forcer la synchronisation avec la base (flush)
Rafraîchir l'entité préalablement persistée avec les données de la base pour retourner la valeur de l'identité générée (refresh).
Ces 3 étapes s'effectuent avec des méthodes définies dans EntityManager.

Remarque :
Sans les opérations *flush* et *merge*, l'identité générée est assignée à l'entité par le fournisseur de persistance lors du « commit » de la transaction (après insertion en base). Donc, si on avait retourné l'instance de Category au lieu de la valeur de l'identité (avant inscription en base), cette instance aurait embarqué la valeur d'identité générée. Cela signifie aussi que la valeur générée de l'identité est accessible dans les méthodes de cycle de vie PostPersist, l'évènement étant déclenché après insertion en base.

Modifiez dans pom.xml la propriété *maven.test.skip* sur **true** pour que les tests soient esquivés.

```
<maven.test.skip>true</maven.test.skip>
```

Lancez bookStore > Accédez à la fonctionnalité Créer une catégorie > Remplissez le formulaire en créant par exemple une catégorie *certifications* enfant de la catégorie Java (id = 157) > cliquez sur Créer :

Si la création de la catégorie ayant un parent a réussi la vue (categoryCreation.xhtml) est mise à jour avec message de création :

Vérifiez en base que la nouvelle catégorie associée à une catégorie parent a bien était persistée. Si vous ne saisissez pas d'id de catégorie parent, vous créez alors une catégorie racine.

6. **Exercice 5 – Supprimer un livre**

Il s'agit de supprimer de la base les informations relatives au livre SCJP, autrement-dit supprimer le livre SCJP. Implémentez la suppression de livre dans la méthode *BookManagerServiceBean.deleteBook(Book book)* où *book* est le livre à supprimer de la base.

Exécutez le projet Java EE > cliquez sur Supprimer un livre > Dans le formulaire de suppression (bookDeletion.xhtml) saisissez l'id valide d'un livre à supprimer. Par exemple OCJPJS8 (vérifiez l'id en base) > cliquez sur supprimer

L'action de suppression entraine juste la mise à jour de la vue avec l'affichage d'un message indiquant que l'entité a été supprimée :

Vérifiez que le livre et ses relations avec les catégories sont bien supprimés de la base.

7. **Exercice 6 – Utiliser l'API de requêtes.**

L'objectif est d'utiliser l'API de requête JPA pour effectuer des requêtes JPQL sur les entités.

Des méthodes de test sont proposées pour tester la syntaxe de vos requêtes. Pour valider la syntaxe via ces tests, il faut réactiver l'exécution des tests :

```
maven.test.skip>false</maven.test.skip>
```

Requête dynamique :

Dans la méthode *BookManagerServiceBean#findByCriteria(String pattern,)* créez une requête dynamique qui retourne l'ensemble des livres dont le titre contient un motif (pattern) donné (par exemple C#).
Malgré le nom de la méthode, il ne s'agit pas d'utiliser l'API Criteria JPA 2 qui n'est pas au programme.
Avant d'implémenter la méthode, vous pouvez tester la requête via le test JPQLTest.**checkRequestWithStringParameter**.

Requêtes nommées :

Sur la classe Category créez 2 requêtes nommées.

- Une requête récupérant les catégories racines (sans parent)
  Vous pouvez utiliser le test JPQLTest.**checkSimpleRequest** pour valider la syntaxe de la chaîne de requête.

- Une requête recherchant les catégories liées à une catégorie mère (par exemple les catégories filles de la catégorie Informatique (le code client de l'exercice 6 utilise l'id 56)
  Vous pouvez utiliser le test JPQLTest.**checkRequestWithCategoryParameter** pour valider la syntaxe de la chaîne de requête.

Les syntaxes validées, vous pouvez passer à l'implémentation des méthodes.

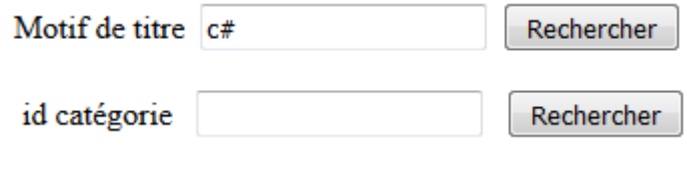
Implémentez *CategoryManagerServiceBean.getRootCategories* pour que la méthode utilise la première requête nommée pour retourner la liste des catégories racines.
Implémentez *CategoryManagerServiceBean.getchildrenCategories(Long parentId)* pour que la méthode utilise la seconde requête nommée pour retourner la liste des catégorie enfants en fonction de l'identité parent.
Dans vos implémentations essayez aussi d'utiliser les requêtes typées (TypedQuery) et pas seulement les requêtes non typées (Query) comme dans les tests.

Si vous avez utilisé les tests pour tester la syntaxe de vos requêtes et que ceux si ont réussi, tous vos tests devraient passés vous n'êtes donc plus obliger d'esquiver les tests pour lancer l'application. Si vous n'avez pas utilisé les tests de validation de la syntaxe JPQL, il faut par contre esquiver les tests.

Lancez le projet pour tester l'API de requêtes.

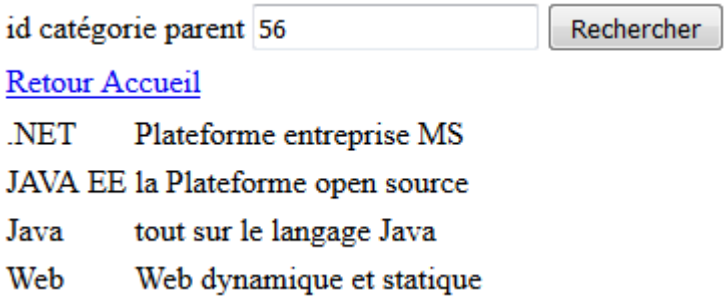
- Pour tester votre requête dynamique de sélection des livres contenant un motif dans le titre : Cliquez sur Rechercher des livres > dans le champ de saisie *Motif de titre*, entrez un motif à retrouver dans les titres de livres (par exemple C#) > cliquez sur Rechercher :

Vous devriez obtenir ce type de résultat :

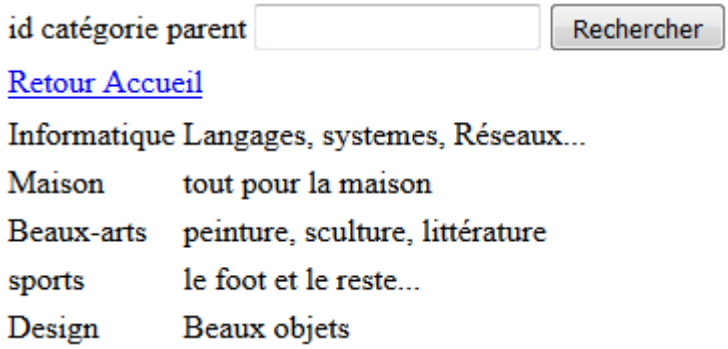


A noter que si vous ne saisissez pas un pattern (motif), l'ensemble des livres est retourné.

- Pour tester vos requêtes nommées retournant la liste des catégories racines ou des catégories enfants d'un parent donné :
  Cliquez sur Lister les catégories > Saisissez une identité de catégorie parent existant en base pour lister les catégories enfants ou n'entrez rien si vous voulez afficher les catégories racines (testez les 2 cas car il s'agit de 2 requêtes différentes) > Cliquez sur Rechercher.
  La capture d'écran ci-dessous présente la vue retournée si vous avez sélectionné la catégorie racine 56 (Informatique) :



La capture suivante correspond à la vue affichant les catégories racines. Une recherche sans id a été effectuée :



Vous avez fini !!!