Persistance des données en Java-JPA-EclipseLink

La 1ère partie présente JPA avec une utilisation assez rudimentaire. La 2ème partie traite d'une application organisée en couches. La 3ème partie approfondit l'utilisation de JPA.

►**L'utilisation de JPA nécessite Eclipse pour Java EE.**

Démarrer le serveur de BDD MySQL. Créer une base **lesinvites**.

Création d'une connexion MySQL dans Eclipse

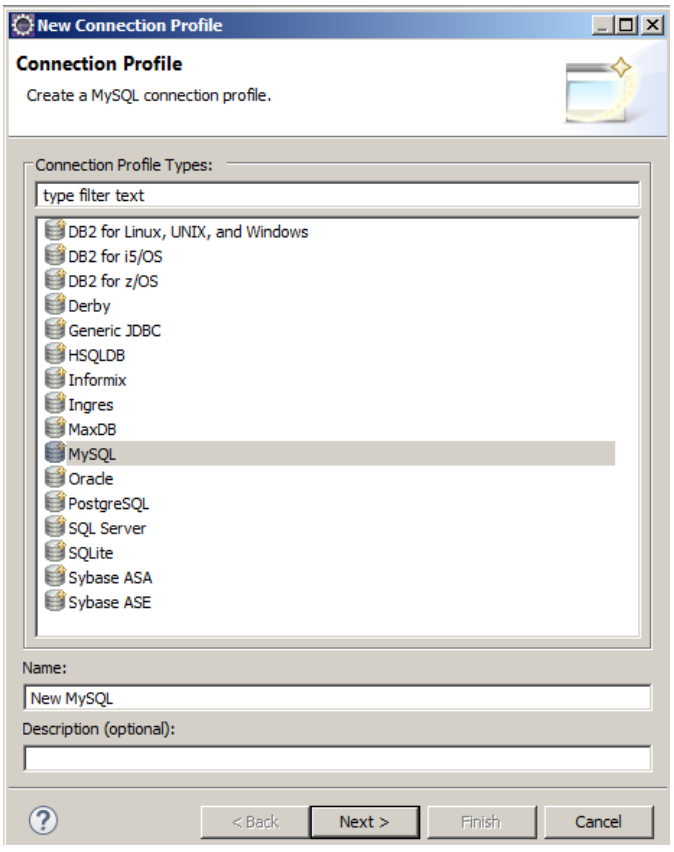
L'utilisation de JPA nécessite la création d'une connexion à une base de données, ici **MySQL**.

Télécharger le driver/connecteur MySQL à partir du lien **http://dev.mysql.com/downloads/connector/j/**. Attention: faire une recherche si le lien a changé.
Installer le driver.

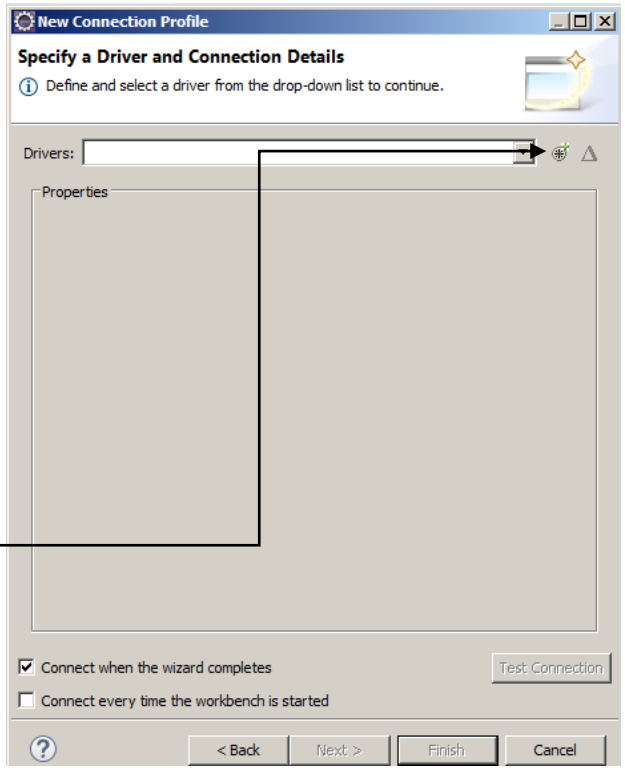
Le fichier jar **mysql-connector-java-5.1.34-bin.jar** est normalement installé dans le dossier **C:\Program Files (x86)\MySQL\MySQL Connector J**.

Ouvrir dans Eclipse la **Perspective JPA** : **Window/Open Perspective/JPA**.
On doit voir la vue **Data Source Explorer** en bas à gauche.

- Ouvrir la vue **Data Source Explorer.**
- Sélectionner **Database Connections** Bouton droit de la souris.
- **New**
- Sélectionner **MySQL** dans la liste proposée.
- Garder le nom par défaut New MySQL (on peut le changer).
- **Next**

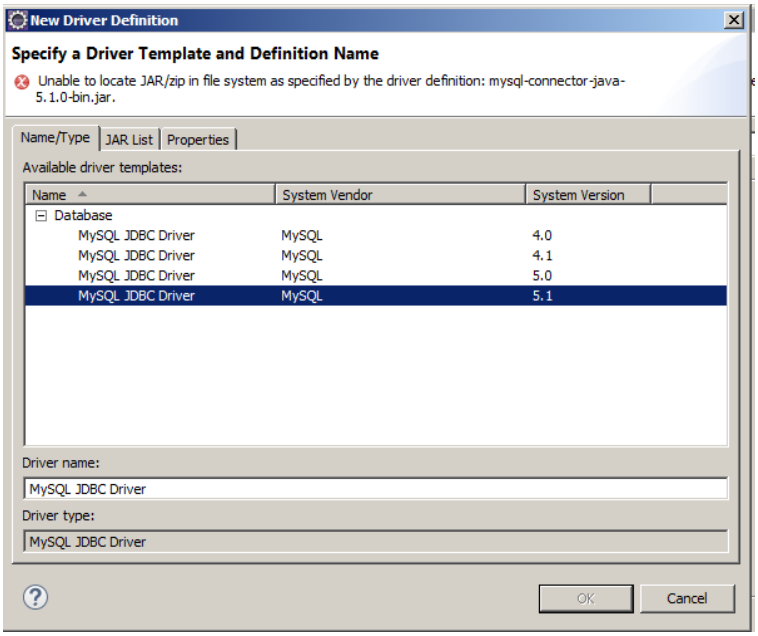


La fenêtre ci-contre est affichée.



Cliquer sur le bouton **New Driver Definition**

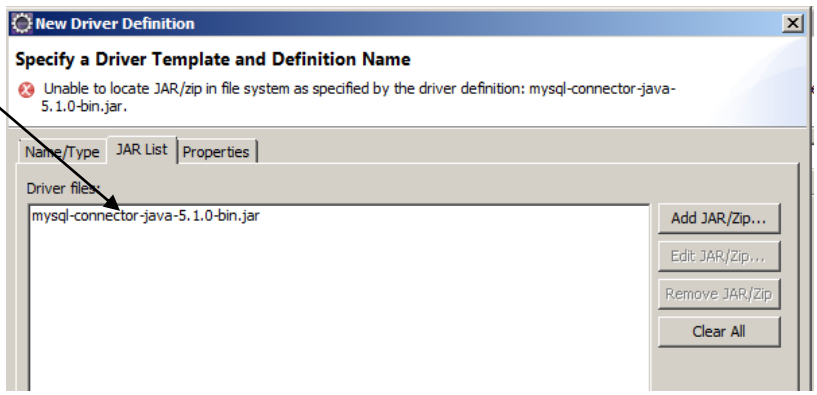
La fenêtre ci-contre est affichée.



Sélectionner **MYSQL JDBC Driver version 5.4**.

Sélectionner l'onglet **JAR List**.

La fenêtre suivante est affichée.



Sélectionner le fichier **mysql-connector-java-5.1.0-bin.jar**.

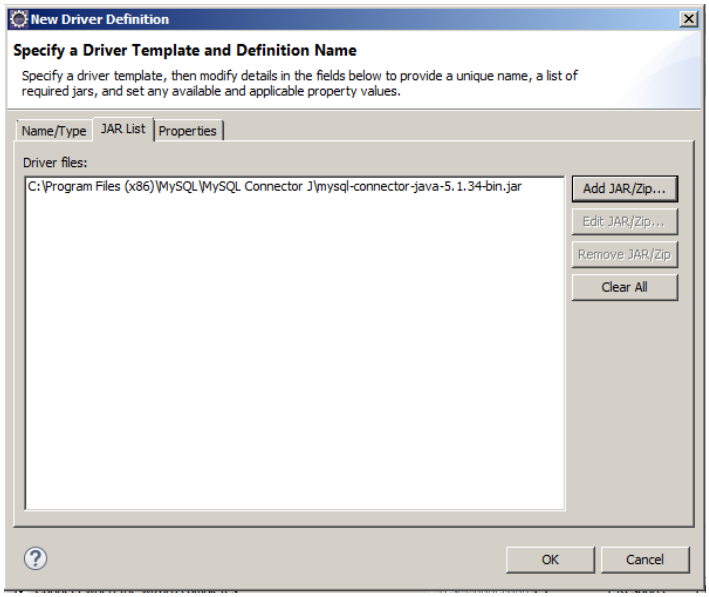
Cliquer sur Remove JAR/Zip... pour le supprimer.

Cliquer sur Add JAR/Zip...

Rechercher le fichier **mysql-connector-java-5.1.34-bin.jar** dans l'arborescence (normalement dans le dossier **C:\Program Files (x86)\MySQL\MySQL Connector J**). Puis Ouvrir.

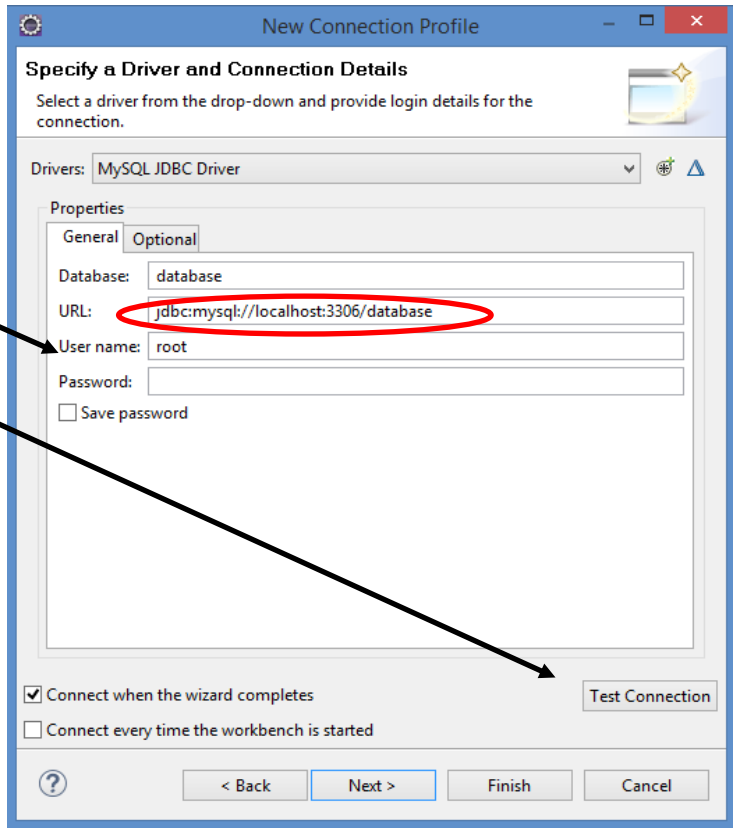
La fenêtre ci-contre est maintenant affichée.

Valider sur OK.

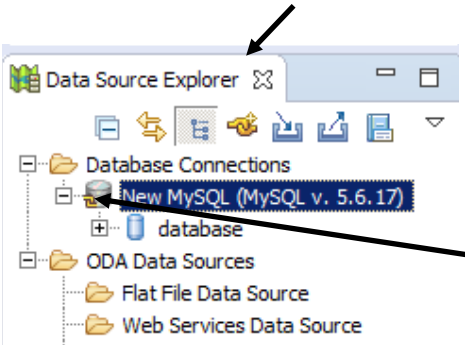


Il faut maintenant compléter les paramètres de la connexion à la base.

- Changer le nom de la base dans URL : jdbc:mysql://localhost:3306/**lesinvites**
- Compléter **User name** et **Password**
- Cocher **Save password**
- Faire un test avec **Test Connection**
- **Finish**



La connexion créée doit apparaître dans la vue **DataSourceExplorer.**



Le signe style "poignée de main" indique une connexion "connectée".

Attention: MySQL utilise le N° de port **3306** habituellement (par défaut), il est possible que votre installation utilise un autre N° de port, remplacer alors dans le champ URL 3306 par le N° de port utilisé par votre base.

Présentation de JPA et d'EclipseLink

JPA 2 (Java Persistent API) est un ensemble de spécifications (classes, interfaces) qui traitent de la persistance des objets dans une base de données relationnelle en utilisant le langage Java.

EclipseLink implémente les spécifications **JPA 2**. **EclipseLink** met en correspondance les classes et objets Java avec les tables et enregistrements d'une base de données: c'est la correspondance -Relationnel-Objet- (**ORM** : Object Relationnal Mapping).
On peut illustrer l'ORM de la manière suivante:

classe ⟷ table
objet ⟷ enregistrement

JPA fournit également des classes pour effectuer les transactions avec la base :

- les opérations **CRUD** : Create, Read, Update et Delete,
- toutes sortes de requêtes SQL.

On utilise le framework **EclipseLink** comme implémentation des spécifications **JPA**. On dit que **EclipseLink** est un fournisseur de persistance. **EclipseLink** implémente les spécifications Java Persistence API (JPA), Java Architecture for XML Binding (JAXB), Java Connector Architecture (JCA) et Service Data Objects (SDO).

Une **entité JPA** est une classe Java qui correspond à une table de la base sélectionnée.
Une instance de l'entité JPA (un objet Java) correspond à un enregistrement de la table (une ligne).
L'état persistant d'un objet est constitué par les valeurs de ses attributs.

Voir **https://docs.oracle.com/javaee/7/tutorial/partpersist.htm chapitre 37**.

1ère partie

Création du projet JPA

- **File/New/JPA Project**
- Donner un nom
- Sélectionner **jre** dans **Target Runtime**
- **Next**

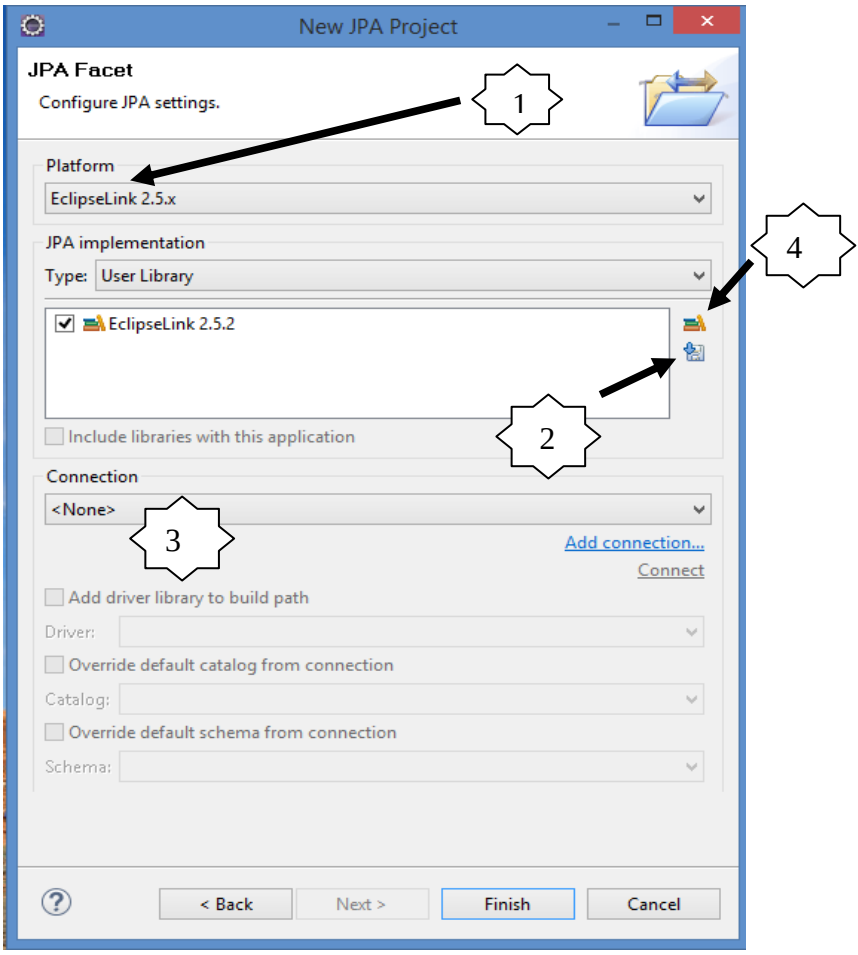
- **Next**

- Si **EclipseLink** est installé, on voit la fenêtre ci-contre.

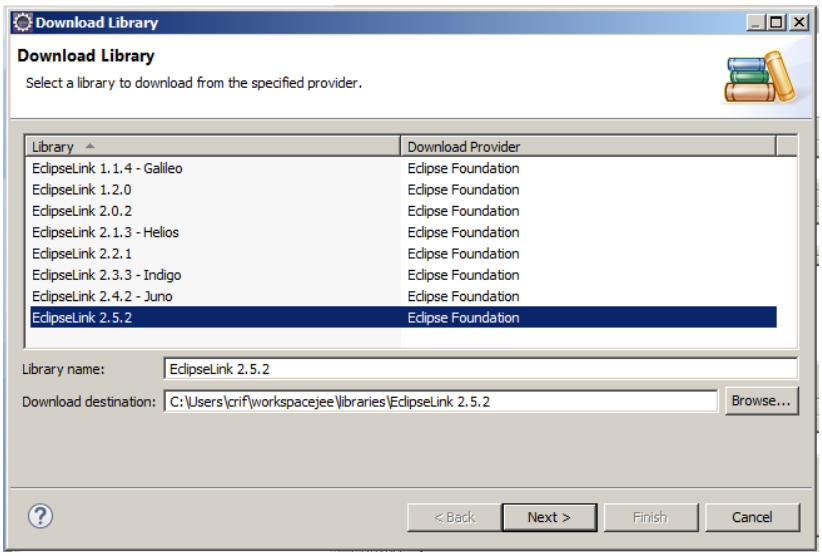
- Sinon cliquer sur **Platform** ①.

  - Sélectionner **EclipeLink 2.5.x**.

  - Cliquer sur l'icône **Disquette** ②.

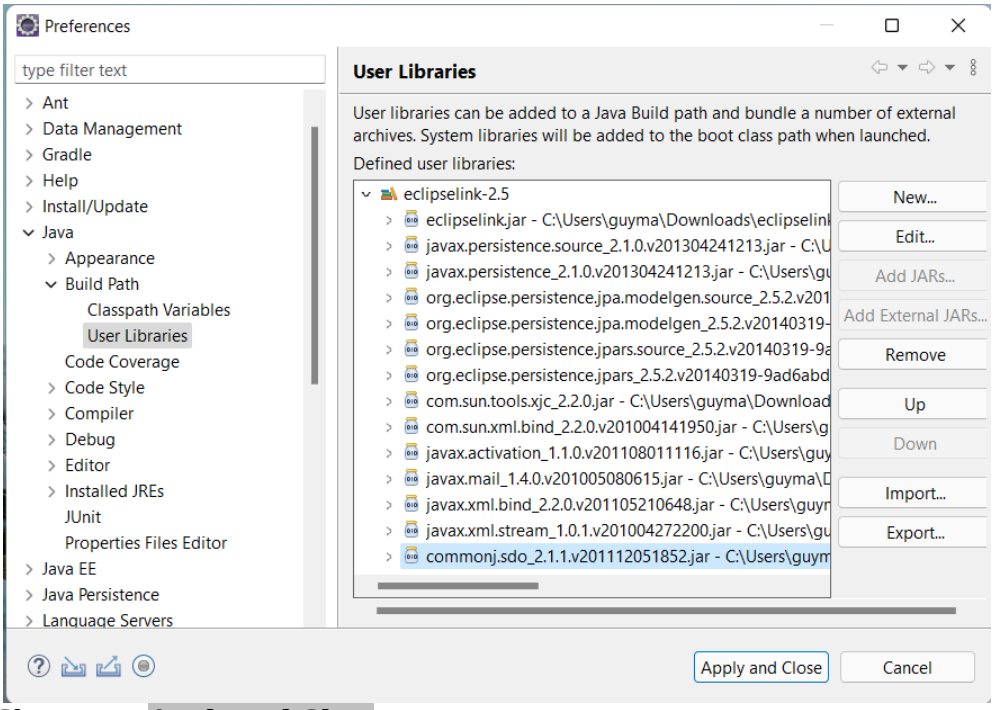


- La fenêtre ci-contre **Download Library** est affichée.
- Sélectionner la librairie **EclipeLink 2.5.2**.
- **Next**.
- Accepter la licence puis **Finish**. On revient à la fenêtre précédente quand le téléchargement est terminé.



Attention: l'étape précédente ne fonctionne plus avec les dernières versions d'Eclipse.
Solution
La librairie n'est plus téléchargée comme cela devrait.
On va créer sa propre librairie EclipseLink avec les fichiers jar d'EclipseLink que l'on doit télécharger.

1) Télécharger la version 2.5 d'EclipseLink à partir du site https://www.eclipse.org/eclipselink/releases/2.5.php .
   Sélectionner le lien **EclipseLink 2.2.5 InstallerZip** pour le téléchargement.
2) Dézipper le fichier, cela donne un dossier eclipselink.
3) Dans Eclipse
   a. Menu **Window>Preferences>Java>Build Path>User Libraries**
   b. Cliquez sur le bouton **New**
   c. Donnez le nom **eclipselink-2.5** à la librairie à créer, inutile de cocher la case
   d. Cliquez maintenant sur le bouton **Add External JARs…**
   e. Rechercher et sélectionnez tous les fichiers jar du dossier….\eclipselink\jlib



   f. Cliquez sur **Apply and Close**
   g. Revenir dans la fenêtre New JPA Project, cliquez sur 4, le bouton **Manage**

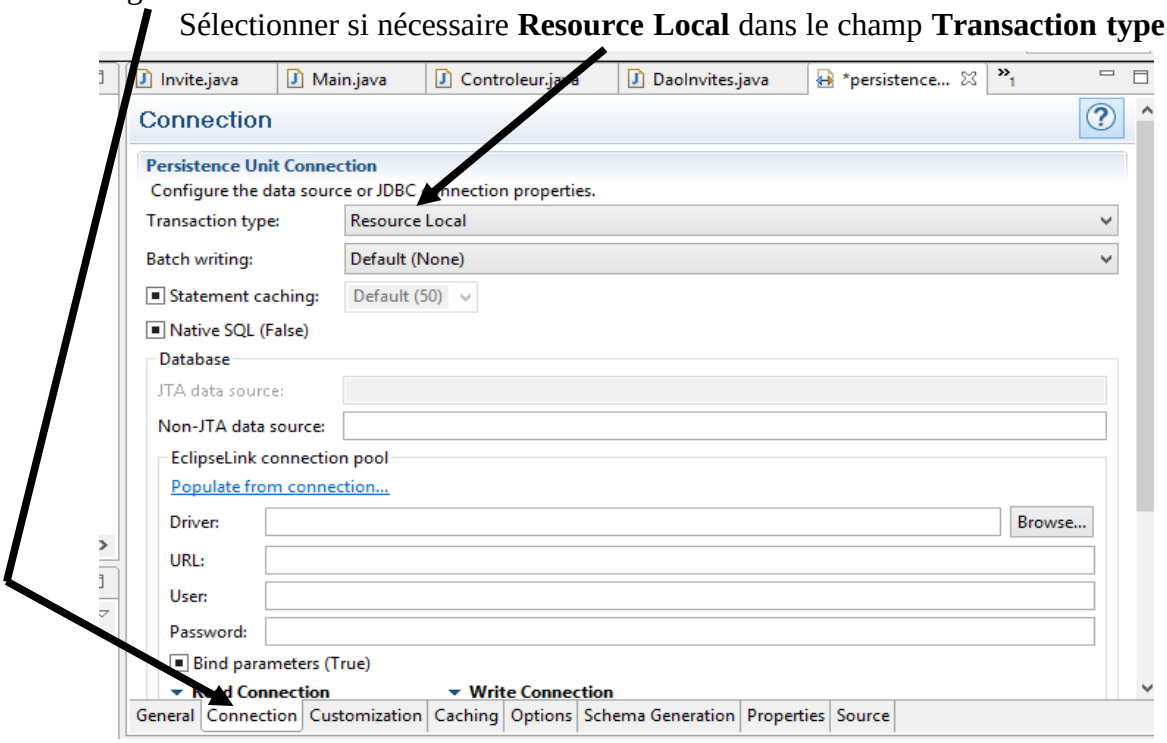
libraries… , et sélectionnez la librairie créée**.**

- Ouvrir **Connection** ③ et sélectionner la connexion précédente créée.
- Cocher la case **Add drive library to buid path**
- Cliquer sur **Finish**.
- Ne pas tenir compte d'un éventuel message d'erreur sur **persistence.xml**.

Le fichier **persistence.xml**

Le fichier **persistence.xml** contient les informations utilisées par **EclipseLink** pour la connexion de l'application développée à la base.
Ce fichier est dans le dossier **NomDuProjet/src/META-INF**.
►Sélectionner **persistence.xml / Bouton droit / Open with / Persistence XML Editor**
►Ouvrir l'onglet **Connection**.

Sélectionner si nécessaire **Resource Local** dans le champ **Transaction type**.



►Cliquer sur **Populate from connection**.

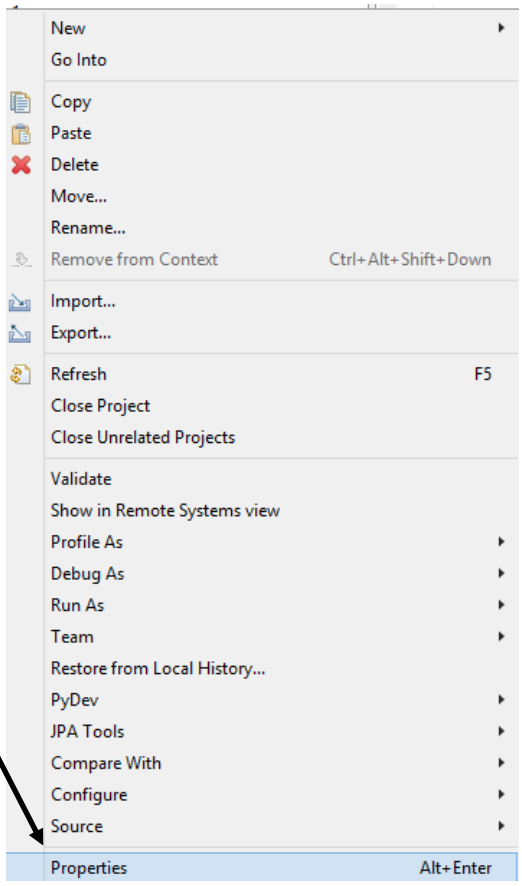
►Sélectionner la connexion précédemment créée puis **OK**.

►Sauver les modifications (la sauvegarde n'est pas automatique).

►Visualiser pour information le fichier **persistence.xml** dans l'onglet **Source** ainsi que l'onglet **Properties**.

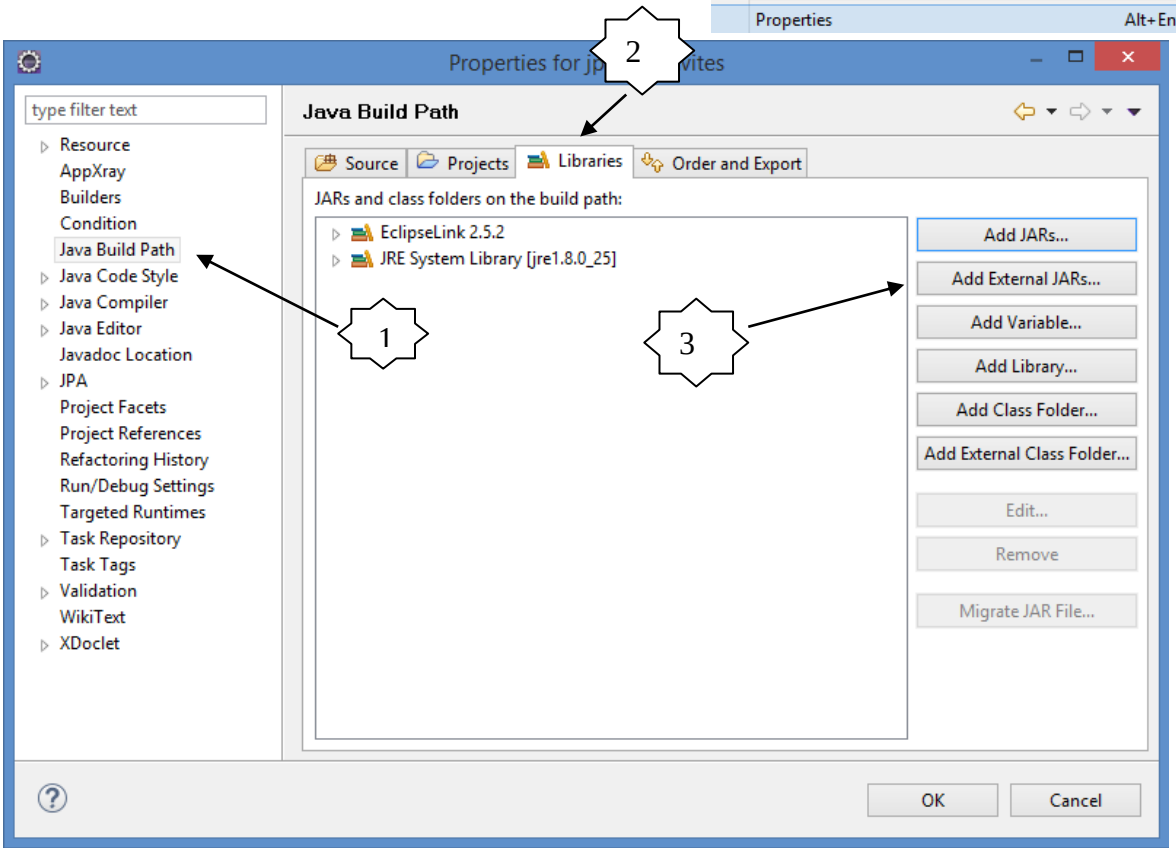
☞Si on a oublié de cocher la case **Add drive library to buid path dans le formulaire précédent,** il faut ajouter le connecteur mysql à l'application:

- Sélectionner le projet.
- Bouton droit.
- Choisir **Properties** (en bas).



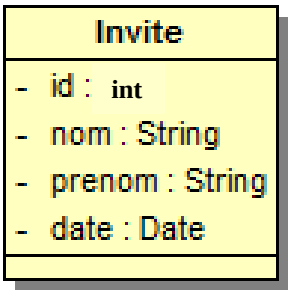
L'interface ci-dessous est affichée.

- Sélectionner Java Build Path ①
  - Onglet Librairies ②
    - Add External JARs... ③



Rechercher dans l'arborescence le fichier **mysql-connector-java-5.1.34-bin.jar** (normalement installé dans le dossier **C:\Program Files (x86)\MySQL\MySQL Connector J**). Puis **Ouvrir**. Le fichier est normalement ajouté dans la fenêtre Libraries. Puis **OK**.

Création du modèle – la classe JPA Entity

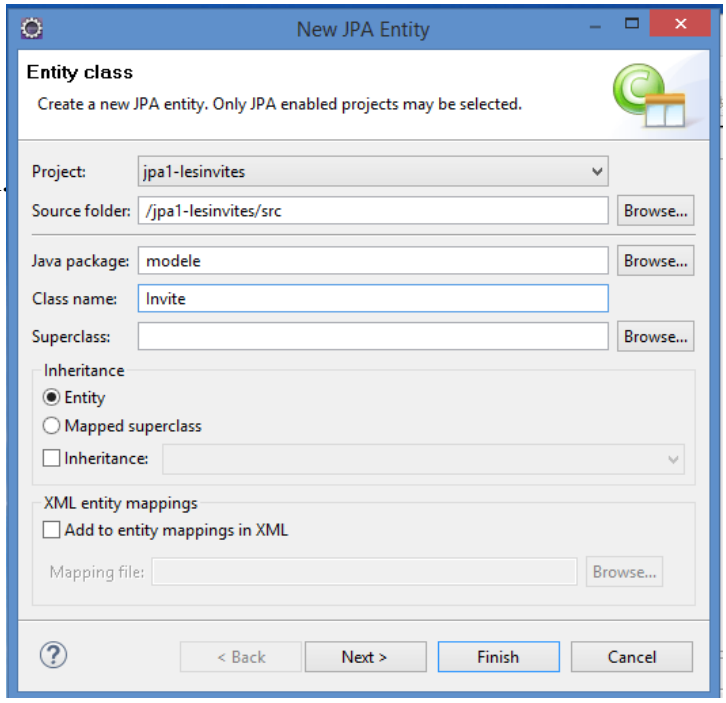


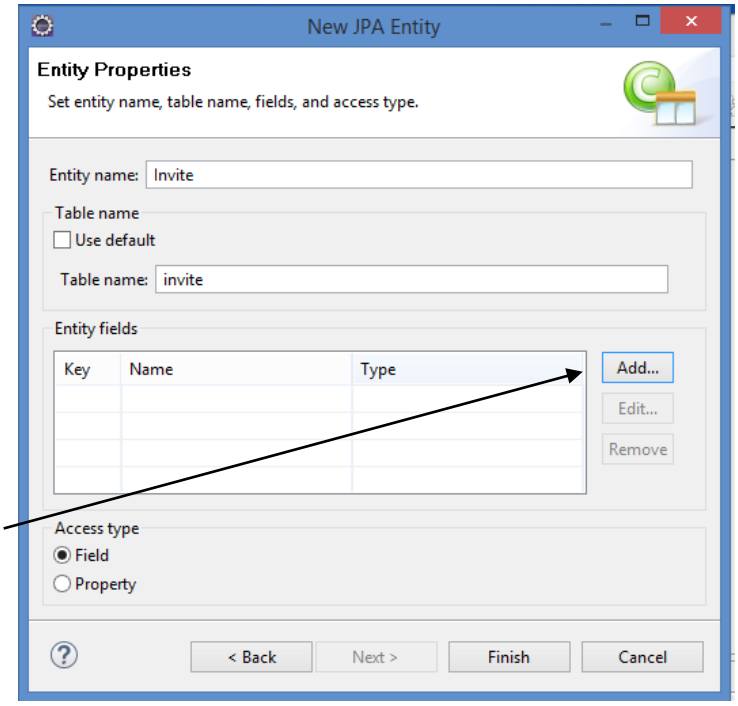
Une entité JPA est une classe Java qui correspond à une table de la base sélectionnée. L'entity JPA sera utilisée pour créer la table invite dans la base.

Voici le diagramme UML de la classe Invite avec uniquement ses attributs.

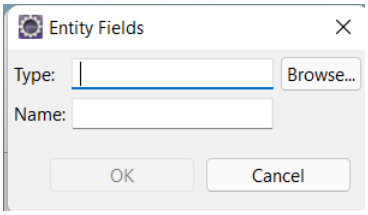
Ce modèle va servir à créer l'entity JPA.

- Créer un package **modele**.
- Cliquer sur **modele**.
- **Bouton droit/New/JPA Entity**
- Donner le nom **Invite** à la classe.
- **Next**

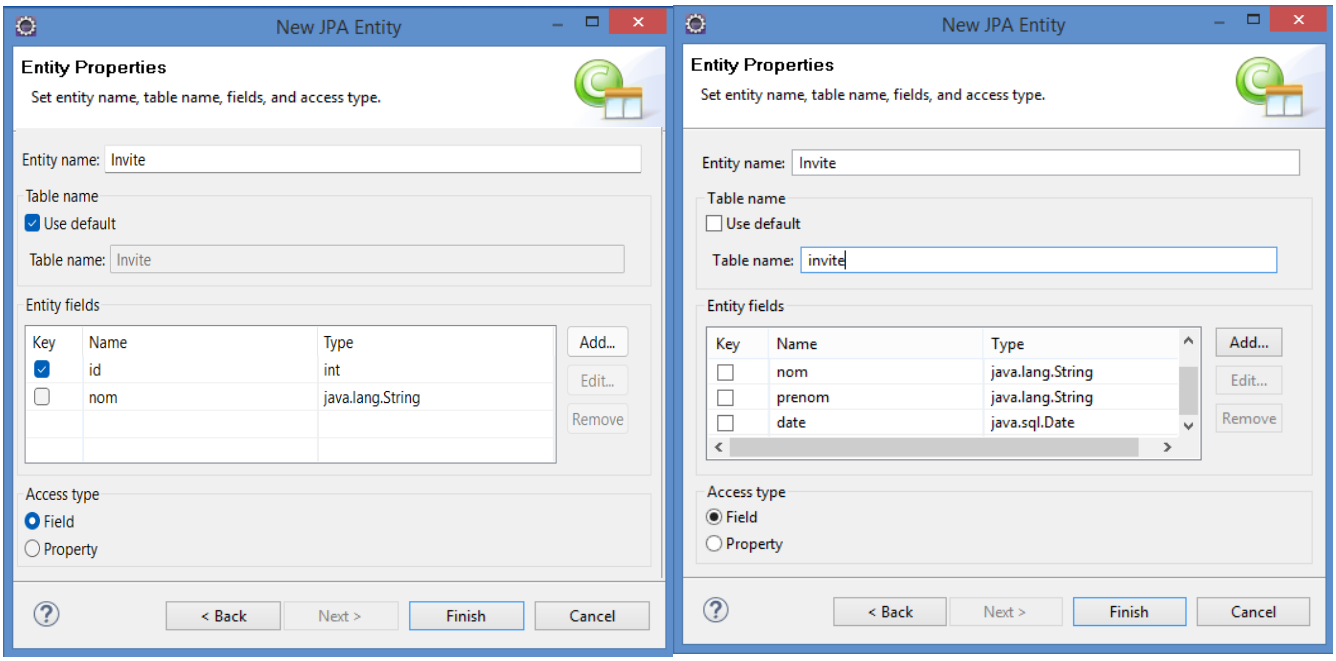




On peut construire la classe en ajoutant les divers champs avec **Add...**



Il faut alors sélectionner dans le formulaire le type de l'attribut puis son nom.
Pour la clef primaire, le type **int** et le nom **id**, voir figure suivante



Cocher la case **Key** pour l'attribut **id** car **id** est utilisé comme **clef primaire**.

Cliquer sur Finish quand tous les attributs (champs) sont créés.
L'entity est alors créée : c'est à dire une classe Java (de type Java Bean) **annotée @Entity**.

Une erreur est éventuellement indiquée sur l'annotation **@Table(name="invite")**, elle est normale car la table **invite** n'est pas encore créée.

☞Ouvrir la classe. Ajouter les éléments écrits en gras:

- l'annotation **@GeneratedValue(strategy=GenerationType.*IDENTITY*)** avant l'attribut **id** servant de clef primaire. **strategy=GenerationType.*IDENTITY*** indique à JPA que la clef primaire sera générée par le SGBD, en l'occurrence ici une auto incrémentation gérée par MySQL,
- le constructeur avec arguments, utilisé par la suite pour construire les objets,
- la méthode toString().

```
package modele;

import java.io.Serializable;
import java.lang.String;
import java.sql.Date;
import java.time.LocalDate;
import javax.persistence.*;

/**
 * Entity implementation class for Entity: Invite
 */
@Entity
@Table(name="invite")

public class Invite implements Serializable {

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

```
	private String nom;
	private String prenom;
	private Date date;
	private static final long serialVersionUID = 1L;

	public Invite() {
		super();
	}
	public Invite(String nom, String prenom) {
		super();
		this.nom = nom ;
		this.prenom = prenom ;
		date = Date.valueOf(LocalDate.now());
	}
	public int getId() {
		return this.id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getNom() {
		return this.nom;
	}
	public void setNom(String nom) {
		this.nom = nom;
	}
	public String getPrenom() {
		return this.prenom;
	}
	public void setPrenom(String prenom) {
		this.prenom = prenom;
	}
	public Date getDate() {
		return this.date;
	}
	public void setDate(Date date) {
		this.date = date;
	}
	public  String toString(){
		return nom+" "+" "+prenom + "  " +
		    date.toLocalDate().format(DateTimeFormatter.ofPattern(
		                                "dd-MM-yyyy"));
	}
}
```

L'annotation **@Entity** sur la classe **Invite** indique que la classe **Invite** est une entité et donc que ses instances peuvent être persistées.
L'annotation optionnelle **@Table(name="invite")** donne le nom de la table associée à la classe, il est inutile si les 2 noms sont identiques. Elle est nécessaire si le nom de la table et le nom de la classe sont différents.

On veut mémoriser pour chaque invité sa date de création dans l'attribut **date**, on a dans le constructeur ajouté;

```
date = Date.valueOf(LocalDate.now());
```

LocalDate.now() retourne un objet **LocalDate** à la date de son exécution.
La méthode statique **java.sql.Date.valueOf(LocalDate.now())** retourne une instance de la date de type **java.sql.Date** au moment de l'exécution du constructeur.

Création de la table invite à partir de la classe JPA Entity Invite (partie non réalisable si utilisation de la Generic Platform)

- Sélectionner le projet: **Bouton droit / JPA tools / Generate Tables from Entities**
- Accepter **Database** pour **Generation Output Mode.**
- **Finish**
- Cliquer sur **Yes** dans la boite du message d'avertissement.

La table doit être créée dans la base **lesinvites**.
Les messages des actions réalisées sont affichés dans la Console.

Une application utilisant la classe JPA Entity Invite

Créer dans le package par défaut la classe suivante.

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import modele.Invite;

public class TestInvite {
	public static void main(String[] args) {
	    EntityManagerFactory factory = Persistence.createEntityManagerFactory("jpa-1");

		EntityManager em ;
		em = factory.createEntityManager();

		Invite invite = new Invite("Durand","Robert");

		em.getTransaction().begin();
		em.persist(invite);
		em.getTransaction().commit();
	}
}
```

Nom du projet JPA!!!

Explications
Les opérations CRUD et les requêtes SQL doivent être exécutées par un **EntityManager**.

Un objet **EntityManagerFactory** est en 1er créé à l'aide de la méthode statique **createEntityManagerFactory()** de la classe **Persistence** qui reçoit comme argument par défaut le nom du projet JPA (ici jpa1-1, à remplacer si nécessaire par le nom du projet).

```
	EntityManagerFactory factory = Persistence.createEntityManagerFactory("jpa1-1");
```

Nom du projet JPA!!!

L'objet **EntityManager** est ensuite créé avec la méthode **createEntityManager()** de la classe **EntityManagerFactory**.

```
private EntityManager em ;
em = factory.createEntityManager();
```

Instanciation d'un objet Invite

```
Invite invite = new Invite("durand","robert");
```

On rend l'objet persistant: il est enregistré dans la base de données.

```
em.getTransaction().begin();
em.persist(invite);
em.getTransaction().commit();
```

La méthode **persit()** correspond à l'opération **Create**.

Vérifier que l'exécution de ce programme a bien créé un enregistrement dans la base **lesinvites**.

►JPA fournit les 4 opérations CRUD (de base) liées à la persistance des données

Opération C – Create – qui consiste persister un enregistrement dans la base à partir d'un objet JPA Entity.

```
em.getTransaction().begin();
em.persist(invite);
em.getTransaction().commit();
```

Opération R – Read – pour extraire un enregistrement.

Exemple d'une méthode pour rechercher un enregistrement à partir de la clef.

```
public Invite recherche(int key) {
        Invite i ;
        em.getTransaction().begin();
        i = em.find(Invite.class, key);
        em.getTransaction().commit();
        return i ;
}
```

La méthode **find()** retourne **null** si l'enregistrement n'est pas trouvé.

Opération U – Update – pour mettre à jour un enregistrement.

Exemple d'une mise à jour d'un invité : recherche par la clef et changement de son prénom.

```
public Invite majPrenom(String prenom, int key) {
        Invite i ;
        em.getTransaction().begin();
        i = em.find(Invite.class, key);
        i.setPrenom(prenom) ;
        em.getTransaction().commit();
        return i ;
}
```

Opération D – Delete – pour supprimer un enregistrement.

```
em.getTransaction().begin();
```

```
        em.remove(i);
        em.getTransaction().commit();
```

►JPA fournit également des requêtes de type SQL

JPA utilise le langage **JPQL** assez semblable à SQL pour créer des requêtes qui manipulent des **objets**.
L'objet **EntityManager** propose des méthodes permettant de créer ces requêtes.

```
TypedQuery<Invite> query = em.createQuery(
                        "SELECT g FROM Invite g ORDER BY g.id", Invite.class);
```

Une documentation complète sur le langage JPQL (Java Persistence Query Langage) se trouve à l'URL **https://docs.oracle.com/javaee/7/tutorial/partpersist.htm chapitre 39**.

2ème partie: réalisation d'une application

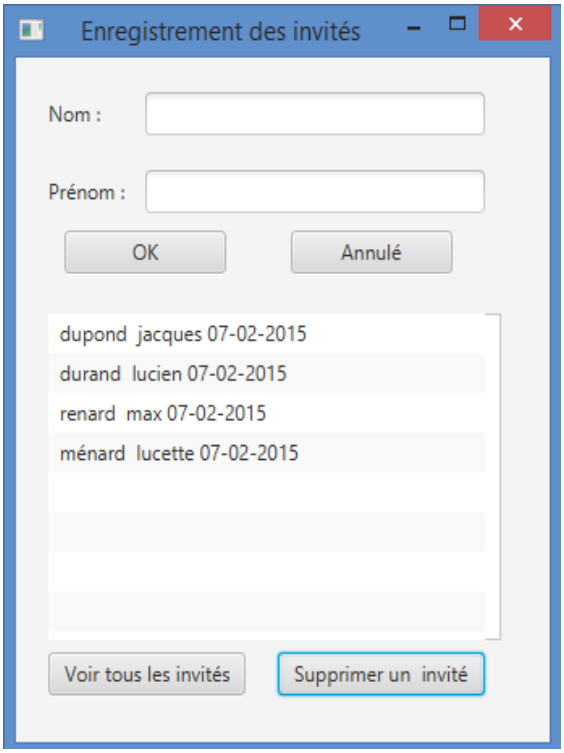
Présentation et spécifications de l'application

L'application proposée consiste à enregistrer des invités dans une base de données.

Les invités sont saisis et visualisés à l'aide de l'interface ci-contre. La date correspond à la date d'enregistrement de l'invité.

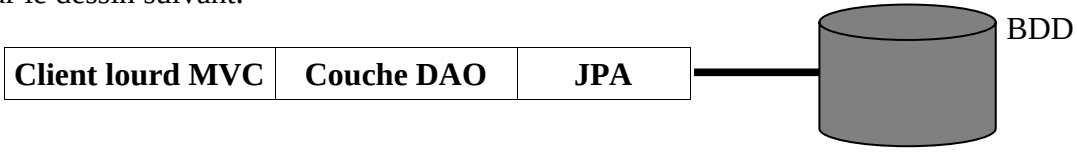
On peut :

- ajouter un invité,
- voir tous les invités enregistrés,
- effacer un invité particulier en le sélectionnant dans la **ListView**.



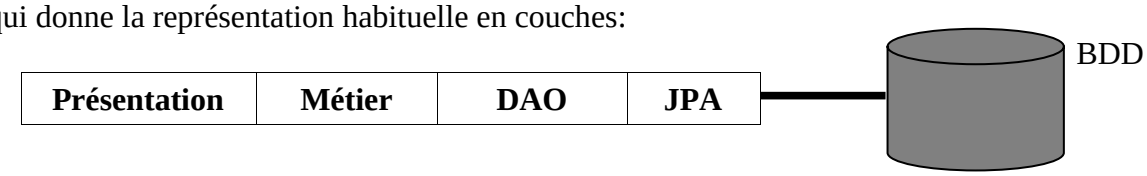
Organisation logicielle

L'application proposée est découpée et organisée en plusieurs couches logicielles comme indiqué sur le dessin suivant.



Le client lourd respecte le patron MVC.
Le contrôleur du client lourd communique avec la couche DAO (Data Access Object) pour réaliser les transactions avec la BDD.
La couche DAO propose des méthodes au client MVC pour effectuer les transactions avec la BDD.
JPA (Java Persistent API) est implémenté par **EclipseLink**. La couche DAO l'utilise pour ses transactions avec la BDD.
Le développement du client lourd est ainsi indépendant de l'implémentation de la BDD: le client ne voit que les méthodes présentées par les classes de la couche DAO. L'ensemble Couche DAO-JPA rend l'implémentation réelle de la BDD et les requêtes utilisées «transparentes» pour le développement du client.

Dans le découpage traditionnel d'une application en couches, la vue est désignée par «couche Présentation» et les classes Contrôleur et Modèle sont rangées dans la «couche Métier»
Ce qui donne la représentation habituelle en couches:

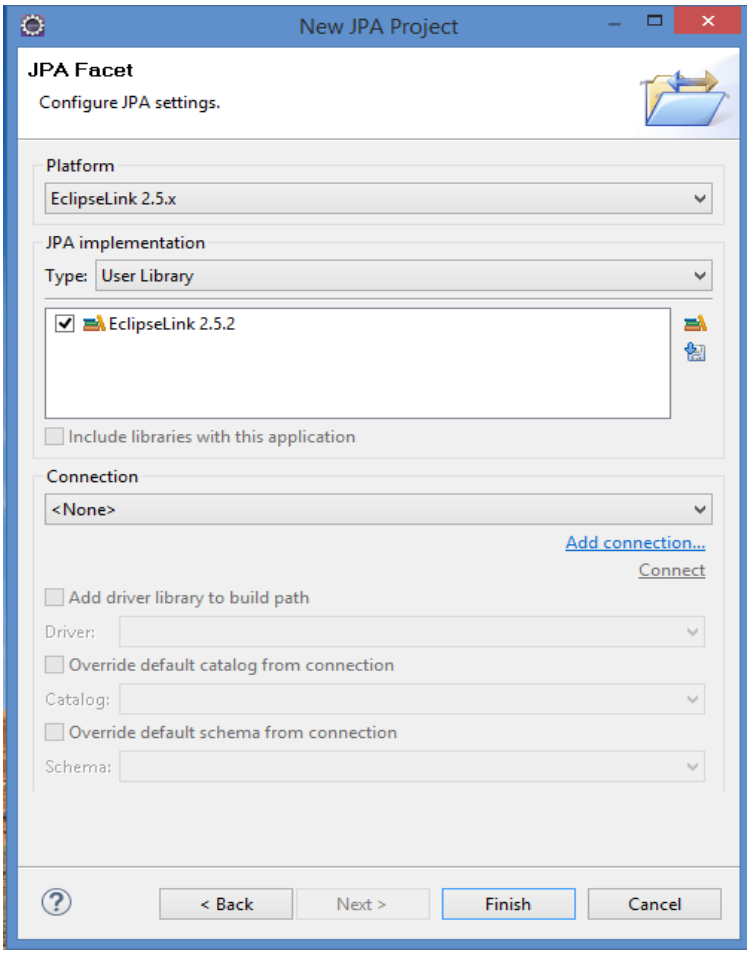


Le projet JPA

On peut utiliser la même connexion que pour l'exercice précédent ou en créer une nouvelle avec les mêmes paramètres de connexion.

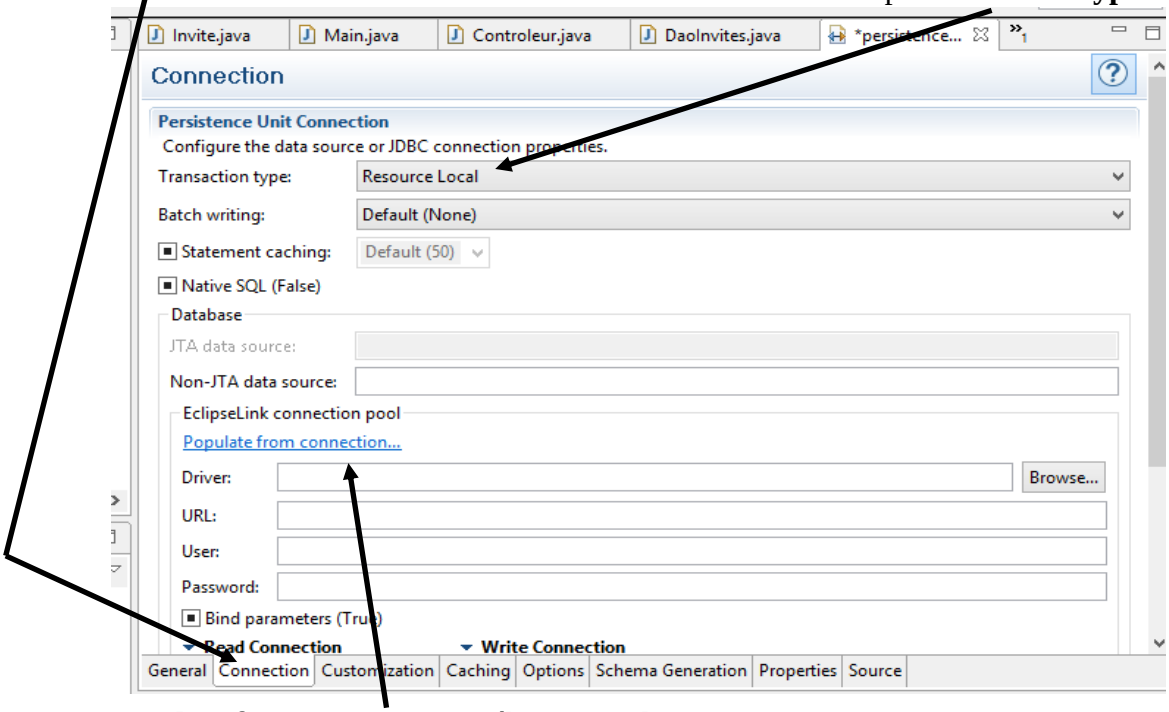
Création du projet JPA

- **File/New/JPA Project**
- Donner un nom, par exemple jpa-lesinvites
- Sélectionner **jre1.8** dans **Target Runtime**
- **Next**
- **Next**
  On peut voir que **EclipseLink** est bien installé, les champs sont correctement remplis.
- Sélectionner dans **Connection** la connexion MySQL puis **connect** si nécessaire
- Cocher éventuellement sur
  Add driver library to build path, pour ajouter le connecteur au projet.
- **Finish**



Le fichier **persistence.xml**

Ce fichier est dans le dossier **NomDuProjet/src/META-INF**.
Sélectionner **persistence.xml / Bouton droit / Open with / Persistence XML Editor**
Ouvrir l'onglet **Connection**.
Sélectionner si nécessaire **Resource Local** dans le champ **Transaction type**.



Cliquer sur **Populate from connection**. Sélectionner la connexion puis **OK**.
Sauver les modifications.
💣Ne pas oublier d'ajouter **mysql-connector-java-5.1.34-bin.jar** dans les librairies du projet si oubli d'avoir coché la case **Add driver library to build path** à la création du projet dans la fenêtre **File/New/JPA Project**.

Création de l'entity JPA

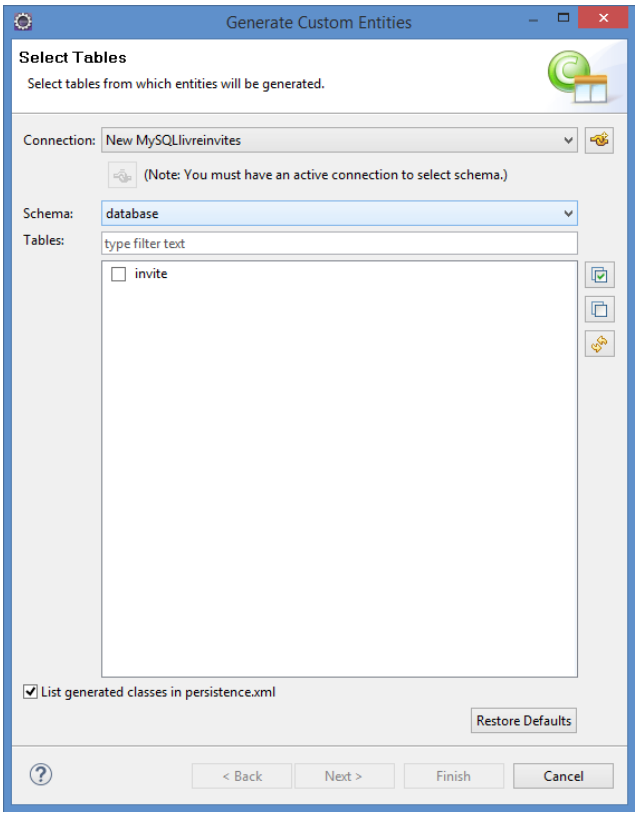
Cette fois-ci, on crée l'entity JPA à partir de la table invite créée lors de la 1ère partie.

Sélectionner le projet.

- **Bouton droit**
- **New**
- **JPA Entities from Tables**

La fenêtre ouverte ci-contre montre :

– la connexion active pour exécuter EclipseLink,
– la table trouvée dans la base.



- Cocher (si nécessaire) la table **invite**.

Le bouton  permet de rechercher ou créer une connexion si aucune n'est sélectionnée.

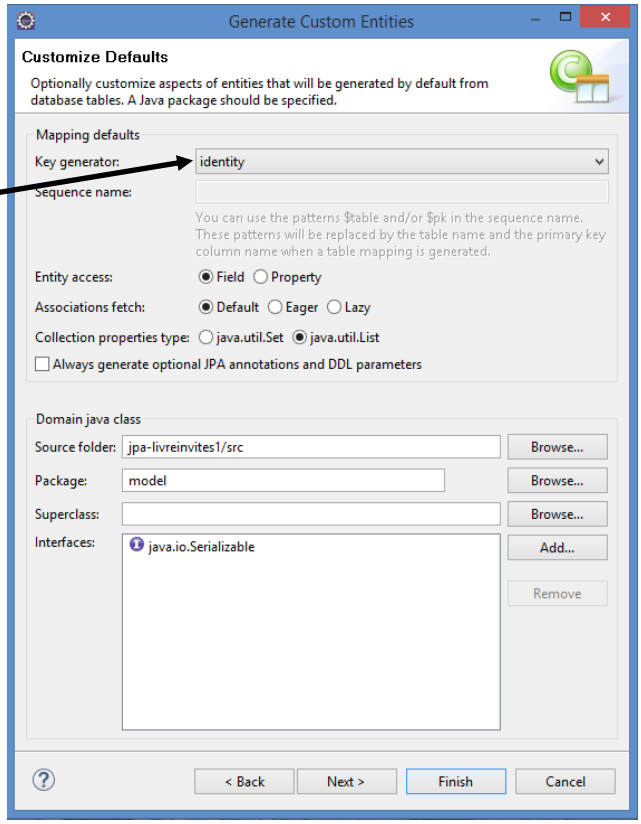
Le bouton  permet de rafraichir la lecture de la base de données et d'afficher les tables.

- **Next**.
- **Next**.

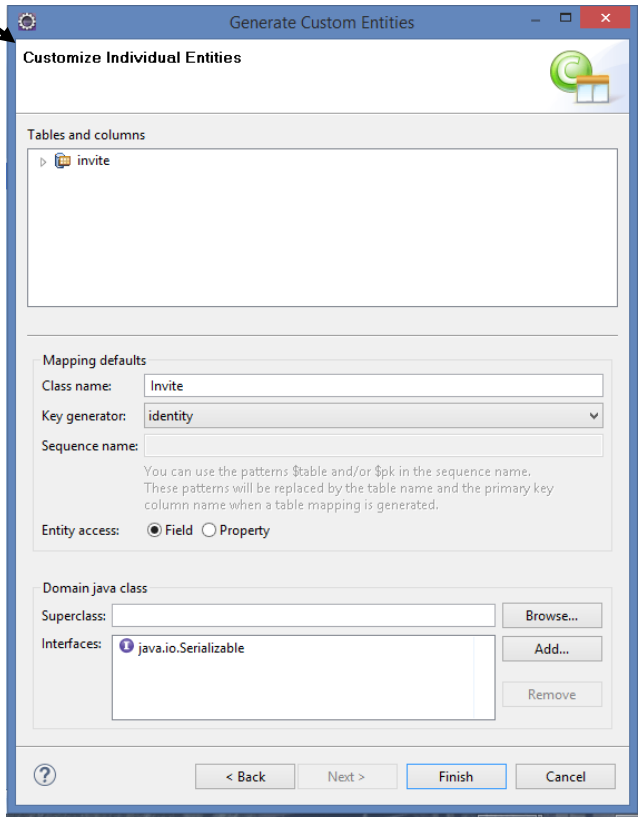
  Sélectionner **identity** dans **Key generator** pour indiquer une clef primaire auto incrémentée.

  On laisse les autres valeurs par défaut.

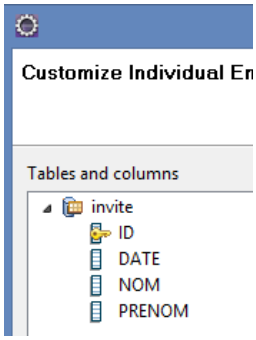
- **Next**. Cela affiche l'interface suivante.



En ouvrant **invite** cette interface sert éventuellement à modifier le type d'un attribut ou à sélectionner les champs de la table qu'on ne veut pas comme attribut.



Pour cela, il suffit de cliquer sur un attribut et de décocher la case **Generate this property**.



💣Vérifier le type de l'attribut **ID**, il doit être de type **int** (ou long).
Laisser les autres valeurs par défaut.

Cliquer sur **Finish**, ce qui entraîne la

génération de la classe Invite.

Classe Invite générée par JPA-EclipseLink

```
package model;

import java.io.Serializable;
import javax.persistence.*;
import java.util.Date;

/**
 * The persistent class for the invite database table.
 *
 */
@Entity
@NamedQuery(name="Invite.findAll", query="SELECT i FROM Invite i")
public class Invite implements Serializable {
	private static final long serialVersionUID = 1L;

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

	@Temporal(TemporalType.DATE)
	private Date date;

	private String nom;

	private String prenom;

	public Invite() {
	}

	public int getId() {
		return this.id;
	}

	public void setId(int id) {
		this.id = id;
	}

	public Date getDate() {
		return this.date;
	}

	public void setDate(Date date) {
		this.date = date;
	}

	public String getNom() {
		return this.nom;
```

```
        }

        public void setNom(String nom) {
                this.nom = nom;
        }

        public String getPrenom() {
                return this.prenom;
        }

        public void setPrenom(String prenom) {
                this.prenom = prenom;
        }
}
```

Compléter la classe Invité comme dans la 1ère partie.

```
        public Invite(String nom, String prenom) {
                super();
                this.nom = nom ;
                this.prenom = prenom ;
                LocalDate localdate = LocalDate.now();
                date = new Date(1000*24*3600*localdate.toEpochDay());
                System.out.println("date de "+nom+" = "+date);
        }
        .........
        public  String toString(){
        String ladate = LocalDate.ofEpochDay((long)(Math.ceil((double)date.getTime()/
          (double)(1000*3600*24)))).format(DateTimeFormatter.ofPattern("dd-MM-yyyy"));
                return nom+" "+" "+prenom + "        " + ladate ;
        }
```

Création de la vue

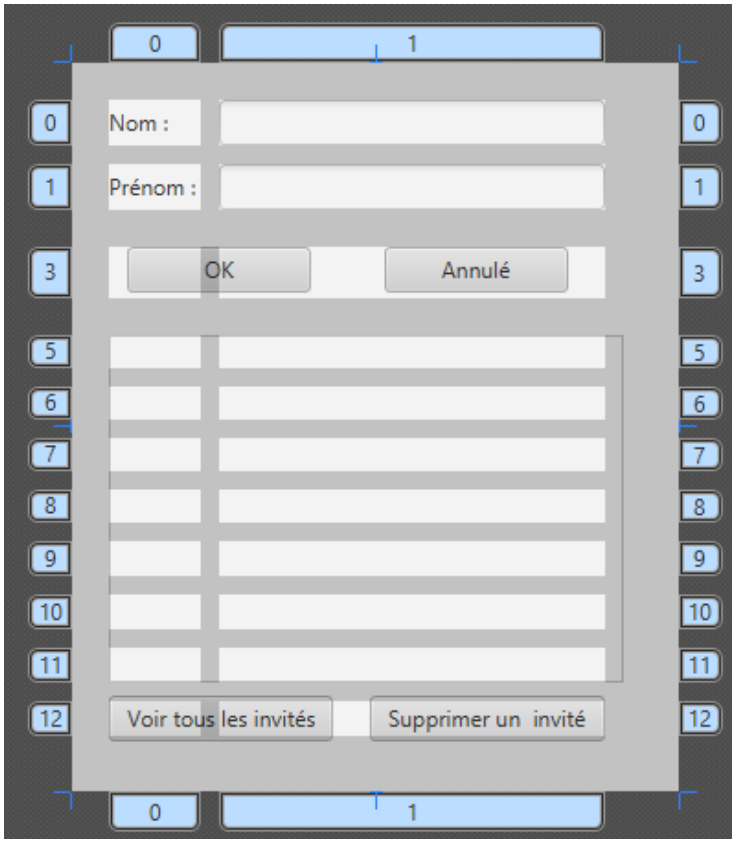
Créer un package **vue**.

Créer dans ce package un fichier **Vue.fxml**. Ouvrir ce fichier avec **SceneBuilder**.

Les noms et prénoms des invités sont saisis et validés en cliquant sur OK.
Le bouton Annulé efface les 2 champs de saisie.

L'affichage de tous les invités dans la **ListView** est obtenu en cliquant sur Voir tous les invités.

La suppression d'un invité



sélectionné dans la ListView est obtenu en cliquant sur Supprimer un invité.

La classe DAO (Data Access Object)

La classe DAO à écrire est spécialisée dans les transactions avec la base de données. Elle utilise l'implémentation **EclipseLink** de **JPA** en charge de gérer les transactions avec la base:

- connecter et déconnecter,
- exécuter des requêtes dans le langage JPQL, proche de SQL,
- transformer un objet java (une Entity JPA) en un enregistrement dans la table pour le stocker,
- transformer des enregistrements en objets java (Entity JPA).

La classe DAO propose aux classes utilisatrices des méthodes pour manipuler les objets java (Entity JPA). Ainsi, la technique d'accès à la base et les diverses requêtes sont complètement masquées pour les classes métiers.

```
package application;

import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;

import model.Invite;

public class DaoInvites {
        private EntityManager em ;
public DaoInvites() {
   EntityManagerFactory factory = Persistence.createEntityManagerFactory("jpa1-lesinvites");
        em = factory.createEntityManager();
        }
public void stocker(Invite invite){
        em.getTransaction().begin();
        em.persist(invite);
        em.getTransaction().commit();
        }
public List<Invite> voirlesinvites() {
        TypedQuery<Invite> query = em.createQuery(
                        "SELECT g FROM Invite g ORDER BY g.id", Invite.class);
        List<Invite> liste = query.getResultList();
        return liste;
        }
public void supprimer(Invite invite){
        em.getTransaction().begin();
        em.remove(invite);
        em.getTransaction().commit();
        }
}
```

Attention: à remplacer par le nom du projet

Explications

L'instanciation d'un objet DaoInvites crée l'objet EntityManager.
Les diverses méthodes proposées permettent :

- d'enregistrer un invité avec la méthode stocker(Invite invite),
- de supprimer un invité avec la méthode supprimer(Invite invite),
- de voir tous les invités avec la méthode voirlesinvites().

La classe **Invite** précise une requête nommée avec l'annotation @NamedQuery:

```
@NamedQuery(name="Invite.findAll", query="SELECT i FROM Invite i")
```

On peut l'utiliser directement dans la méthode **voirlesinvites()** de la classe **DaoInvites**:

```
public List<Invite> voirlesinvites() {
TypedQuery <Invite> query = em.createNamedQuery("Invite.findAll", Invite.class);
		List<Invite> liste = query.getResultList();
		return liste;
	}
```

La classe Contrôleur

```
package vue;

import java.util.List;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.fxml.FXML;
import javafx.scene.control.ListView;
import javafx.scene.control.TextField;
import application.DaoInvites;
import model.Invite;

public class Controleur {
	private DaoInvites daoinvites ;
	public Controleur() {
		daoinvites = new DaoInvites();		//le contrôleur crée l'objet dao
	}
	@FXML
	private TextField nom ;
	@FXML
	private TextField prenom ;
	@FXML
	private ListView<Invite> invites ;

	private ObservableList<Invite> listeinvites = FXCollections.observableArrayList();

	@FXML
	private void initialize() {
		invites.setItems(listeinvites);
	}
	@FXML
	private void ajouterInvite() {
		String Nom = nom.getText() ;
```

```
                String Prenom = prenom.getText();
                Invite i = new Invite(Nom,Prenom);
                daoinvites.stocker(i);
                nom.setText("");
                prenom.setText("");
        }
        @FXML
        private void annuler() {
                nom.setText("");
                prenom.setText("");
        }
        @FXML
        private void voirtouslesinvites() {
                listeinvites.clear();
                List<Invite> lesinvites = daoinvites.voirlesinvites();
                for (Invite i : lesinvites)
                        listeinvites.add(i);
        }
        @FXML
        private void supprimerinvite(){
                Invite sel = invites.getSelectionModel().getSelectedItem();
                if (sel !=null) {
                        daoinvites.supprimer(sel);
                        voirtouslesinvites();
                        }
        }
}
```

Utiliser SceneBuilder pour connecter la vue avec la classe Contrôleur et les divers champs manipulés. Voir le cours sur SceneBuilder.

Le programme de lancement de l'application

```
package application;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.GridPane;

public class Main extends Application {
        @Override
        public void start(Stage primaryStage) {
                try {
                        FXMLLoader  loader = new FXMLLoader() ;
                        loader.setLocation(Main.class.getResource("../vue/Vue.fxml"));

                        GridPane root = (GridPane)loader.load();
                        Scene scene = new Scene(root);
```

```
                primaryStage.setTitle("Enregistrement des invités");
                primaryStage.setScene(scene);
                primaryStage.show();
            } catch(Exception e) {
                e.printStackTrace();
            }
        }

        public static void main(String[] args) {
            launch(args);
        }
}
```

Tester le fonctionnement de cette application.
Apporter des modifications au code afin d'améliorer les fonctionnalités de cette application.

Déploiement de l'application

L'application étudiée peut être classifiée dans la catégorie 2 tiers:

- le client lourd MVC et la couche DAO dans un 1er tiers,
- la persistance des données (la BDD) dans un 2ème tiers.

Le terme **tiers** désigne souvent par extension la machine où sont déployés des couches et des éléments de l'application.

Le déploiement en 2 tiers physiques nécessite de créer un compte utilisateur sur MySQL car la connexion distante avec le compte "root" sur MySQL est interdite par défaut. On peut l'autoriser mais cela n'est pas conseillé.

►Création d'un utilisateur autorisé à réaliser tous types d'opérations sur la base nomBDD:
mysql> create user 'nomUtilisateur'@'%' identified by 'motDePasse';
mysql>grant all privileges on nomBDD.* to 'nomUtilisateur'@'%';
mysql>flush privileges;

►Ouvrir si nécessaire le port **mysqld** (3306) sur le pare-feu.

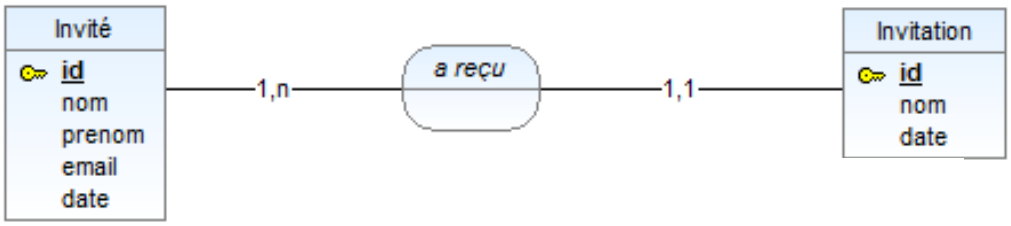
Accès au serveur Apache distant de Wamp (pour faire des tests)
►Ouvrir le fichier **C:\wamp\bin\apache\apache2.4.9\conf\httpd.conf**
Faire les modifications suivantes dans la section Directory
<Directory "c:/wamp/www/">
…..
 # Require local Ligne à mettre en commentaire en ajoutant un **#** devant
Require all granted # Ligne à ajouter
</Directory>
►Ouvrir si nécessaire le port httpd (80) sur le pare-feu.

3ème partie

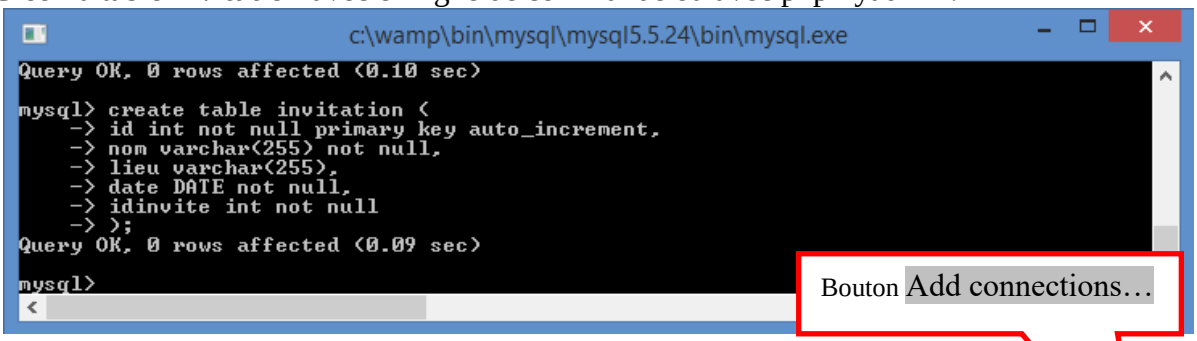
On ajoute à la base **lesinvités** une table **invitation** contenant les champs suivants:

- un entier **int** comme clef primaire,
- le nom de la manifestation,
- la date de la manifestation,
- l'identifiant de l'invité concerné (la clef primaire de la table invite),
- on peut ajouter le lieu de la manifestation (la ville).

Un invité peut recevoir des invitations à des manifestations différentes.

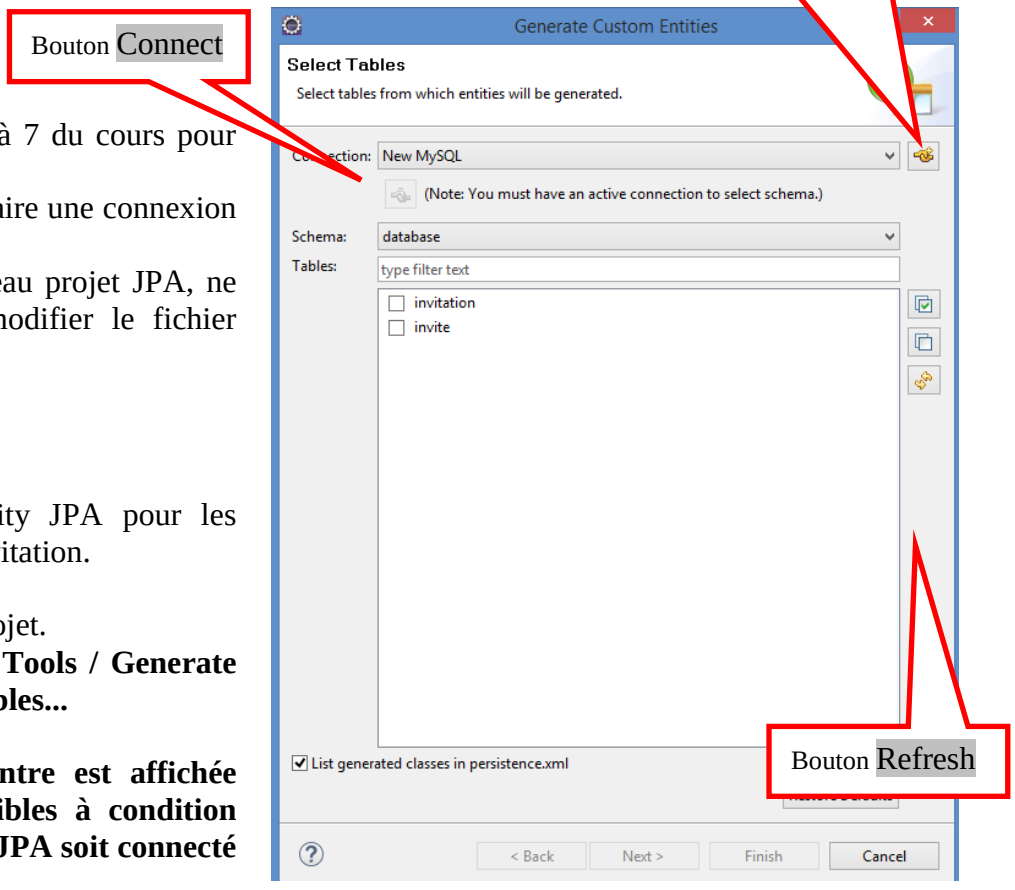


Créer la table **invitation** avec en ligne de commande ou avec phpmyadmin.



1ère étape

Voir les pages 1 à 7 du cours pour cette 1ère étape;
►Créer si nécessaire une connexion à la base.
►Créer un nouveau projet JPA, ne pas oublier de modifier le fichier **persistence.xml**.



2ème étape

Création des entity JPA pour les tables invité et invitation.

Sélectionner le projet.
Clic droit / JPA Tools / Generate Entities from Tables...

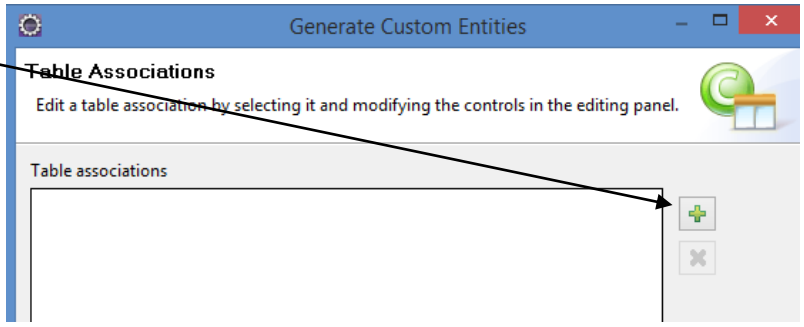
L'interface ci-contre est affichée et les tables visibles à condition que votre projet JPA soit connecté

à la base. Si les tables ne sont pas affichées, cliquez sur le bouton Connect si la connexion utile est visible dans la liste déroulante ou sur Add connections… pour créer une nouvelle connexion.
Cocher **invite** et **invitation**.
Cliquer sur le bouton **Refresh** si les 2 tables ne sont pas affichées après la connexion.

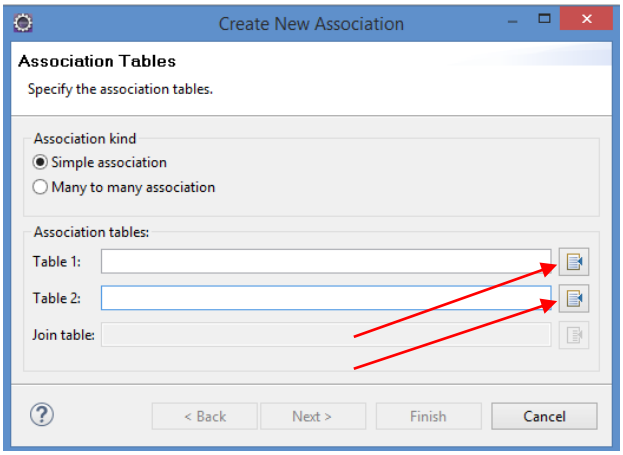
Puis **Next >**.

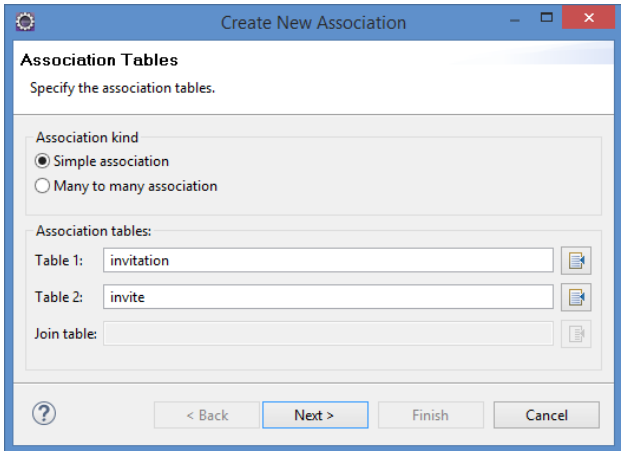
L'interface suivante est affichée: elle sert à préciser les associations entre les tables avant de générer les classes.

Cliquer sur ✚

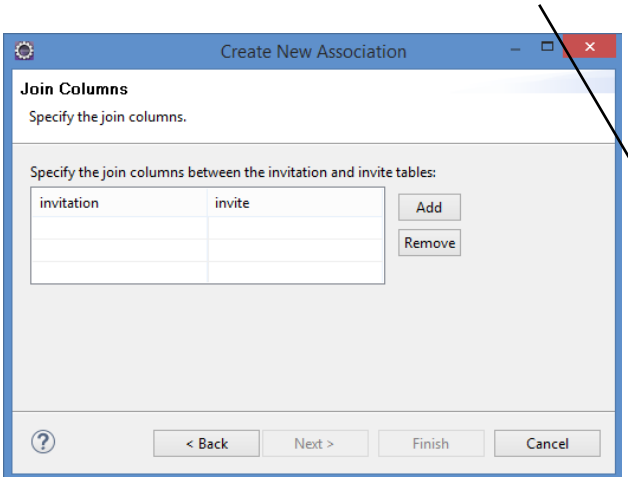


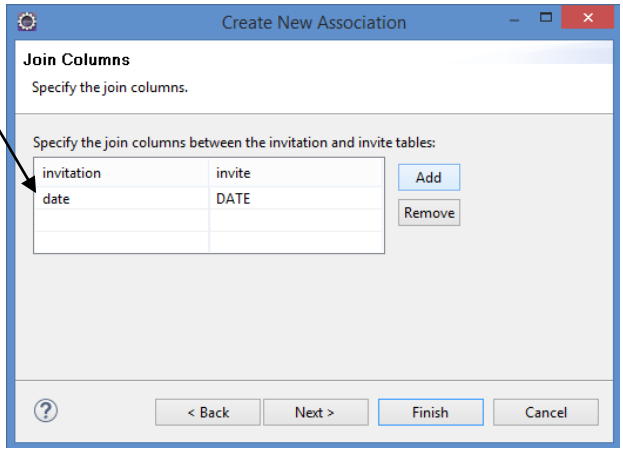
L'interface ci-dessous est affichée. Garder **Simple association**. Cliquer sur l'icône indiquée pour ajouter les tables dans **Table 1** et **Table 2** puis **Next >**.





L'interface ci-dessous est affichée. Cliquer sur **Add** pour préciser la colonne de jointure. Eclipse propose par défaut la colonne *date* dans les 2 tables.

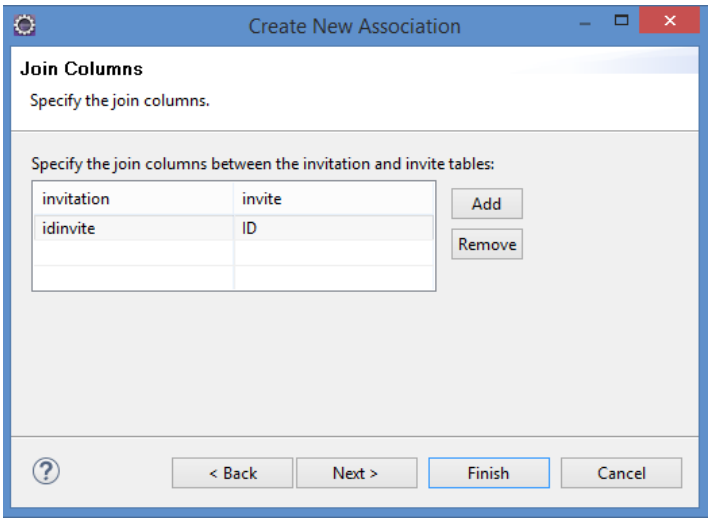




Cliquer sur *date* dans invitation. Une liste déroulante proposant les champs de la table est affichée. Sélectionner **idinvite**.

Cliquer sur DATE dans invite. Une liste déroulante proposant les champs de la table est affichée. Sélectionner **ID**.

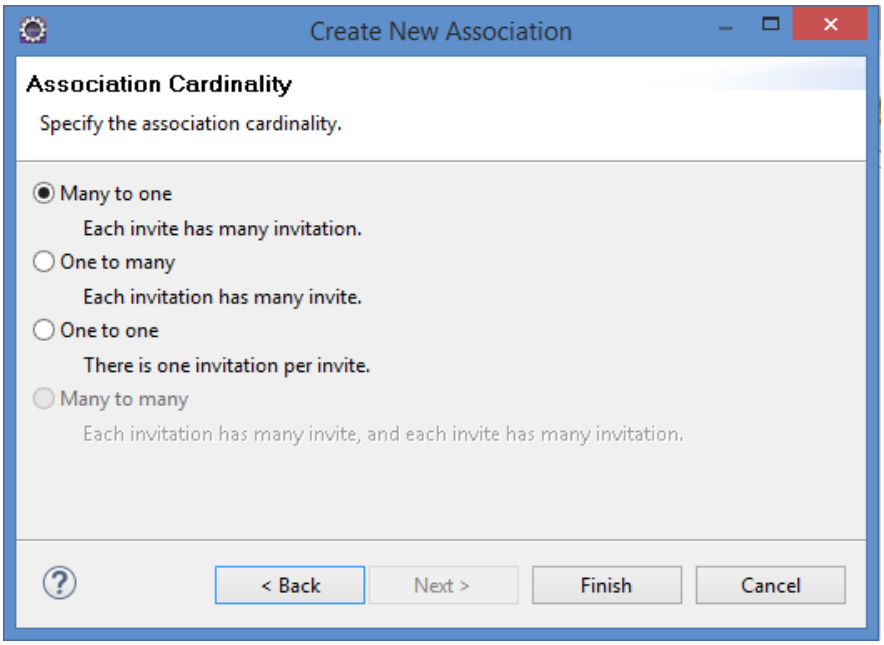
Puis Next >.



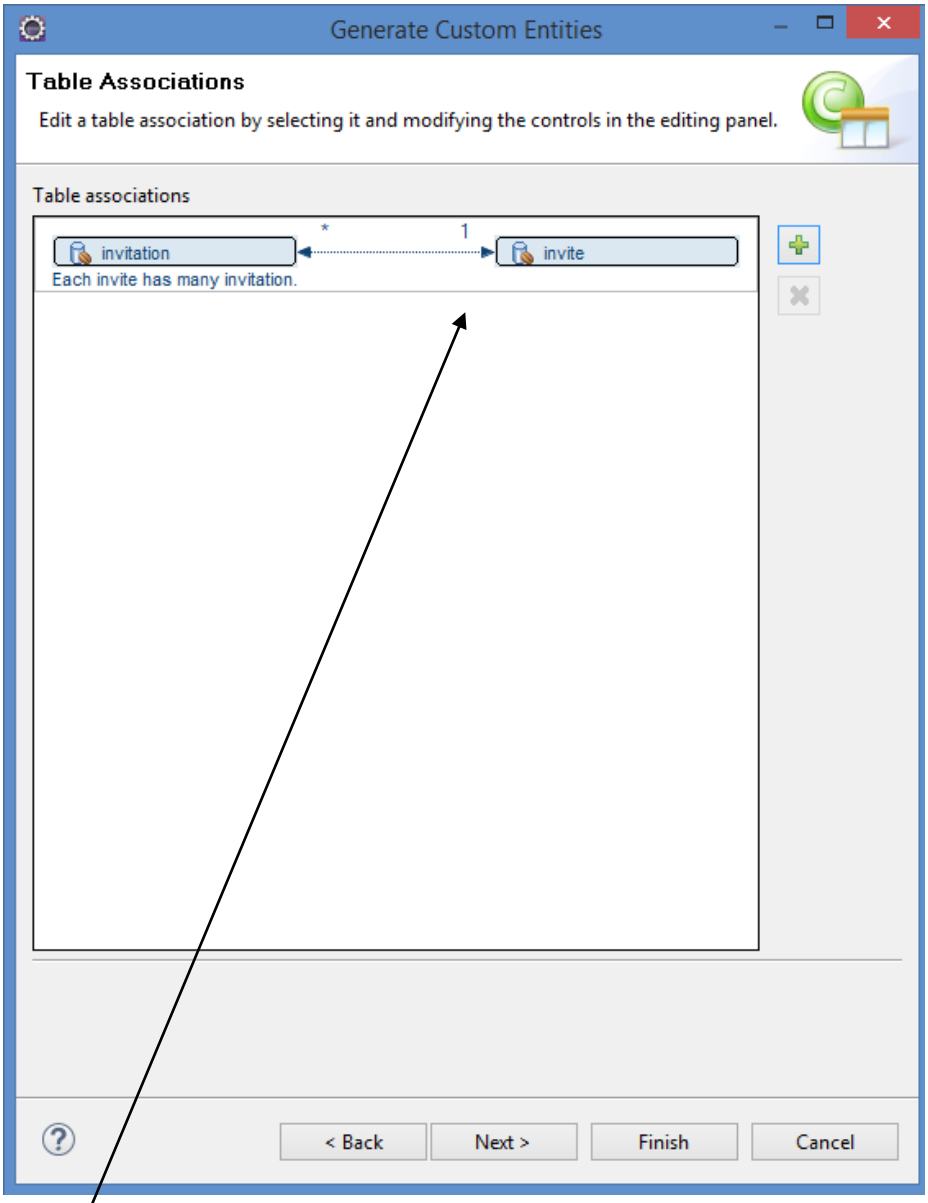
L'interface suivante propose une nouvelle association **Many to one** où chaque invité a plusieurs invitations.

💣L'interface peut proposer une association non convenable. Il faut impérativement vérifier la proposition d'association et la modifier le cas échéant.

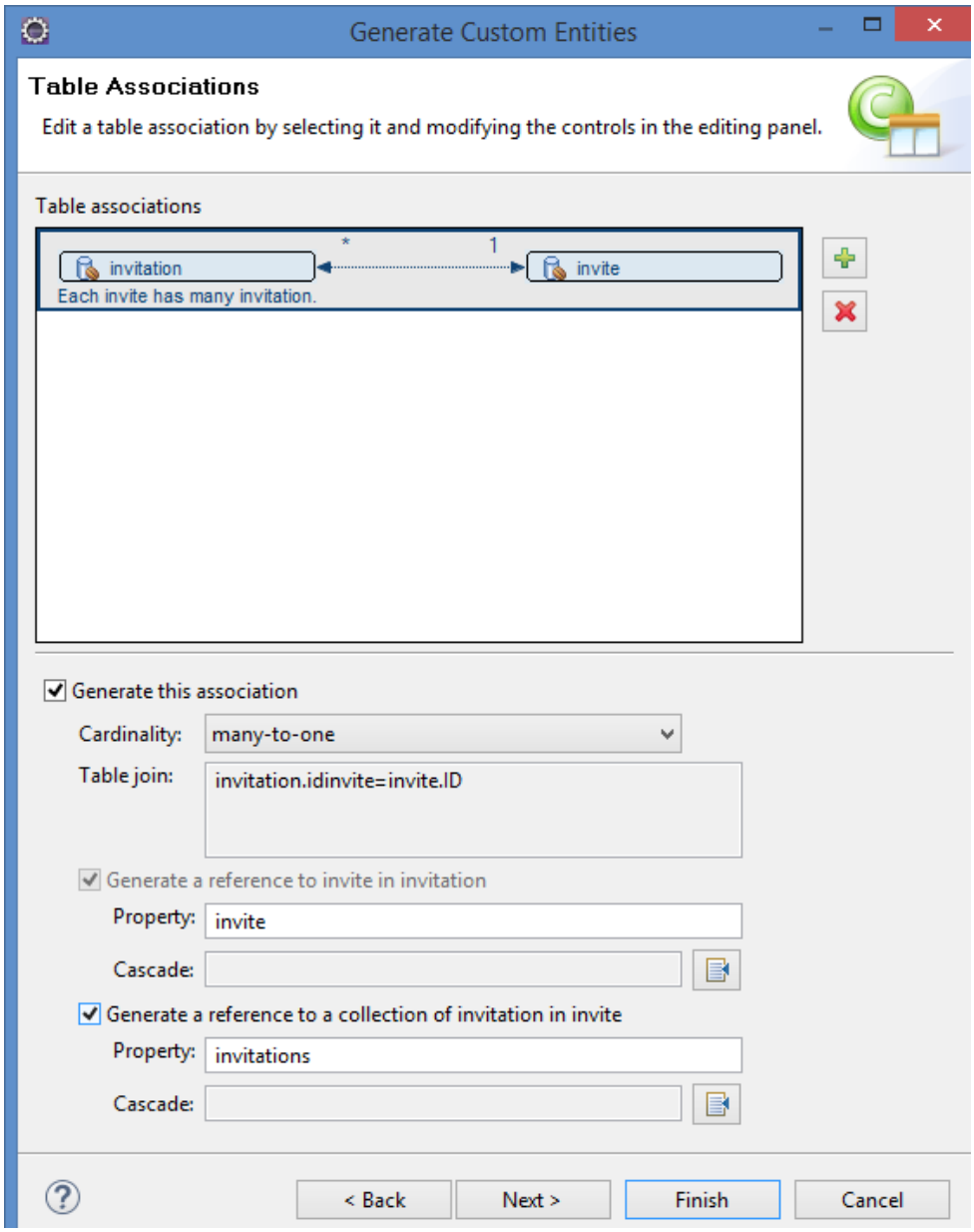
On garde ici ce choix. Puis Finish.



L'interface ci-contre montrant l'association entre les tables est affichée.



Cliquer sur **l'image de l'association** entraine l'affichage de la vue suivante.

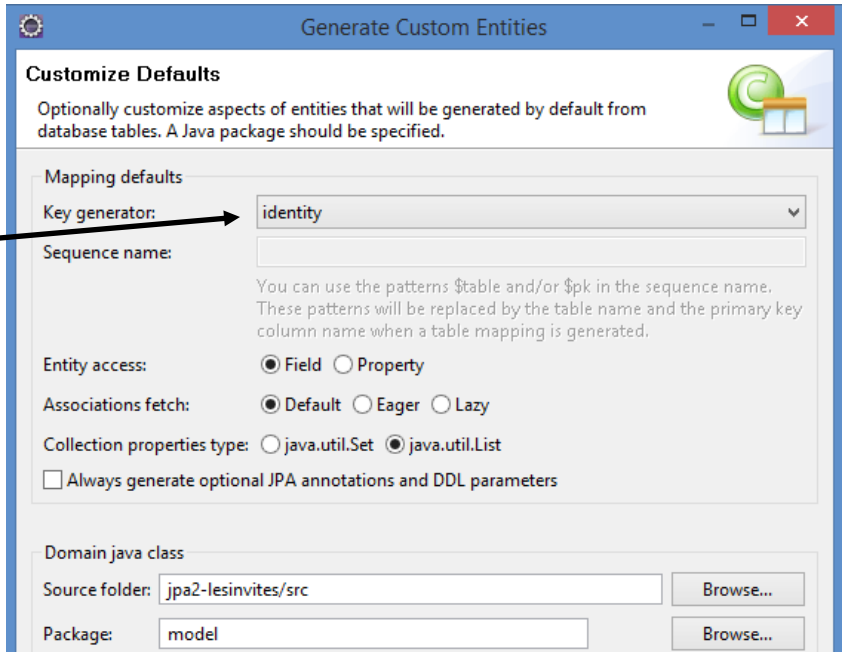


Eclipse propose de générer dans la classe **Invité** une collection d'Invitation.
Cliquer sur **Next >**.

L'interface suivante est affichée.

Sélectionner **identity** pour **Key generator**.

On délègue à MySQL la responsabilité de gérer les clefs primaires (ici avec auto incrémentation).



Cliquer sur **Next >**.

L'interface suivante permet de sélectionner/désélectionner la génération d'un attribut (property) à partir d'une colonne. Vérifier champ par champ la génération et la bonne correspondance entre le type d'une colonne et le type de l'attribut (surtout pour les clefs).

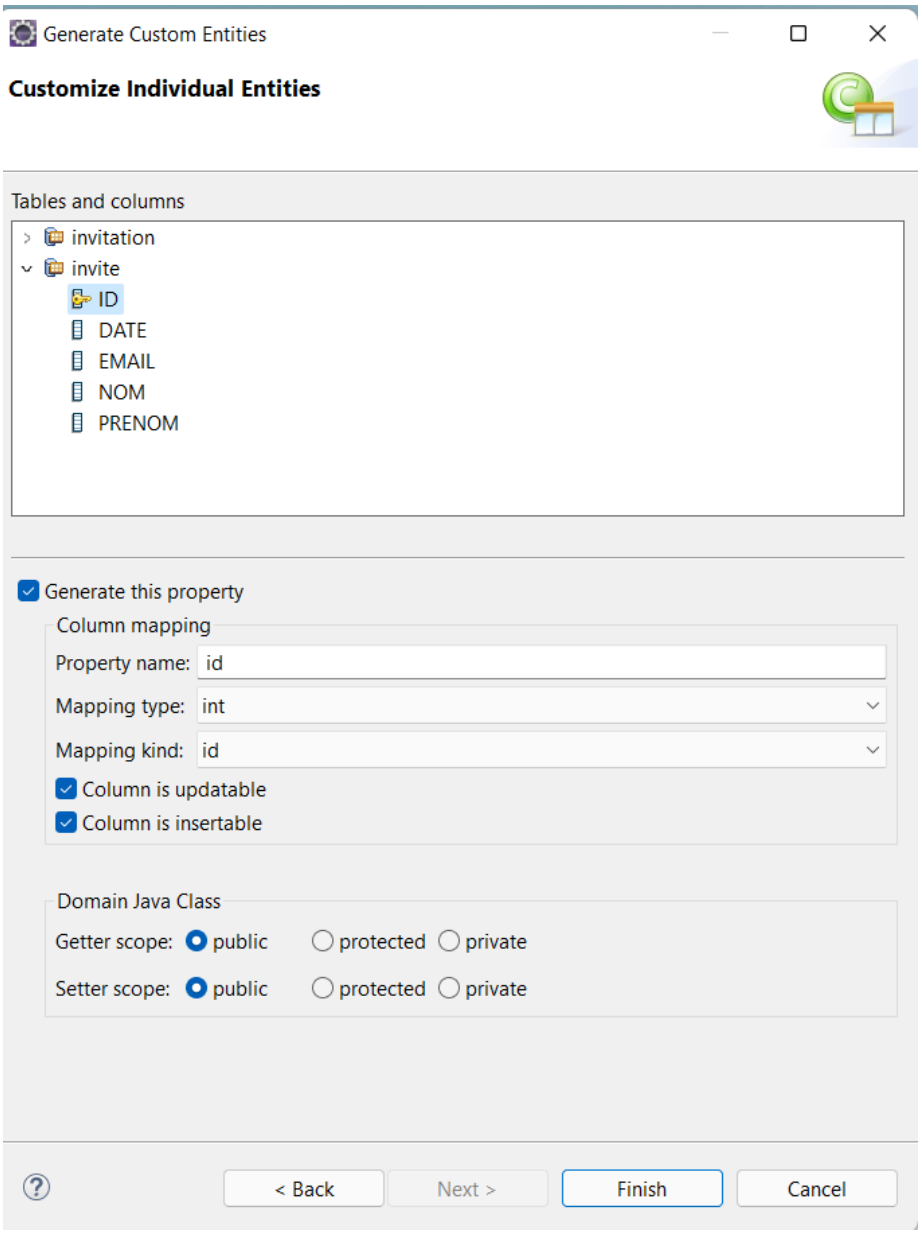
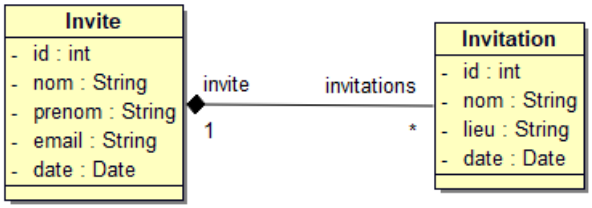


Diagramme de classes UML correspondant (sans les méthodes)

JPA génère un attribut (property) de type collection d'objets Invitations dans Invite, cela correspond en UML à la composition dessinée.
De même, il génère un attribut invite dans la classe Invitation



Les classes générées de type javabean, les attributs sont private et munis de leurs accesseurs, un constructeur sans argument est également créé (il ne faut pas le supprimer).

La classe Invitation générée

```
package model;

import java.io.Serializable;
import javax.persistence.*;
import java.util.Date;
/**
 * The persistent class for the invitation database table.
 */
@Entity
@NamedQuery(name="Invitation.findAll", query="SELECT i FROM Invitation i")
public class Invitation implements Serializable {
	private static final long serialVersionUID = 1L;

	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	private int id;
	@Temporal(TemporalType.DATE)
	private Date date;

	private String lieu;

	private String nom;

	//bi-directional many-to-one association to Invite
	@ManyToOne
	@JoinColumn(name="idinvite")
	private Invite invite;

	public Invitation() {
	}
	public int getId() {
		return this.id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public Date getDate() {
		return this.date;
	}
	public void setDate(Date date) {
		this.date = date;
	}
	public String getLieu() {
		return this.lieu;
	}
	public void setLieu(String lieu) {
		this.lieu = lieu;
	}
	public String getNom() {
```

```
		return this.nom;
	}
	public void setNom(String nom) {
		this.nom = nom;
	}
	public Invite getInvite() {
		return this.invite;
	}
	public void setInvite(Invite invite) {
		this.invite = invite;
	}
}
```

L'implémentation de l'association côté classe Invitation est réalisée par l'attribut :

```
	@ManyToOne
	@JoinColumn(name="idinvite")
	private Invite invite;
```

Ajoutons le constructeur avec arguments:

```
public Invitation(String nom, String lieu, Date date) {
	this.nom = nom ;
	this.lieu = lieu ;
	this.date = date ;
	}
```

La classe Invité générée

```
package model;

import java.io.Serializable;
import javax.persistence.*;
import java.util.Date;
import java.util.List;
/**
 * The persistent class for the invite database table.
*/
@Entity
@NamedQuery(name="Invite.findAll", query="SELECT i FROM Invite i")
public class Invite implements Serializable {
	private static final long serialVersionUID = 1L;
	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	private int id;
	@Temporal(TemporalType.DATE)
	private Date date;

	private String nom;
	private String prenom;
	//bi-directional many-to-one association to Invitation
	@OneToMany(mappedBy="invite")
	private List<Invitation> invitations;

	public Invite() {
```

```
	}
	public String getId() {
		return this.id;
	}
	public void setId(String id) {
		this.id = id;
	}

	public Date getDate() {
		return this.date;
	}
	public void setDate(Date date) {
		this.date = date;
	}
	public String getNom() {
		return this.nom;
	}
	public void setNom(String nom) {
		this.nom = nom;
	}
	public String getPrenom() {
		return this.prenom;
	}
	public void setPrenom(String prenom) {
		this.prenom = prenom;
	}
	public List<Invitation> getInvitations() {
		return this.invitations;
	}
	public void setInvitations(List<Invitation> invitations) {
		this.invitations = invitations;
	}
	public Invitation addInvitation(Invitation invitation) {
		getInvitations().add(invitation);
		invitation.setInvite(this);
		return invitation;
	}
	public Invitation removeInvitation(Invitation invitation) {
		getInvitations().remove(invitation);
		invitation.setInvite(null);
		return invitation;
	}
}
```

L'implémentation de l'association côté classe Invité est réalisée par l'attribut :

```
@OneToMany(mappedBy="invite")
	private List<Invitation> invitations;
```

L'association **OneToMany** de 1 vers plusieurs correspond à l'association "1 invité peut avoir plusieurs invitations", ces invitations seront alors stockées dans la liste proposée.

Ajoutons le constructeur avec arguments et la méthode toString().

```
	public Invite(String nom, String prenom, String email) {
```

```
                super();
                this.nom = nom ;
                this.prenom = prenom ;
                this.email = email ;
                date = new Date();
        }
        public  String toString(){
                return nom+" "+" "+prenom + "        " +
                        LocalDate.ofEpochDay(date.getTime()/(1000*3600*24))
                                .format(DateTimeFormatter.ofPattern("dd-MM-yyyy"));
        }
```

3ème étape

1er exemple d'une application (dans le package par défaut ou autre) L'exemple créé 2 objets managés invite et invitation.

```
import java.time.LocalDate;
import java.util.Date;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import model.Invitation;
import model.Invite;

public class Main1 {
        public static void main(String[] args) {
                 EntityManagerFactory factory = Persistence.
                                        createEntityManagerFactory("lesinvites");
                EntityManager em ;
                em = factory.createEntityManager();

                Invite invite = new Invite("magnum","rémi","r.magnum@orange.fr");

                em.getTransaction().begin();
                em.persist(invite);
                em.getTransaction().commit();

                LocalDate dateevenement = LocalDate.of(2022, 10, 11);
                Date date = new Date(24*3600*1000*dateevenement.toEpochDay());

                Invitation invitation = new Invitation("réunion sur la presse","toulouse",date);
                invitation.setInvite(invite);
                em.getTransaction().begin();
                em.persist(invitation);
                em.getTransaction().commit();
        }
}
```

A remplacer par le nom de votre projet JPA

Les opérations CRUD et les requêtes SQL doivent être exécutées par un **EntityManager**.

Un objet **EntityManagerFactory** est en 1er créé à l'aide de la méthode statique **createEntityManagerFactory()** de la classe **Persistence** qui reçoit comme argument par défaut le nom du projet JPA (ici **lesinvites**, à remplacer si nécessaire par le nom du projet).

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("lesinvites");
```

Nom du projet JPA!!!

L'objet **EntityManager** est ensuite créé avec la méthode **createEntityManager()** de la classe **EntityManagerFactory**.

```
private EntityManager em ;
em = factory.createEntityManager();
```

Instanciation d'un objet Invite, par exemple

```
Invite invite = new Invite("Durand","Robert","r.durand@gmail.com");
```

On rend l'objet persistant: il est enregistré dans la base de données.

```
em.getTransaction().begin();
em.persist(invite);
em.getTransaction().commit();
```

La méthode **persit()** correspond à l'opération **Create**.

La méthode **LocalDate.of(int année, int mois, int jourdumois)** sert à instancier un objet correspondant à la date de l'évènement. La méthode **toEpochDay()** retourne le nombre de jours de l'objet LocalDate concerné depuis le 1er janvier 1970. L'objet date est ensuite créé avec le constructeur Date(argument = temps en ms depuis le 1er janvier 1970).
Création de l'invitation.

```
Invitation i1 = new Invitation("Réunion sur la presse","Toulouse",date);
```

Enregistrement de l'invitation dans l'objet Invite.

```
i1.setInvite(i);
```

L'invitation est persistée.

```
em.persist(i1);
```

Remarque : les persistances de l'invité et de l'invitation peuvent être effectuées dans une même transaction:

```
Invitation invitation = new Invitation("Réunion sur la presse","Toulouse",date);
 invitation.setInvite(invite);
 em.getTransaction().begin();
 em.persist(invite);
 em.persist(invitation);
 em.getTransaction().commit();
```

Les opérations CRUD de base

Opération C – Create – qui consiste persister un enregistrement dans la base à partir d'un objet JPA Entity.

```
em.getTransaction().begin();
em.persist(invite);
em.getTransaction().commit();
```

Opération R – Read – pour extraire un enregistrement.

Exemple d'une méthode pour rechercher un enregistrement à partir de la clef.

```
public Invite recherche(int key) {
```

```
        Invite i ;
        em.getTransaction().begin();
        i = em.find(Invite.class, key);
        em.getTransaction().commit();
        return i ;
}
```

La méthode **find()** retourne **null** si l'enregistrement n'est pas trouvé.

Opération U – Update – pour mettre à jour un enregistrement.

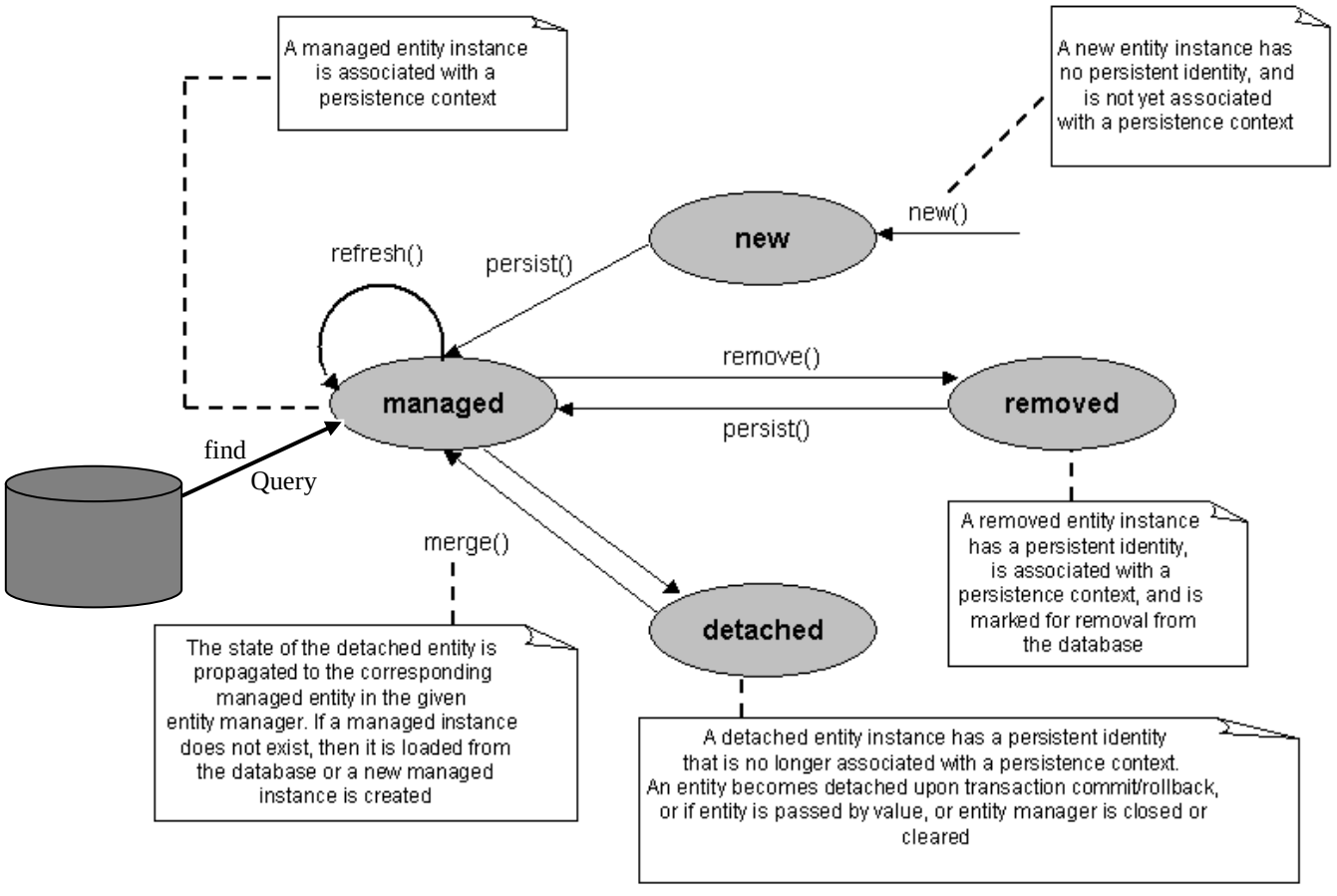
Exemple d'une mise à jour d'un invité : recherche par la clef et changement de son prénom.

```
public Invite majPrenom(String prenom, int key) {
        Invite i ;
        em.getTransaction().begin();
        i = em.find(Invite.class, key);
        i.setPrenom(prenom) ;
        em.getTransaction().commit();
        return i ;
}
```

Opération D – Delete – pour supprimer un enregistrement.

```
        em.getTransaction().begin();
        em.remove(i);
        em.getTransaction().commit();
```

4ème partie : cycle de vie d'un objet persistent



5ème partie : autres exemples

2ème exemple d'un client

Modification d'un attribut (le mail) d'une entité managée.

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;

import model.Invite;

public class Main2 {
	public static void main(String[] args) {
		EntityManagerFactory factory = Persistence.
				createEntityManagerFactory("lesinvites");
		EntityManager em ;
		em = factory.createEntityManager();

		Invite i = new Invite("mouton","alain","a.mouton@gmail.com");

		String requete = "SELECT g FROM Invite g where g.nom = " ;
		requete = requete+"'"+i.getNom()+"'"+"and"
				+" g.prenom = "+"'"+i.getPrenom()+"'";
		Invite i1 ;

		TypedQuery<Invite> query = em.createQuery(requete,Invite.class);
		if (query.getResultList().size()!=0){
			i1=query.getSingleResult();
			System.out.println("trouvé:"+i1.getNom()+" "+i1.getPrenom());
			em.getTransaction().begin();
			i1.setEmail(i.getEmail()) ;
			em.getTransaction().commit();
		}
		else System.out.println("invite "+i.getNom()+" pas trouvé");	}
}
```

Nouveau mail

3ème exemple : requête **paramétrée**, recherche de tous les invités avec Jean comme prénom

```
String queryString = "SELECT  i  FROM  Invite i  WHERE  i.prenom = :p";
Query query = em.createQuery(queryString);
query.setParameter("p", "Jean");
List<Invite> liste = query.getResultList();
for (Invite i : liste)
	System.out.println(i.getNom());
```

Paramètre nommé

Le paramètre est ici nommé, on peut également utilisé un paramétrage **numéroté**:

```
String queryString = "SELECT  i  FROM  Invite i  WHERE  i.prenom = ?1 ";
Query query = em.createQuery(queryString);
query.setParameter(1, "Jean");
```

4ème exemple : requête paramétrée, invitation de tous les invités avec Jean comme prénom à la fête des "Jean Jean"

```
import java.time.LocalDate;
import java.util.Date;
```

```
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;

public class Main3 {
	public static void main(String[] args) {
		EntityManagerFactory factory =
Persistence.createEntityManagerFactory("lesinvites");
		EntityManager em;
		em = factory.createEntityManager();

		LocalDate dateevenement = LocalDate.of(2022, 10, 11);
		Date date = new Date(24 * 3600 * 1000 * dateevenement.toEpochDay());

		String requete = "SELECT g FROM Invite g where g.prenom =:p";
		TypedQuery<Invite> query = em.createQuery(requete, Invite.class);
		query.setParameter("p", "Jean");
		List<Invite> liste = query.getResultList();
		liste.forEach(t->System.out.println(t));

		em.getTransaction().begin();
		for (Invite i : liste) {
			Invitation invitation = new Invitation("Fête des Jean Jean",
"Saint Jean de Luz", date);
			invitation.setInvite(i);
			em.persist(invitation);
		}
		em.getTransaction().commit();
	}
}
```

5ème exemple d'un client

Modification d'un attribut (le prénom) d'une entité avec fermeture puis ouverture de l'entity manager

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;

import model.Invite;

public class TestInviteJpa3 {
	public static void main(String[] args) {
	    EntityManagerFactory factory = Persistence.
				createEntityManagerFactory("jpa3-lesinvites");
		EntityManager em ;
		em = factory.createEntityManager();

		Invite i = new Invite("durand","robert");
		String requete = "SELECT g FROM Invite g where g.nom = " ;
		requete = requete+"'"+i.getNom()+"'"+"and"+
				" g.prenom = "+"'"+i.getPrenom()+"'";
		Invite i1=null ;
		TypedQuery<Invite> query = em.createQuery(requete,Invite.class);
		if (query.getResultList().size()!=0){
```

```
                        i1=query.getSingleResult();
                        System.out.println("trouvé:"+i1.getNom()+" "+i1.getPrenom());
                }
                else System.out.println("pas trouvé");
        // on ferme l'entity manager initial
                em.close();

                if (i1!=null) {
                        em = factory.createEntityManager();
                        i1.setPrenom("Paul");
                        Invite i2 = i1 ;
                        em.getTransaction().begin();
                        em.merge(i2);
                        em.getTransaction().commit();
                }
        }
}
```

Explications

L'entity manager initial est fermé

```
        em.close();
```

Instanciation d'un nouvel entity manager

```
        em = factory.createEntityManager();
```

Modification du prénom

```
        i1.setEmail("Paul");
```

Création d'un nouvel objet copie du premier (pas nécessaire, juste pour montrer que si i1 est managé alors toute copie de i1 est également managée)

```
        Invite i2 = i1 ;
```

Démarrage de la transaction

```
        em.getTransaction().begin();
```

Mise à jour avec merge

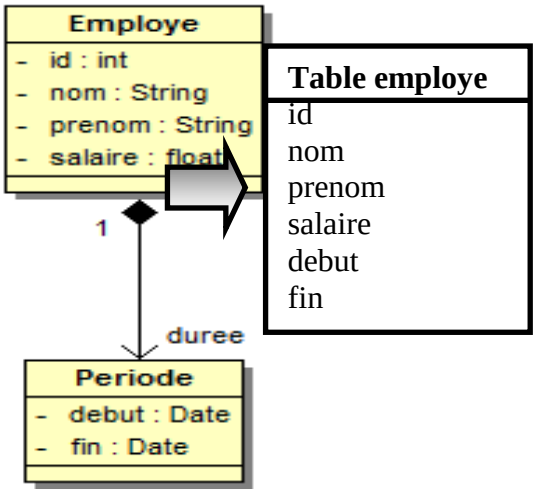
```
        em.merge(i2);
        em.getTransaction().commit();
```

6ème partie : compléments sur JPA

► Les attributs annotés @Transient ne sont pas persistants.

► Les classes Embeddable.

Une classe **Embeddable** ne correspond pas à une table dans la base, Un objet **Embeddable** ne peut pas être directement persisté dans une table correspondante.
Elle permet cependant de créer des attributs de type **Embedded** dans une entité qui eux correspondent à des champs de la table de l'entité.
Cela peut être intéressant dans le cas d'agrégation ou de composition UML.



Prenons l'exemple d'une entité **Employe** avec une classe **embeddable Periode**:
Les classes Java sont données en suivant.

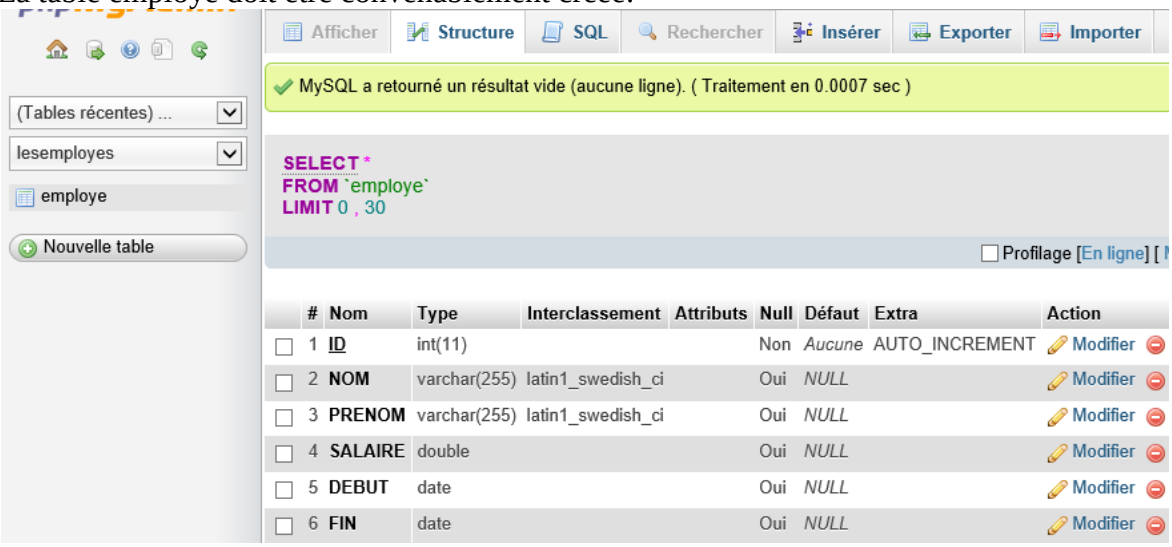
Création de l'application

1. créer la base de données *lesemployes*,
2. créer un projet JPA,
3. créer une connexion vers la base *lesemployes*,
4. modifier le fichier **persistence.xml**,
5. créer la classe **Periode** comme habituellement pour une classe java "ordinaire",
6. ajouter l'annotation **@Embeddable** devant **public class Periode**,
7. ajouter la gestion par JPA de la classe **Periode** dans le fichier **persistence.xml**,
8. créer la classe **Employe** comme dans la 1ère partie du cours,
9. ajouter les constructeurs avec arguments pour les 2 classes,
10. annoter **@Embedded** l'attribut de type **Periode** de la classe **Employe**:

```
@Embedded
private Periode periode ;
```

11. sélectionner le projet | clic droit | JPA tools | Generate Tables from Entities… | Finish

La table employe doit être convenablement créée:



On donne ici les classes Java de l'application:

```
package modele;

import java.io.Serializable;
import java.lang.Integer;
import java.lang.String;
import javax.persistence.*;

/**
 * Entity implementation class for Entity: Employe
 */
@Entity
public class Employe implements Serializable {

  @Id @GeneratedValue(strategy=GenerationType.IDENTITY)
  private Integer id;
  private String nom;
  private String prenom;
  private double salaire;
  @Embedded                              // à ajouter
  private Periode periode ;

  private static final long serialVersionUID = 1L;

  public Employe(String nom, String prenom, double salaire, Periode periode) {
          this.nom = nom;
          this.prenom = prenom;
          this.salaire = salaire;
          this.periode = periode;
  }
  public Employe() {
          super();
  }
  public Integer getId() {
          return this.id;
  }

  public void setId(Integer id) {
          this.id = id;
  }
  public String getNom() {
          return this.nom;
  }

  public void setNom(String nom) {
          this.nom = nom;
  }
  public String getPrenom() {
          return this.prenom;
  }
```

```
  public void setPrenom(String prenom) {
        this.prenom = prenom;
  }
  public double getSalaire() {
        return this.salaire;
  }
  public void setSalaire(double salaire) {
        this.salaire = salaire;
  }
}
```

```
package modele;

import java.sql.Date;
import javax.persistence.Embeddable;

@Embeddable          //annotée Embeddable, écrite comme une classe java normale
public class Periode {
  private Date debut ;
  private Date fin ;
public Periode(Date debut, Date fin) {
        this.debut = debut;
        this.fin = fin;
  }
public Periode() {
}
public Date getDebut() {
  return debut;
}
public void setDebut(Date debut) {
  this.debut = debut;
}
public Date getFin() {
  return fin;
}
public void setFin(Date fin) {
  this.fin = fin;
}
}
```

Programme de test

```
import java.time.LocalDate;
import java.sql.Date;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import modele.Employe;
import modele.Periode;

public class Test {
```

```
 public static void main(String[] args) {
          EntityManagerFactory factory = Persistence.
                                createEntityManagerFactory("employes");
          EntityManager em ;
          em = factory.createEntityManager();

          LocalDate datedebut = LocalDate.of(2015, 10, 11);
          Date debut = new Date(24*3600*1000*datedebut.toEpochDay());
          LocalDate datefin = LocalDate.of(2017, 12, 6);
          Date fin = new Date(24*3600*1000*datefin.toEpochDay());
          Periode duree = new Periode(debut,fin);

          Employe dupond = new Employe("Dupond","Jean",2580,duree);

          em.getTransaction().begin();
          em.persist(dupond);
          em.getTransaction().commit();
  }
}
```

►Les opérations en cascade

Par défaut, toute modification sur une entité n'entraine pas de modifications sur les autres. On peut changer ce comportement en modifiant le type **javax.persistence.CascadeType** des entités. Plusieurs possibilités:

Cascade Operations	Description
PERSIST	In this cascade operation, if the parent entity is persisted then all its related entity will also be persisted.
MERGE	In this cascade operation, if the parent entity is merged then all its related entity will also be merged.
DETACH	In this cascade operation, if the parent entity is detached then all its related entity will also be detached.
REFRESH	In this cascade operation, if the parent entity is refreshed then all its related entity will also be refreshed.
REMOVE	In this cascade operation, if the parent entity is removed then all its related entity will also be removed.
ALL	In this case, all the above cascade operations can be applied to the entities related to parent entity.

Exemple: on décide que la création d'une entité Individu associée à une nouvelle entité Invitation entraine la persistence de ces 2 entités.

La classe Invite modifiée:

```
import java.io.Serializable;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import javax.persistence.*;
```

```java
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

@Entity
@NamedQuery(name="Invite.findAll", query="SELECT i FROM Invite i")
public class Invite implements Serializable {
	private static final long serialVersionUID = 1L;

	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	private int id;
	@Temporal(TemporalType.DATE)
	private Date date;
	private String email;
	private String nom;
	private String prenom;

	//bi-directional many-to-one association to Invitation
	@OneToMany(mappedBy="invite", cascade=CascadeType.PERSIST)
	private List<Invitation> invitations;

	public Invite() {
		invitations = new ArrayList<Invitation>();
	}
	public Invite(String nom, String prenom, String email) {
		super();
		this.nom = nom;
		this.prenom = prenom;
		this.email = email;
		date = new Date();
		invitations = new ArrayList<Invitation>();
	}

	public String toString() {
		return nom + " " + " " + prenom + "    " +
LocalDate.ofEpochDay(date.getTime() / (1000 * 3600 * 24))
				.format(DateTimeFormatter.ofPattern("dd-MM-yyyy"));
	}
	public int getId() {
		return this.id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public Date getDate() {
		return this.date;
	}
	public void setDate(Date date) {
		this.date = date;
	}
	public String getEmail() {
		return this.email;
	}
	public void setEmail(String email) {
		this.email = email;
	}
	public String getNom() {
		return this.nom;
	}
	public void setNom(String nom) {
```

La persistence d'un invité entraîne la persistence de son invitation

Attention: il faut créer la liste d'invitations

```
		this.nom = nom;
	}
	public String getPrenom() {
		return this.prenom;
	}
	public void setPrenom(String prenom) {
		this.prenom = prenom;
	}
	public List<Invitation> getInvitations() {
		return this.invitations;
	}
	public void setInvitations(List<Invitation> invitations) {
		this.invitations = invitations;
	}
	public Invitation addInvitation(Invitation invitation) {
		getInvitations().add(invitation);
		invitation.setInvite(this);
		return invitation;
	}
	public Invitation removeInvitation(Invitation invitation) {
		getInvitations().remove(invitation);
		invitation.setInvite(null);
		return invitation;
	}
}
```

Le programme de test

```
import java.time.LocalDate;
import java.util.Date;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class Main1 {
	public static void main(String[] args) {
		 EntityManagerFactory factory = Persistence.
				createEntityManagerFactory("lesinviteslazy");
		EntityManager em ;
		em = factory.createEntityManager();

		Invite i1 = new Invite("Groscolas","Jean","j.groscolas@orange.fr");

		LocalDate dateevenement = LocalDate.of(2022, 5, 12);
		Date date = new Date(24*3600*1000*dateevenement.toEpochDay());

		Invitation invitation = new Invitation("Salon de la
voile","Brest",date);
		i1.addInvitation(invitation);
		em.getTransaction().begin();
		em.persist(i1);
		em.getTransaction().commit();
	}
}
```

Propagation de la persistence et de la suppression d'une invitation

```
  //bi-directional many-to-one association to Invitation
  @OneToMany(mappedBy="invite", cascade= {CascadeType.PERSIST,CascadeType.REMOVE})
  private List<Invitation> invitations;
```

►L'héritage

L'héritage en SQL n'existe pas.

Voir le lien: https://wiki.eclipse.org/EclipseLink/UserGuide/JPA/
Basic_JPA_Development/Entities/Inheritance

Il y a 3 stratégies pour mettre en œuvre l'héritage avec JPA.

SINGLE_TABLE: toutes les classes de la hiérarchie sont persistées dans une seule table. La table doit contenir une colonne de discrimination (@DiscriminatorColumn) qui identifie la sous-classe de chaque enregistrement (un enregistrement par ligne).

TABLE_PER_CLASS: à chaque classe correspond une table. Toutes les propriétés d'une classe, incluant les propriétés héritées, correspondent à des colonnes de la table.

JOINED: la classe mère est représentée par une table et chaque sous-classe par une table séparée. Chaque table de sous-classe ne contient que les colonnes correspondantes aux propriétés propres (non héritées) de la sous-classe. La colonne clef primaire des tables des sous-classes sert de clef étrangère liée à la clef primaire de la table de la super classe.

La colonne de discrimination @DiscriminatorColumn

La colonne **@DiscriminatorColumn** est nécessaire pour les stratégies SINGLE_TABLE et JOINED.
Elle apparait dans la seule table pour SINGLE_TABLE et dans la table de la super classe pour JOINED.
Si l'annotation **@DiscriminatorColumn** n'explicite pas précisément une colonne, cette colonne par défaut s'appelle **DTYPE**, est de type **String** et contiendra le nom de la classe fille pour chaque enregistrement créé associé.

Exemple d'une stratégie JOINED

1) Le modèle des classes

Création d'un projet JPA, ici de nom heritage.
La super classe

```
package heritage;

import java.io.Serializable;
import java.lang.String;
import javax.persistence.*;

/**
 * Entity implementation class for Entity: Personne
*/
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Personne implements Serializable {
```

```
	@Column(name="id")
	@Id @GeneratedValue(strategy = GenerationType.IDENTITY)
	private int id;
	private String nom;
	private static final long serialVersionUID = 1L;
	public Personne() {
		super();
	}
	public Personne(String nom) {
		super();
		this.nom = nom ;
	}
	public int getId() {
		return this.id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getNom() {
		return this.nom;
	}
	public void setNom(String nom) {
		this.nom = nom;
	}
}
```

L'annotation @Inheritance(strategy=InheritanceType.***JOINED***) portée sur la classe précise la stratégie utilisée.

La sous-classe Etudiant

```
package heritage;

import heritage.Personne;
import java.io.Serializable;
import java.lang.String;
import javax.persistence.*;

/**
 * Entity implementation class for Entity: SousTraitant
*/
@Entity
@PrimaryKeyJoinColumn(name = "id", referencedColumnName = "id")
public class Etudiant extends Personne implements Serializable {
	private int id;
	private int age ;
	private static final long serialVersionUID = 1L;

	public Etudiant() {
		super();
	}
```

```
	public Etudiant(String nom, int age) {
		super(nom);
		this.age = age ;
	}
	public int getId() {
		return this.id;
	}

	public void setId(int id) {
		this.id = id;
	}
	public int  getAge() {
		return this.age;
	}

	public void setAge(int age) {
		this.age = age;
	}

}
```

L'annotation portée @PrimaryKeyJoinColumn(name = "id", referencedColumnName = "id") sur la classe précise la colonne de jointure.

On ne définit pas de clef primaire pour les sous-classes (pas d'annotation @Id).

Si l"annotation @PrimaryKeyJoinColumn(name…) n'est pas indiquée, JPA crée une colonne clef étrangère dans la table sous-classe de même nom que la clef primaire de la table super classe.

La sous-classe Professeur

```
package heritage;

import heritage.Personne;
import java.io.Serializable;
import java.lang.String;
import javax.persistence.*;

/**
 * Entity implementation class for Entity: Employe
*/
@Entity
@PrimaryKeyJoinColumn(name = "id", referencedColumnName = "id")
public class Professeur extends Personne implements Serializable {
	private int id;
	private String matiere;
	private static final long serialVersionUID = 1L;

	public Professeur() {
		super();
	}
```

```
	public Professeur(String nom, String matiere) {
		super(nom);
		this.matiere = matiere ;
	}
	public int getId() {
		return this.id;
	}

	public void setId(int id) {
		this.id = id;
	}
	public String getMatiere() {
		return this.matiere;
	}

	public void setMatiere(String matiere) {
		this.matiere = matiere;
	}
}
```

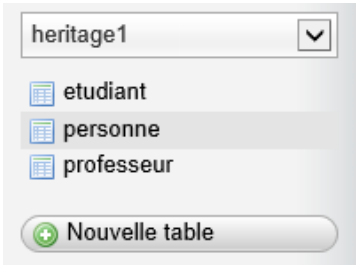
2) Création des tables

Créer la connexion à la base.
Modifier le fichier persistence.xml pour indiquer la connexion utilisée.
Clic droit sur le projet | JPA Tools | Generate Tables from Entities …

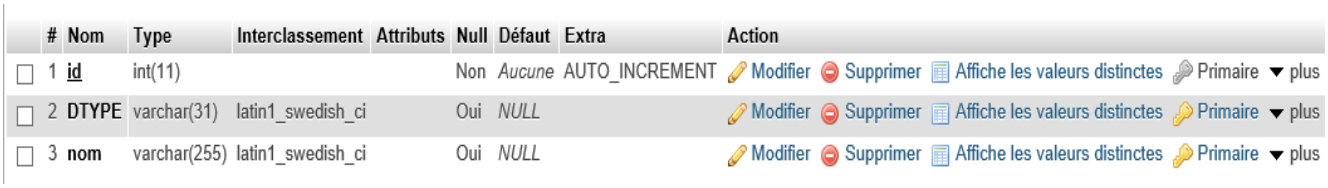
On peut observer les commandes sql générées:

```
[EL Fine]: sql: Connection(278536229)--CREATE TABLE PERSONNE (id INTEGER
AUTO_INCREMENT NOT NULL, DTYPE VARCHAR(31), nom VARCHAR(255), PRIMARY KEY (id))
[EL Fine]: sql: Connection(278536229)--CREATE TABLE ETUDIANT (id INTEGER NOT NULL,
AGE INTEGER, PRIMARY KEY (id))
[EL Fine]: sql: Connection(278536229)--CREATE TABLE PROFESSEUR (id INTEGER NOT NULL,
MATIERE VARCHAR(255), PRIMARY KEY (id))
[EL Fine]: sql: Connection(278536229)--ALTER TABLE ETUDIANT ADD CONSTRAINT
FK_ETUDIANT_id FOREIGN KEY (id) REFERENCES PERSONNE (id)
[EL Fine]: sql: Connection(278536229)--ALTER TABLE PROFESSEUR ADD CONSTRAINT
FK_PROFESSEUR_id FOREIGN KEY (id) REFERENCES PERSONNE (id)
```

Les 3 tables sont bien créées:



La super classe Personne:



#	Nom	Type	Interclassement	Attributs	Null	Défaut	Extra	Action
1	id	int(11)			Non	*Aucune*	AUTO_INCREMENT	Modifier Supprimer Affiche les valeurs distinctes Primaire plus
2	DTYPE	varchar(31)	latin1_swedish_ci		Oui	*NULL*		Modifier Supprimer Affiche les valeurs distinctes Primaire plus
3	nom	varchar(255)	latin1_swedish_ci		Oui	*NULL*		Modifier Supprimer Affiche les valeurs distinctes Primaire plus

On constate bien la création de la colonne DTYPE.

2) Ajout d'enregistrement

```
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

import heritage.Etudiant;
import heritage.Professeur;

public class Main {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		EntityManagerFactory factory =
				Persistence.createEntityManagerFactory("heritage");
		EntityManager em;
		em = factory.createEntityManager();
		Professeur p1 = new Professeur("Bellelettre","Français");
		Professeur p2 = new Professeur("Saitout","Maths");
		Etudiant e1 = new Etudiant("Michon",21);
		Etudiant e2 = new Etudiant("Michou",22);
		em.getTransaction().begin();
		em.persist(p1);
		em.persist(p2);
		em.persist(e1);
		em.persist(e2);
		em.getTransaction().commit();
	}
}
```

Résultats :

La table super classe Personne

	id	DTYPE	nom
Modifier Copier Effacer	1	Etudiant	Michou
Modifier Copier Effacer	2	Etudiant	Michon
Modifier Copier Effacer	3	Professeur	Bellelettre
Modifier Copier Effacer	4	Professeur	Saitout

La table sous-classe Etudiant

	id	AGE
Modifier Copier Effacer	1	22
Modifier Copier Effacer	2	21

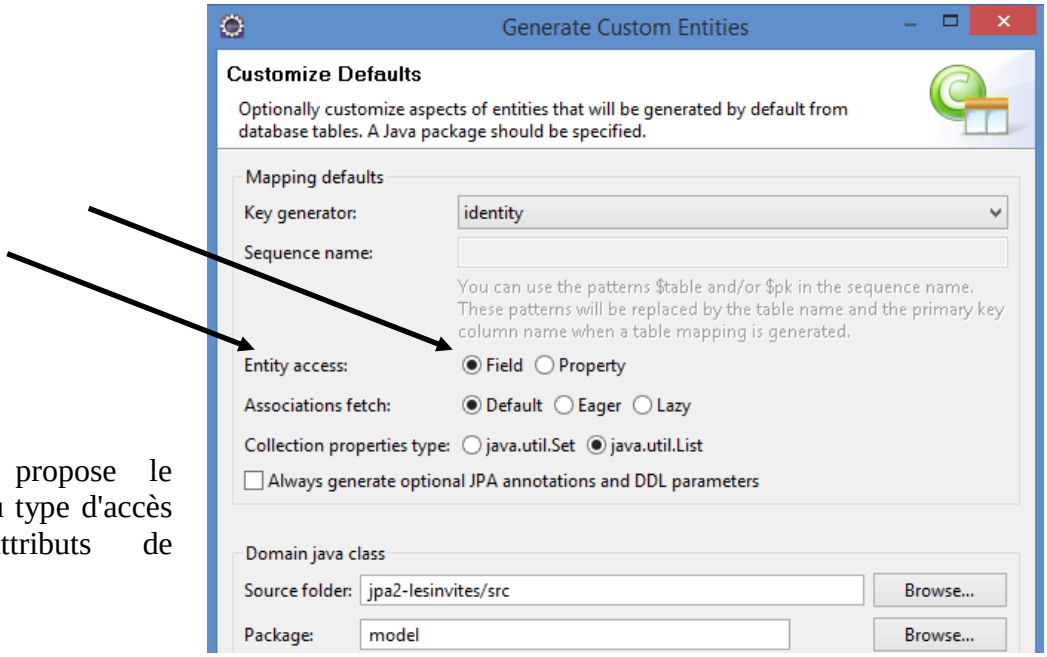
La table sous-classe Professeur

	id	MATIERE
Modifier Copier Effacer	3	Français
Modifier Copier Effacer	4	Maths

►JPA propose 2 méthodes pour accéder aux attributs d'une instance :

- soit JPA accède directement aux attributs de l'objet, c'est l'accès par **Field**,
- soit JPA accède aux attributs en passant par les accesseurs, c'est l'accès par **Property**.

Eclipse propose le choix du type d'accès aux attributs de l'objet.



L'intérêt de l'accès par **Property** (par les accesseurs) réside dans la possibilité d'ajouter des algorithmes dans les accesseurs lors de la correspondance entre les attributs et les champs de la table concernée.

La position de l'annotation @Id indique le type d'accès choisi :

- Accès par **Field**

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

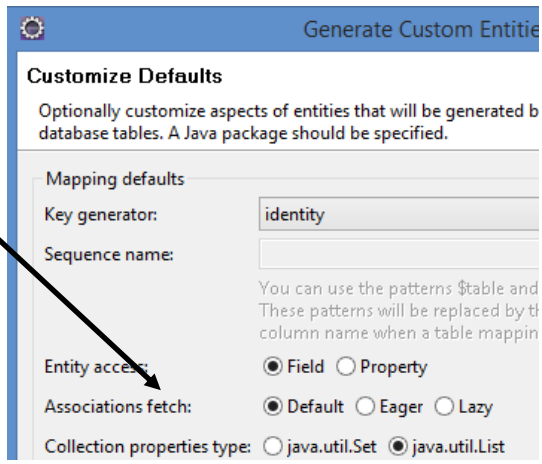
L'annotation @Id est posée devant l'attribut Id.

- Accès par **Property**

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
public String getId() {
        return this.id;
}
```

L'annotation @Id est posée devant l'accesseur getId().

► **Association Eager ou Lazy**, voir la figure ci-contre, le choix **Association fetch.**



Un objet entité -*invite ou invitation* - chargé par l'application cliente contient des attributs objets "association" correspondant aux associations entre les classes:

- un *invite* contient un attribut objet "liste des invitations",
  private List<Invitation> invitations;
- une *invitation* contient un attribut objet "invité".
  private Invite invite;

Les attributs "objets association" correspondent à des enregistrements dans la base de données.
Comment JPA gère t-il ces attributs "association" quand, par exemple, un client récupère un objet entité *invite* ou *invitation* ?

Il a y 2 possibilités.

- Le choix **"fetch Lazy"** ne force pas la récupération des attributs "objets association" lorsque l'application cliente récupère un objet entité *invite* ou *invitation*. Il faut alors faire une requête supplémentaire à la base pour récupérer ces objets si on le souhaite.
  Ce qui veut dire que dans notre exemple on récupère une instance d'invité sans avoir la liste d'invitations remplie.

- Le choix **"fetch Eager"** force le chargement des attributs "objets association" lorsque l'application cliente récupère un objet entité *invite* ou *invitation*.
  La lecture d'un objet invité entraine alors automatiquement l'initialisation de l'attribut
  List<Invitation> invitations
  avec sa liste d'invitations.

Le choix **"fetch Eager"** est identifié dans le code au niveau des attributs créés pour satisfaire les associations.

```
public class Invite implements Serializable {
  private static final long serialVersionUID = 1L;
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private int id;
  @Temporal(TemporalType.DATE)
  private Date date;
  private String nom;
  private String prenom;

  //bi-directional many-to-one association to Invitation
```

```
@OneToMany(mappedBy="invite", fetch=FetchType.EAGER)
private List<Invitation> invitations;
----------------
```

Il en est de même dans la classe **Invitation** si on choisit **Eager** pour les attributs créés pour satisfaire les associations:

```
public class Invitation implements Serializable {
  private static final long serialVersionUID = 1L;
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private int id;
  @Temporal(TemporalType.DATE)
  private Date date;
  private String nom;

  //bi-directional many-to-one association to Invite
  @ManyToOne(fetch=FetchType.EAGER)
  @JoinColumn(name="idinvite")
  private Invite invite;
```

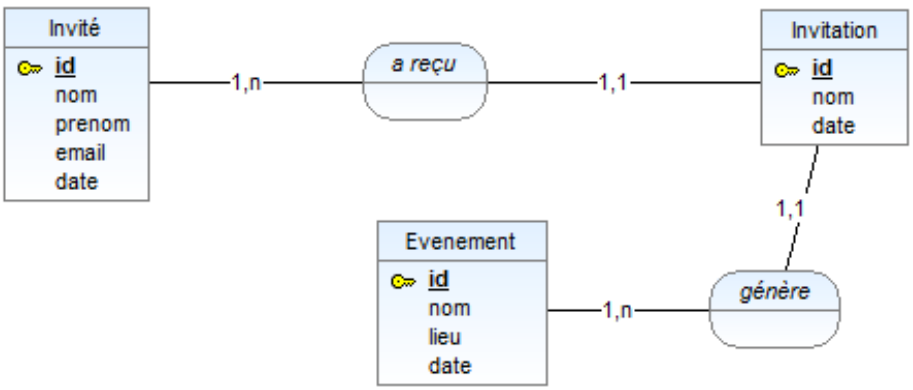
6ème partie : les exercices

1er exercice

Proposer un modèle cohérent UML d'une classe DAO pour la base lesinvites (exemple développé dans les 3ème et 5ème parties.
Coder cette classe DAO. Tester.

2ème exercice

On propose un MCD enrichi par-rapport à application étudiée.



Un événement génère plusieurs invitations.
Un invité ne peut recevoir qu'une seule invitation pour un événement donné.
Un invité peut être invité à plusieurs événements.

Le nom de l'invitation peut être différent du nom de l'événement.
Un invité est enregistré à une certaine date.

Une invitation est créée à une certaine date.
Un événement se produit à une certaine date.

1. Créer une base manifestations. Créer les 3 tables. Faire apparaître dans les 3 tables les associations.
2. Créer un nouveau projet JPA.
3. Générer les **entities jpa** à partir des 3 tables.
4. Créer une application simple mettant en œuvre les classes précédentes pour créer quelques enregistrements.
5. Coder une classe DAO. Coder une application simple mettant en œuvre la classe DAO.

3ème exercice

Reprendre les cours sur JavaFX et SceneBuilder développant l'application Agenda.
Créer une base de données avec une table utilisateur et une table agenda.
Ajouter à l'application Agenda la couche DAO en utilisant JPA.
Trouver une solution simple pour l'identification de l'utilisateur.