TP2 : Persistance des données avec JPA

Contenu

[1. Le JPA : 2](#_Toc307261229)

[1.1. Définition : 2](#_Toc307261230)

[1.2. Création de la couche JPA : 2](#_Toc307261231)

[1.3. Avantages et inconvénients : 5](#_Toc307261232)

[2. La DAO : 5](#_Toc307261233)

[2.1. Définition : 5](#_Toc307261234)

[2.2. Mise en place de la DAO : 6](#_Toc307261235)

Après avoir réalisé lors du précédent TP un accès à la base de données utilisant JDBC, l’objectif de ce TP est de réaliser une couche supplémentaire, le JPA. J’expliquerai pour ce TP dans un premier temps la réalisation de la persistance avec JPA, puis la réalisation des méthodes demandées dans le DAO.

# Le JPA :

## 1.1. Définition :

La Java Persistence API (abrégée en JPA), est une interface de programmation Java permettant aux développeurs de créer un pont entre le monde relationnel de la base de données et le monde objet manipulé par le langage Java. Le pont peut se faire grâce à des fichiers Xml, ou, depuis le JDK 1.5, par des annotations java. Le JPA permet donc un mapping d’un objet avec une ou plusieurs tables de la base donnée.

Elle permet également de manipuler des entités pour réaliser les opérations de persistance comme la récupération et la mise à jour d’objet. Les entités sont des classes qui représentent un ensemble de données de la base et récupérer sous la forme d’un tout.

JPA fournit également un langage de requête standard pour la récupération d’objet : le JP QL. Inspiré du SQL, il permet de réaliser des requêtes pour les entités sur les bases de données.

Il existe 3 types de relations entre entité :

- le one-to-one : une entité est lié à une et une seule entité.

- le many-to-one : plusieurs entités peuvent être reliées à une autre.

- le many-to-many : plusieurs entités peuvent être reliées à plusieurs autres.

Le mapping doit pouvoir gérer ces types de relations.

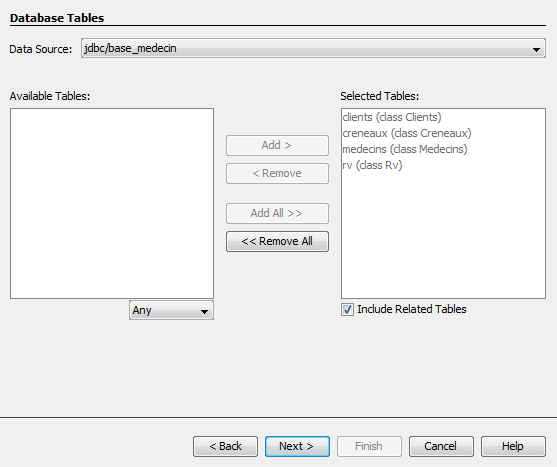
## 1.2. Création de la couche JPA :

Tout d’abord, il faut créer la JPA. Elle se fait automatiquement en créant une nouvelle « Persistance Unit ». On choisit comme fournisseur de persistance « Eclipse Link ». Eclipse Link est un framework de mapping objet-relationnel qui créé des correspondances entre notre base de données et les objets java. Les autres outils de mapping utilisés couramment sont Hibernate et TopLink.

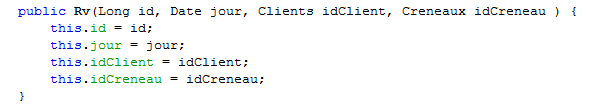
On utilise également les transactions Java API. C’est une interface Java standard entre un gestionnaire de transaction et les différentes parties impliquées dans un système de transactions distribuées : le gestionnaire de ressources, le serveur d'application et les applications transactionnelles.

Une fois terminé, un fichier «persistence.xml» est créé. Celui-ci décrit l’ensemble des propriétés de notre JPA, comme la base utilisée ou le type de transaction.

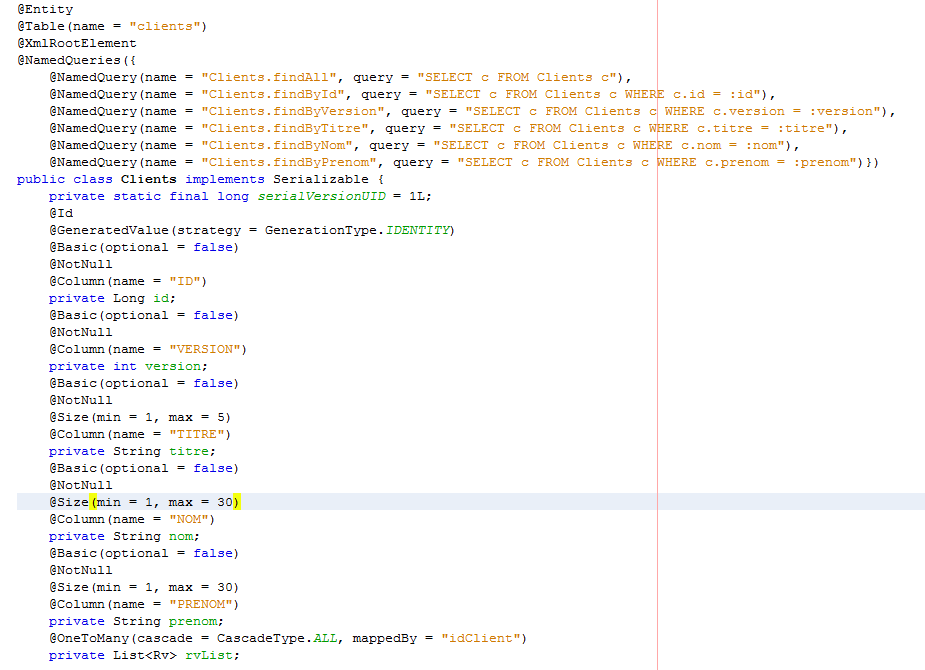
Après la création de la JPA, on crée les entités JPA. Pour cela, on utilise l’interface utilisateur de NetBeans. On choisit « Classe d’entité de la base de données », puis on choisit les classes d’entités que l’on veut générer en fonction de la base choisie.



Une fois choisie, on crée un nouveau package « JPA », puis on décide de générer les requêtes nommée pour les champs persistants. On a donc après ces étapes la création de 4 classes : Clients.java, Creneaux .java, Medecin.java et RV.java. Une grande partie du code a été généré, mais il faut toutefois compléter la classe RV car le constructeur avec l’id du médecin et l’id du créneau n’a pas été construit.



Voici ci-dessus un exemple de code générer.

**

Ceci représente un morceau de la classe Clients. Des annotations java permettent de réaliser le mapping avec la base de données :

@Entity : Permet de définir la classe comme entité.

@Table : Désigne la table correspondante à cette entité.

@nameQueryies : Correspond aux requêtes nommées.

@id : Indique quel champ de la classe est la clé primaire.

@GenerateValue : Permet de générer la clé selon une stratégie définie.

@NotNull : Le champ ne peut pas être nul.

@Column : Permet de lier le champ à une colonne de la table.

@Size : Définie la taille du champ.

@one-to-many : Définit quel type de relation il existe entre deux entités. Par exemple, dans ce cas, un client peut être liés à un rendez-vous (et un rendez-vous ne peut avoir qu’un seul client). On a donc une relation de type on-to-many lié sur l’idClient.

@basic : Définie la stratégie de récupération des données. La stratégie EAGER permet de charger les données directement alors que le Lazy loading consiste à charger des objets uniquement lorsque l'on y accède explicitement. Pour les collections (relations one-to-many et many-to-many) la valeur par défaut est LAZY mais pour les objets contenus dans un autre (relations many-to-one) la valeur par défaut est EAGER ce qui signifie que tout sous-objet est donc chargé.

## 1.3. Avantages et inconvénients :

JPA permet de réaliser plus facilement une couche. Les requêtes sont plus simples, et les erreurs remontées en cas de modification de la base sont plus simple à gérer. Elle donne également une grande stabilité et une standardisation, puisqu’il n’est pas nécessaire de changer de DAO en fonction de l’outil de mapping utilisé.

L’inconvénient de JPA est qu’elle est plus lourde que l’utilisation de JDBC uniquement. JPA étant de plus haut niveau que JDBC, les performances seront donc réduites.

Le choix ou non de son utilisation doit être fait selon l’utilisation de la base de données. Pour une utilisation intensive de la base de données, JDBC doit être utilisé, sinon on utilise JPA pour le gain de temps lors du développement.

# La DAO :

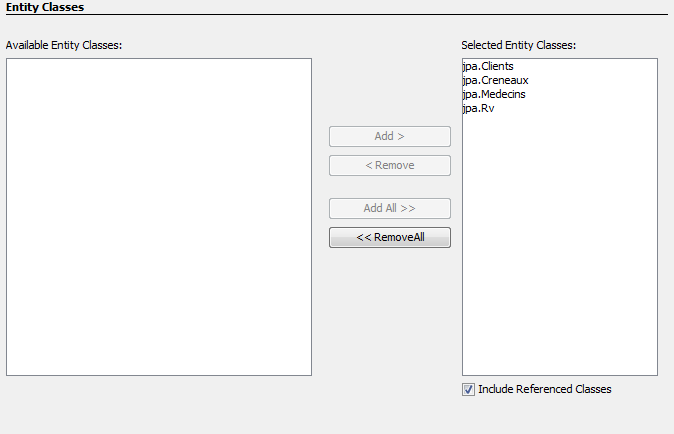
## Définition :

La DAO est une couche supplémentaire correspondant à la couche métier. Il permet de séparer la manipulation des objets dans l’application et les échanges avec la base de données comme la récupération de données ou la mise à jour.

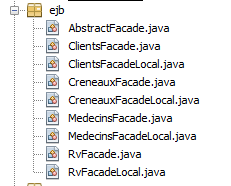
La DAO va utiliser un Entity Manager. Celui-ci est une interface qui permet de communiquer avec la persistance. On peut donc utiliser un ensemble de fonction que l’on peut utiliser dans le code métier pour interagir avec notre JPA.

## Mise en place de la DAO :

On va d’abord créer le « Session Bean » pour les classes d’entités grâce à l’interface. On choisit ensuite les classes d’entités JPA que l’on veut utiliser avec à notre DAO.



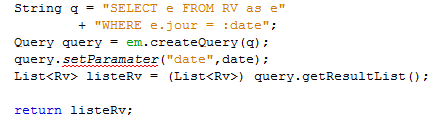
Une fois les classes choisies, on entre le nom de notre package, puis on choisit de créer l’interface Local. On a donc la création de 7 classes.



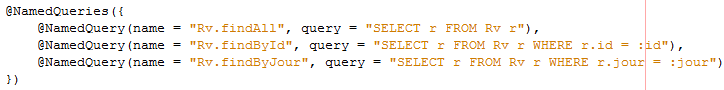
La classe AbstractFacade possède un ensemble de méthode commune à toutes les classes du DAO. C’est une méthode générique avec les comportements semblables des objets, comme la recherche par rang, la création, la mise à jour, la recherche ou le nombre de résultats.Il y a ensuite pour chaque objet une classe « facade » qui dérive d’AbstractFacade et qui implémente une interface « FacadeLocal ». Par exemple, pour l’entité Client, on a une classe ClientFacade, et une interface ClientFacadeLocal. Les façades utilisent un entity Manager.

On crée ensuite l’EJB session. C’est un objet proposant des services permettant d’utiliser les objets crées précédemment. Avec l’interface utilisateur, on choisit le nom de l’EJB. On choisit s’il est Stateless ou Stateful. La particularité principale d'un Stateful Session Bean est de conserver son état entre différents appels de méthodes, contrairement au Stateless Session Bean qui ne le conserve pas. On choisit pour ce tp un Stateless pour éviter que le serveur conserve un état pour chaque connexion, c’est qui peut être lourd en cas de connexions nombreuses. On a donc créé une classe « DaoJpa» qui implémente une interface « DaoJpaLocal » qui va permettre de réaliser des méthodes plus complètes comme getRvByAll() ;

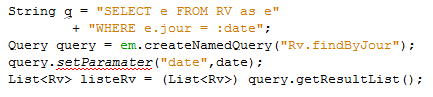
Il existe plusieurs façon de réaliser la méthode getRvbyAll(). La première consiste à réaliser une requête nommée qui sera utilisée par le JPA grâce à l’Entity Manager (ici em).



On peut aussi créer la requête dans les entités JPA :



Puis les appeler ensuite dans le DAO :



Ces méthodes pourront ensuite être appelées par les webServices utilisant notre EJB.