TP3 : Le conteneur EJB

Contenu

[1. L‘EJB: 2](#_Toc310376489)

[1.1. Définition : 2](#_Toc310376490)

[1.2. Création de l’EJB : 3](#_Toc310376491)

[1.3. Déploiement de l’EJB : 4](#_Toc310376492)

[2. Création d’un webService de test : 4](#_Toc310376493)

Après avoir réalisé lors du précédent TP la couche JPA pour permettre la persistance des données, nous devons réaliser une nouvelle couche afin de permettre aux clients d’accéder à des services. Nous allons ensuite tester ce conteneur à l’aide d’un WebService.

# L‘EJB:

## Définition :

L’EJB (Entreprise Java Bean) est un composant serveur pour les plateformes JEE. Il permet d’implémenter la logique métier de notre application. Il est accessible par les clients et s’exécute dans un conteneur EJB. Il permet de réaliser facilement une application multicouche. Le client permettra d’afficher la présentation, et l’EJB servira de logique métier, grâce à la proposition de service. Le serveur prend en charge la création et la destruction des EJB.

Depuis la version 3.0, les fichiers de configurations ne sont plus obligatoires, et les annotations java permettent de spécifier entièrement la configuration et les propriétés de l’EJB.

Il existe deux modes de gestion d’EJB :

* Stateless : L’état de l’EJB n’est pas conservé entre 2 appels. Il est partagé par plusieurs clients. Il supporte donc les montées en charge.
* Stateful : L’état est conservé pendant la session Bean-client. Il n’y a donc qu’un seul client à la fois.

Le choix du mode dépend de l’application (nombre d’utilisateur, besoin des données pendant l’ensemble de la transaction…). Dans notre cas, on utilisera un mode Stateless. En effet,

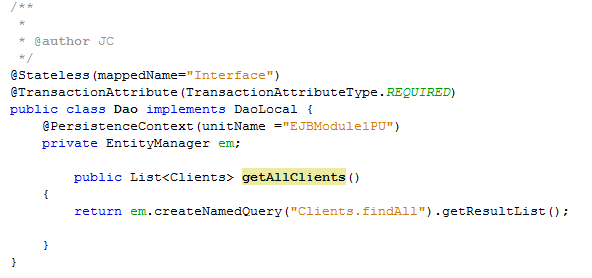
* Un grand nombre d’utilisateur peut utiliser ces services en même temps. Il faut donc avoir une bonne montée en charge.
* Nous n’avons pas besoin de garder les informations du client pendant l’ensemble de la transaction.

L’EJB utilisera ici la persistance que l’on a mise en place lors du Tp précédent. Nous pourrons donc utiliser les requêtes nommées créée précédemment.

Le client va rechercher l’EJB grâce à son nom logique JNDI. JNDI(Java Naming and Directory Interface) est une API Java de connexions à des annuaires qui permet de lier un nom à une information.

## 1.2. Création de l’EJB :

La création de l’EJB se fait par l’interface graphique de Netbeans. On choisit d’ajouter un nouveau « Session Bean », en choisissant si la session doit être Stateless ou Statefull. Une fois créé, on a donc une classe Dao et son interface, DaoLocal. On complète donc cette classe afin d’ajouter la méthode getAllClients retournant tout les clients.



Tout d’abord, l’annotation @stateless permet de dire que la classe Dao est un bean de type Stateless.

@TransactionAttribute permet de configurer comment le container gère les transactions quand un client invoque une méthode. Le TransactionAttributeType.REQUIRED correspond à deux propriétés :

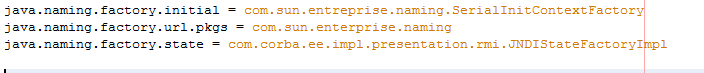
* Le bean est toujours dans une transaction.
* Si l’appelant est dans une transaction, alors le bean la rejoint, sinon, le container crée une nouvelle transaction.

L’EJB 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. L’annotation @PersistenceContext permet de définir avec quelle persistance l’Entity Manager va interagir. EntityManager va, pour la méthode getAllClients, appeler la requêtes nommées Clients.findAll créé précédemment dans la JPA et retourner le résultat sous forme de liste pour pouvoir être traité par les clients qui utiliseront cette méthode.

## 1.3. Déploiement de l’EJB :

Après avoir créé notre DAO, on doit déployer notre EJB pour qu’il puisse être utilisé par un WebService. Tout d’abord, on ajoute les librairies permettant de faire fonctionner l’EJB, c’est-à-dire les librairies EclipseLink et MySQL.

On ajoute ensuite un fichier de configuration permettant de configurer le JNDI. Il contient 3 lignes :



La valeur des paramètres correspondent à ceux d’un serveur GlassFish. Ils ont des valeurs différentes si on utilise un autre serveur.

Il permet de définir le contexte JNDI :

Java.naming.factory.initial : donne le registre du fournisseur de service dans le contexte initial.

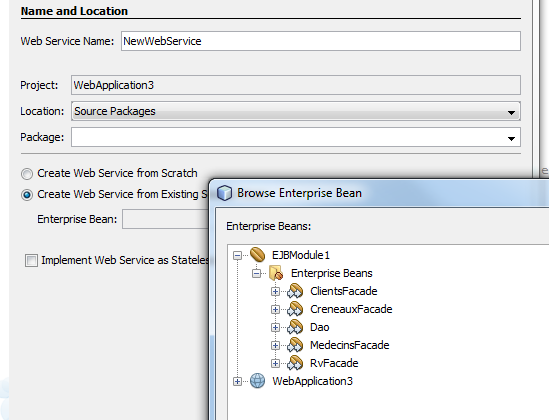
TODO

Une fois le fichier créé, on peut déployer l’EJB. Une fois déployé, on obtient un jar qui pourra être utilisé par un webService.

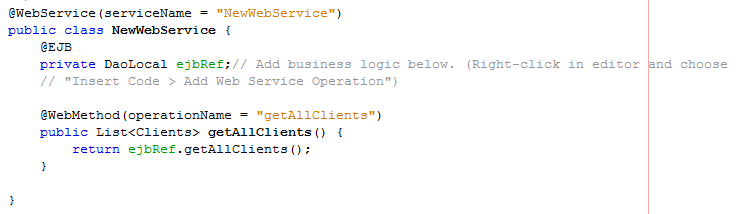
# Création d’un webService de test :

On créé d’abord un nouveau projet WebApplication. Une fois créé, on ajoute le Jar créé lors de l’étape précédente afin de pouvoir utiliser les beans créés précédemment.

On peut donc créé le WebService s’appuyant sur notre Bean créé dans l’EJB. On choisit le Bean « DAO ». Le webService est ensuite créé automatiquement.



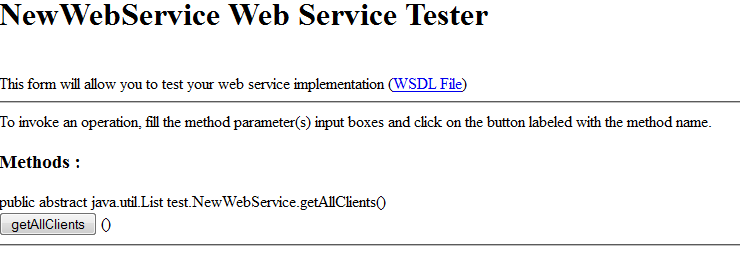
On a donc la création du WebService.



L’annotation @WebService permet de définir que cette classe est un WebService, ainsi que son nom. @EJB permet de définir notre DaoLocal est l’EJB qui sera utilisé par le WebService. La méthode getAllClients sera une WebMethod, comme l’indique l’annotation. Celle-ci ne fait que renvoyer le contenu de la méthode getAllClients de l’EJB.

Une fois créé, on peut tester le webService en cliquant droit dessus et en choisissant « Tester le webService ». On ouvre donc notre navigateur Internet à l’adresse :

<http://localhost:8080/WebApplication3/WebService?Tester> avec le port et l’adresse du serveur, la WebApplication et le WebService. Une page recense les méthodes et permet de les appeler.



En cliquant sur getAllClients(), on affiche l’appel au WebService et la réponse.



L’appel se fait grâce au Protocol SOAP, qui permet la transmission de messages entre objets distants ? On a donc l’adresse du package (ici « test ») et le nom de la méthode.



La réponse est également sous forme de SOAP. On a donc la liste des clients dans la base de données.