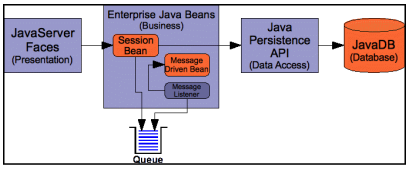
TP1 2015/2016 EJB/JPA/JSF2

Introduction

Ce TP permet une prise de contact directe et violente avec les technologies EJB, JPA, et JSF dans l'IDE NetBeans. Les cours et TPs suivants expliqueront le détail de ce que vous allez mettre en oeuvre aujourd'hui. Le thème est repris de l'article paru sur le site developer Zone : http://netbeans.dzone.com/articles/d...-ee-6-app-jsf2, mais le code demandé dans ce TP est différent du code donné dans l'article.

Voici l'architecture de l'application que vous allez développer :



Logiciels utilisés

Nous allons travailler avec la dernière version de NetBeans, **la version 8.02 Java EE** qui vient avec la dernière version du serveur JavaEE GlassFish. Veillez à avoir un la dernière version du JDK installée sur votre machine (JDK 8u60). Nous utiliserons également à la fin de ce TP quelques composants JSF2 provenant du site http://www.primefaces.org/

Ne démarrez pas le TP si vous n'avez pas ces deux versions exactes:

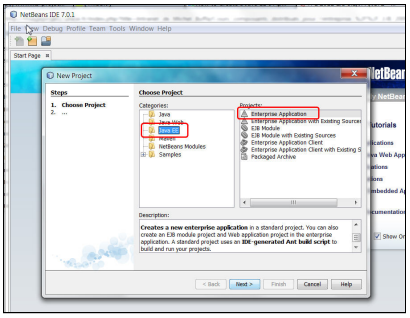
- NetBeans 8.0.2, version Java EE - INSTALLEZ TOUTES LES UDPDATES !
- JDK 8u60 ou supérieur

Remarque : les captures d'écrans de ce TP peuvent ne pas correspondre exactement à ce que vous verrez car les options proposées peuvent varier selon les versions de NetBeans. Ignorez les options supplémentaires qui sont éventuellement affichées.

Création d'un projet de type Entreprise Application

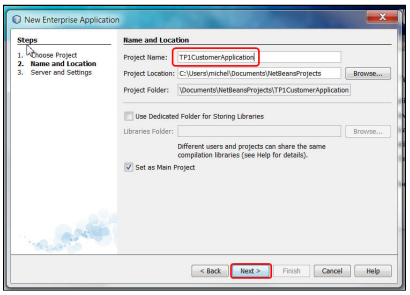
L'année dernière nous avions travaillé avec des projets de type "Web" qui incluaient tout dans un seul fichier .war, cette année nous allons aussi voir que l'on peut développer des applications de type "entreprise" qui font une séparation très nette entre les parties "Web" (servlets, html, css, JS etc.) et les parties métier/accès aux données d'une application.

Pour créer un projet "entreprise" utilisez le menu File/New Project puis choisissez la catégorie Java EE, et un projet de type "Enterprise Application".



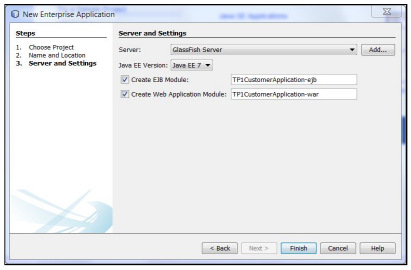
Cliquez ensuite sur le bouton "Next" puis choisissez l'emplacement du projet et son nom. **Attention, pas de caractères bizarres dans le chemin ni le nom s'il vous plait, pas de "_" par exemple, pas d'accents, etc. Ceci pour éviter les mauvaises surprises qui peuvent arriver par la suite (hibernate n'aime pas les noms de chemins bizarres, toplink n'aime pas les "_" dans les noms de projets, etc...)**

J'ai choisi comme nom de projet TP1CustomerApplication (comme vous voyez : pas d'espace ou de caractère bizarre dans le nom) et j'ai conservé l'endroit où NetBeans place les projets par défaut (là aussi, je préfère un nom de chemin sans espaces...)



Dans la fenêtre suivante, on indique le nom du serveur et la version de java EE que l'on souhaite pour le projet, la dernière en date est Java EE 7 (attention, ce n'est pas la même chose que Java SE qui est lui en version 8).

Prenez les valeurs par défaut, si elles sont différentes de celles ci-dessous, assurez-vous que cela correspond bien à la dernière version de GlassFish et à la version 7 de Java EE (qui est celle étudiée en cours).



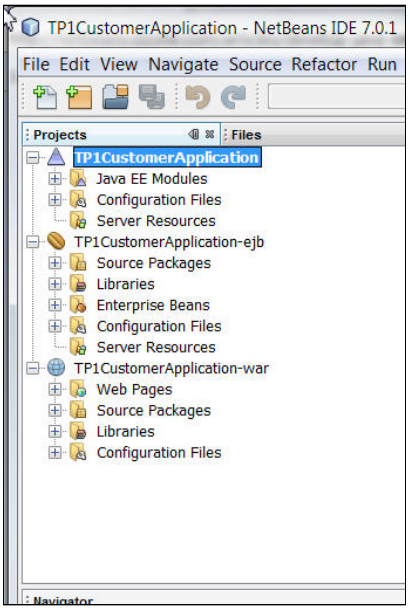
Cliquez ensuite sur le bouton "Finish", normalement NetBeans génère un projet pour votre application.

En réalité, une application "entreprise" se compose de trois projets :

1. un projet précédé par un triangle bleu qui est en fait celui qui réunit les deux autres projets au sein d'un même fichier .ear (enterprise archive), équivalent du .war mais pour les applications entreprises,

2. Un projet précédé par un grain de café, c'est le projet "métier" qui va contenir la partie métier de l'application ainsi que la partie accès aux données, il est précédé d'un grain de café (un "bean") car il contiendra certainement des "beans", des "composants java" de type "managed beans" "enterprise java beans (EJBs)" ou "entities".

3. Un projet précédé par une icône en forme de globe terrestre, c'est le projet "Web" qui va contenir le front-end web de l'application : les jsps, servlets, pages jsf ou html, les web services, etc.



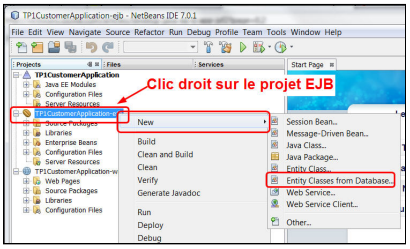
Créations de classes entités à partir d'une base de données existante

Dans ce projet nous allons utiliser une base de données. Comme nous l'avons vu en cours, on ne manipulera pas des données directement en SQL mais sous forme d'objets particuliers qu'on appelle des "classes entités".

Généralement une classe entité correspond à une table, porte le même nom (aux majuscules/minuscules près), et ses attributs correspondent aux colonnes de la table. Les instances de cette classe entité seront des objets correspondant à des lignes dans cette table. Vous en apprendrez plus dans le cours JPA.

Dans un premier temps nous allons utiliser un wizard de netbeans pour générer une classe entité à partir d'une table.

On va ajouter la classe entité dans le projet "EJBs", celui précédé d'un grain de café. Faire clic droit puis New/Entity Class from Database :



Une fenêtre apparait dans laquelle vous allez indiquer la source de données (le "nom" de la base de données sur laquelle vous allez travailler. Par défaut le jdk vient avec un SGBD qui s'appelle JavaDB (ou "Derby") qui contient une base "sample" dont le nom est "jdbc/sample". Elle contient quelques tables correspondant à un magasin de produits informatiques.

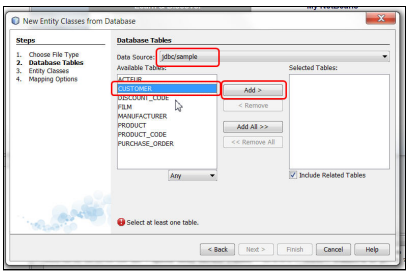
---- A NE LIRE QU'EN CAS DE PROBLEME ---

SI LA FENETRE N'APPARAIT PAS ET QUE VOUS AVEZ UNE ERREUR DE CONNEXION A LA BASE DE DONNEES, OU UNE ERREUR POUR OUVRIR LA BASE SAMPLE:

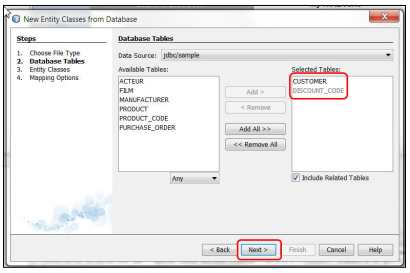
- Dans l'onglet "services", faites clic droit sur la base de données Derby puis "properties"/"propriétés":

- Assurez-vous que vous pointez bien sur l'exécutable de derby qui est dans le répertoire d'installation de GlassFish 4.1, une erreur possible est que c'est le derby du jdk que vous utilisez.
- Si malgré tout vous avez une erreur lors de l'ouverture de la base sample, allez dans votre home dir, dans le sous-répertoire netbeans-derby, c'est là que se trouvent les BDs de Derby (c:/users/xxx/documents and settings/.netbeans-derby) et supprimez le répertoire sample. Récupérez cette archive : sample.zip et dezippez-là dans ce même répertoire, elle contient une base de données sample qui marche bien.

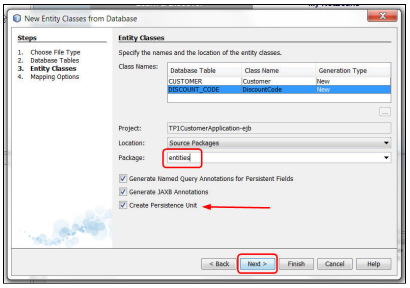
Choisissez donc jdbc/sample dans la liste des sources de données :



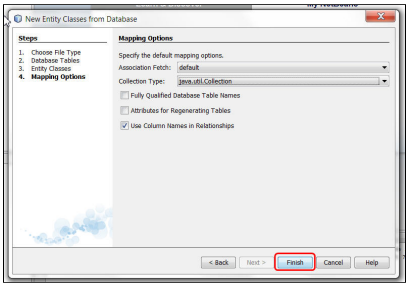
Normalement dès que vous avez choisi le nom de la base, les tables s'affichent dans la colonne de gauche. Choisissez "CUSTOMER" et cliquez sur le bouton "add", cela va ajouter en fait trois tables : la table CUSTOMER mais aussi la table DISCOUNT_CODE car il existe une jointure entre ces deux tables, et aussi la table MICRO_MARKET, qui elle aussi a une jointure avec la table CUSTOMER. Le screenshot ci-dessous date d'il y a deux ans et ne montre pas la table MICRO_MARKET, ce n'est pas grave...



Cliquez sur le bouton "Next". Maintenant on va vous proposer de changer le nom des classes entités correspondants à chacune des tables. Je conseille de laisser tout par défaut, cependant je conseille aussi de faire générer ces classes dans un package "entities" pour mieux structurer le projet :



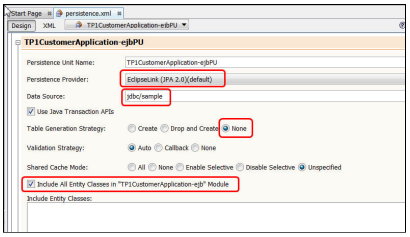
Cliquez sur le bouton "Next". L'étape suivante propose de modifier les valeurs par défaut pour les types de données correspondant à la relation 1-N entre les deux tables :



Laissez tout par défaut et cliquez sur le bouton Finish. NetBeans va un peu travailler puis vous verrez trois nouvelles choses dans le projet :

1. Les classes correspondant aux trois tables, dans le package "entities", regardez donc à quoi elles ressemblent...
2. Un fichier persistence.xml sous "Configuration Files", qui correspond à la définition d'une "persistence unit" (unité de persistence) par défaut. En réalité la "persistence unit" est l'objet qui va s'occuper de manipuler les données relationnelles à travers les classes et instances des entités. C'est la "persistence unit" qui connait la base de données, qui génère le SQL, qui va synchroniser les objets en mémoire et les données dans les tables, etc. (vous l'étudierez dans le cours sur JPA - Java Persistence API).

Double-cliquez sur le fichier persistence.xml, vous devriez voir un formulaire de configuration (une "vue" sur le fichier xml, en réalité) :



Sans entrer dans les détails, nous voyons ici que :

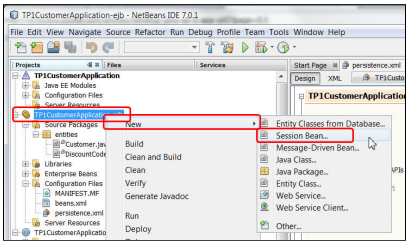
1. On n'autorise pas la création de nouvelles tables ni la suppression (None),
2. On va utiliser les transactions JTA (Java Transaction API) lors de l'accès aux données,
3. Cette persistence Unit va gérer les accès BD pour toutes les classes entités du projet.

Vous pouvez voir le code XML de ce fichier en cliquant sur "Source" et revoir la vue en cliquant sur "Design".

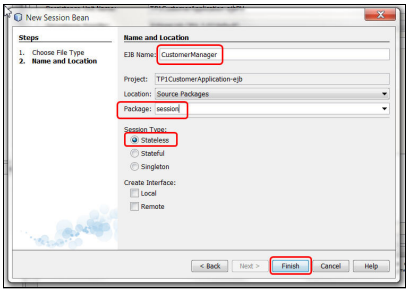
Création d'un Stateless Session Bean CustomerManager pour la gestion des clients

On va centraliser la gestion des Customers (clients, en français) dans un Stateless Session Bean. De manière classique on crée une "façade" pour les opérations élémentaires sur les clients : création, suppression, recherche, modification.

Faire clic droit New/Session Bean sur le projet EJB (celui précédé par un grain de café) :

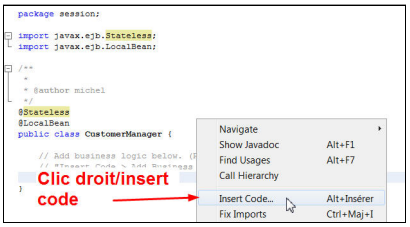


Donnez un nom à votre gestionnaire de client, et indiquez que vous le voulez dans le package "session" qui contiendra les session beans. On ne va pas préciser d'interface (locale ou remote) pour le moment (on verra plus tard à quoi cela peut servir) :

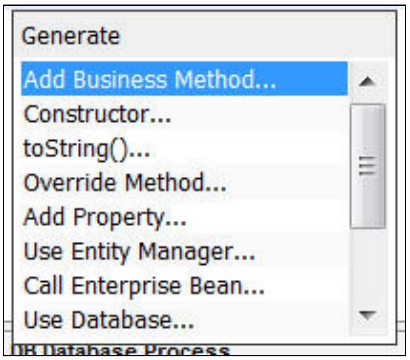


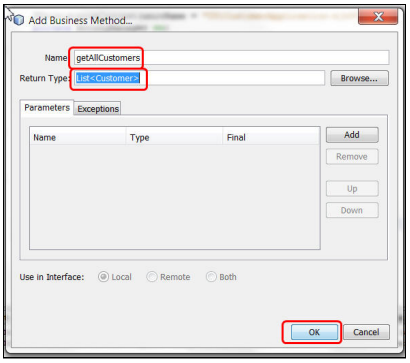
Ajout de méthodes métier dans le session bean CustomerManager

Double cliquez sur "CustomerManager.java" dans le projet pour voir le source dans l'éditeur. Cliquez ensuite dans le source de la classe, entre les { et } de la classe et faites clic droit/insert code (ou Alt+Insert). Une fenêtre propose alors diverses options, choisissez "Add Business Method" :



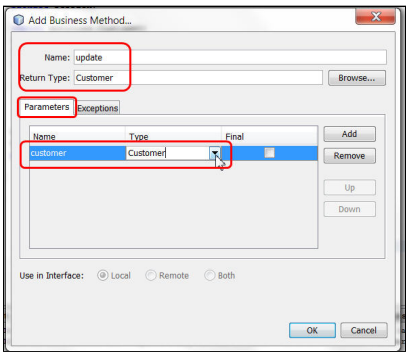
Un menu apparait proposant diverses options de génération automatique de code, choisir "business méthod" puis créez une méthode getAllCustomers qui renvoie une collection de Customer. Cette méthode servira à afficher la liste des clients :





Dans le source, ajoutez les imports qui vont bien pour ne plus avoir d'erreurs (un par un: en cliquant sur la petite lampe jaune, ou tous d'un coup: en faisant clic droit/fix imports, ou Ctrl-Shift-I, ou ⌘-Shift-I pour les Mac). **Attention** à ne pas aller trop vite ! Si la fenêtre "Fix All Imports" s'affiche c'est qu'il peut y avoir plusieurs possibilités pour certains imports et le choix proposé peut ne pas être le bon. Par exemple, il faut choisir java.util.List pour List.

Ajoutez de la même manière une méthode "update" qui prend en paramètre un Customer. Cette méthode servira à mettre à jour un client.



Vous devriez avoir un code source comme celui-ci :

```
package session;

import entities.Customer;
import java.util.Collection;
import javax.ejb.Stateless;
import javax.ejb.LocalBean;
```

```
@Stateless
@LocalBean
public class CustomerManager {

    public List<Customer> getAllCustomers() {
        return null;
    }

    public Customer update(Customer customer) {
        return null;
    }

}
```

Bien sûr vous auriez pu aussi bien taper les méthodes à la main... C'est juste pour vous montrer quelques trucs avec NetBeans...

Maitenant nous allons modifier le contenu de ces méthodes. Les explications sont dans le cours. On va commencer par ajouter un "PersistenceContext" à l'application, c'est une variable qui correspond à la persistence unit et qui va nous servir à envoyer des requêtes, insérer/supprimer des objets ou faire des mises à jour.

Faites clic droit dans le source/insert code/Use Entity Manager. Cela va ajouter automatiquement dans le code une variable avec une annotation, et une méthode :

```
@PersistenceContext(unitName = "TP1CustomerApplication-ejbPU")
 private EntityManager em;
 public void persist(Object object) {
    em.persist(object);
 }
```

La variable em n'a pas besoin d'être initialisée, en réalité son code d'initialisation va être "injecté" par l'annotation. Regardez le nom "TP1CustomerManager-ejbPU", on le retrouve dans le fichier persistence.xml, en gros, ce que l'on doit comprendre par ces deux lignes :

1. La variable em correspond à une persistence unit (celle dont le nom est précisé, qui gère la base jdbc/sample),

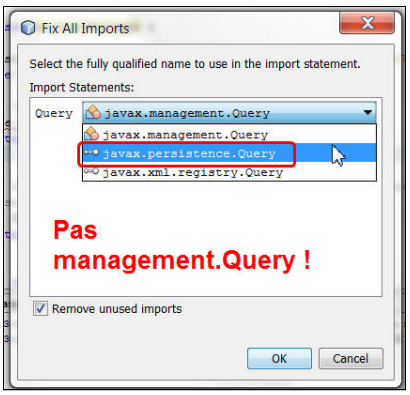
2. Je n'ai pas besoin de l'initialiser.

On va maintenant modifier le code des deux méthodes pour qu'elles jouent leur rôle :

```
package session;

import entities.Customer;
import java.util.Collection;
import javax.ejb.Stateless;
import javax.ejb.LocalBean;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
@LocalBean
public class CustomerManager {

    @PersistenceContext(unitName = "TP1CustomerApplication-ejbPU")
    private EntityManager em;

    public List<Customer> getAllCustomers() {
        Query query = em.createNamedQuery("Customer.findAll");
        return query.getResultList();
    }

    public Customer update(Customer customer) {
        return em.merge(customer);
    }

    public void persist(Object object) {
        em.persist(object);
    }
}
```

Voilà, on a rempli les deux méthodes avec pour la première un code qui va exécuter une requête JPQL dont le nom est "findAll" et qui est définie dans la classe Customer.java (allez voir, la requête correspond à un select * sur les clients), la seconde, la méthode update indique que l'on souhaite modifier un client dans la table avec les valeurs de celui qui est passé en paramètre. Cela suppose que l'attribut id du client passé en paramètre correspond à une clé primaire dans la table des clients.

Normalement vous devez ajouter un import pour enlever l'erreur de syntaxe sur le type Query. Attention, un piège classique est d'ajouter le mauvais import. Faites clic droit/fix import :



Voilà, on a terminé pour la partie "métier"/EJB dans ce projet. On va passer au front-end web.

Création de la partie front-end web

Dans cette partie on va utiliser le framework MVC de java EE qui se base sur l'utilisation de backing beans et de pages JSF.

Une **page JSF** est en gros une page HTML avec des balises HTML ou des balises qui représentent des composants JSF similaires aux composants HTML (par exemple une liste déroulante ou une table) ou plus complexes (par exemple un calendrier ou un arbre). Cette page contient des emplacements délimités par #{ ... } qui contiennent des données dynamiques (leur valeur change avec le contexte). Ces données dynamiques proviennent le plus souvent de **propriétés** de classes Java appelées des backing beans (un bean qui "épaule" la page JSF) ou beans gérés (car ces beans sont gérés par le container Web du serveur d'application). Les backing beans peuvent aussi contenir du code qui sera exécuté à la suite d'une action de l'utilisateur (validation d'un formulaire ou clic sur un lien hypertexte de la page JSF par exemple). Une propriété d'un Java Bean est définie par un getter (getNom() par exemple) et/ou par un setter (setNom(String nom) par exemple).

Il y aura 2 pages JSF :

- CustomerList.xhtml, liste de tous les clients ;
- CustomerDetails.xhtml, page qui affiche les détails sur un client particulier et qui permet de modifier les informations sur ce client.

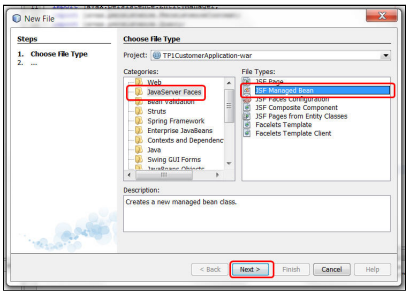
Un lien placé dans chaque ligne de la liste des clients permettra de lancer une requête HTTP GET pour faire afficher les détails sur le client concerné.

Création d'un Bean Managé en tant que "contrôleur web"

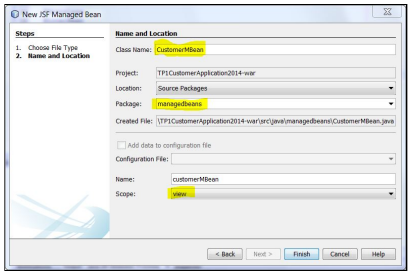
OBLIGATOIRE CE QUI EST CI-DESSOUS, NE SCROLLEZ PAS SANS LE LIRE !

Création du fichier beans.xml : les backing beans (ou managed beans) que vous allez créer sont gérés par CDI. Pour que NetBeans les crée correctement il faut commencer par créer un fichier beans.xml qui indique que vous allez utiliser CDI. **Pour cela, New sur le projet web puis "Other...", puis "Contexts and Dependency Injection", et enfin "beans.xml (CDI Configuration File)"**. Ceci devrait ajouter un fichier beans.xml dans "Web Pages" > WEB-INF" ; vérifiez. vous pouvez maintenant créer les backing beans.

Faites sur le **projet web** (celui avec le globe terrestre) clic droit/new/other et choisissez dans la catégorie "java server faces", "jsf managed bean" :



Ensuite, renseignez le nom de la classe que vous souhaitez ainsi que le package où va se trouver la classe générée (Attention, pas de majuscule dans le nom de package à cause d'un bug de NetBeans). Choisissez la portée (Scope) View (avec les versions de NetBeans antérieures à 8.0.1 il n'est pas possible de choisir cette portée ; vous choisissez la portée session et vous la changer par la portée view ensuite directement dans le source) :



Si tout va bien vous devriez obtenir une classe CustomerMBean.java dans le package "managedbeans" , et le source devrait être celui-ci (vérifiez que vous avez bien "import javax.faces.view.ViewScoped") :

```
package managedbeans;

import javax.inject.Named;
import javax.faces.view.ViewScoped;

@Named(value = "customerMBean")
@ViewScoped
public class CustomerMBean {

    public CustomerMBean() {
    }
}
```

Nous allons ajouter du code pour que le bean puisse communiquer avec le session bean stateless de l'autre module (celui avec les EJBs) : CustomerManager.

Pour que le bean soit client d'un EJB, faites dans le code clic droit/insert code/call enterprise bean, et sélectionnez l'EJB CustomerManager, ceci devrait insérer dans le source les lignes suivantes :

```
@EJB
  private CustomerManager customerManager;
```

Là aussi, l'annotation @EJB injectera le code d'initialisation, vous n'avez pas à faire de "new" (vous ne DEVEZ PAS faire de "new" !) sur des session beans. Voir le cours pour plus de détails sur ce mécanisme.

Vous allez compléter la classe du bean avec le code qui suit. Quelques détails sur ce code :

- la classe doit implémenter Serializable car un bean de portée View peut être temporairement retiré de la mémoire centrale si la mémoire est saturée ;
- méthode getCustomers() qui retourne la liste de tous les clients et qui va permettre d'afficher tous les clients dans une table ;
- méthode showDetails qui va être utilisée pour naviguer vers la page CustomerDetails.xhtml qui affiche les détails sur un client.

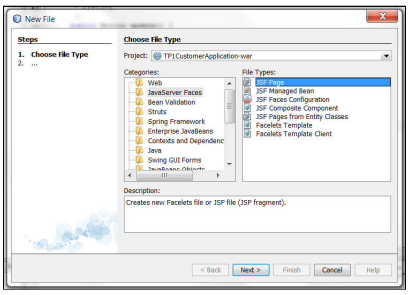
La méthode getCustomers() utilise une variable d'instance customerList qui sera utilisée plus loin dans ce TP par PrimeFaces pour trier la liste des clients.

```
package managedbeans;

import entities.Customer;
import javax.ejb.EJB;
import javax.inject.Named;
import javax.faces.view.ViewScoped;
import java.io.Serializable;
import java.util.List;
import session.CustomerManager;

@Named(value = "customerMBean")
@ViewScoped
public class CustomerMBean implements Serializable {
    private List<Customer> customerList;

    @EJB
    private CustomerManager customerManager;

    public CustomerMBean() {
    }

    /**
     * Renvoie la liste des clients pour affichage dans une DataTable
     * @return
     */
    public List<Customer>getCustomers() {
        return customerManager.getAllCustomers();
    }

    /**
```

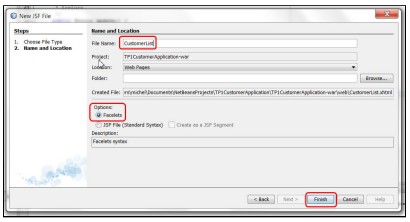
```
      * Action handler - appelé lorsque l'utilisateur sélectionne une ligne dans
      * la DataTable pour voir les détails
      */
     public String showDetails(int customerId) {
         return "CustomerDetails?idCustomer=" + customerId;     }
 }
```

Ajout d'une page JSF pour afficher la liste des clients

Pour ajouter une page JSF, faire sur le projet web clic droit/new/other/Java Server Faces et choisir "JSF Page" :



Puis faites "next" et renseignez le nom de la page :



Notez que cela ajoute un fichier CustomerList.xhtml dans le projet, à côté de la page index.jsp. Double cliquez pour voir le source. Modifiez le Title pour mettre "Liste des clients", vous pouvez rajouter un titre HTML <h1>liste des clients </h1> dans le <h:body>...</h:body> etc.

Ajout d'une DataTable JSF dans la page

Remarque : les captures d'écrans de ce TP ont été faites avec une version de JSF antérieures à la version 2.2. Les nouveaux espaces de noms introduits par JSF 2.2 sont http://xmlns.jcp.org à la place de

http://java.sun.com ; par exemple xmlns:f="http://xmlns.jcp.org/jsf/core". NetBeans met correctement les nouveaux espaces de noms.

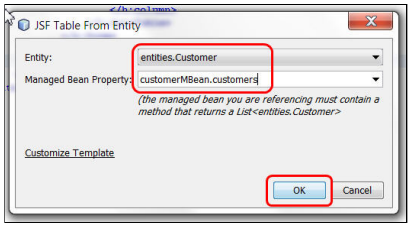
Faites apparaitre la Palette dans NetBeans (menu Window/Palette ou raccourci ctrl-shift-8). Ouvrez la partie JSF puis faite un drag'n'drop de "JSF Data Table from Entity", en dessous du body. Le <h1>... présent dans la photo d'écran est inutile, car le code que va générer netbeans en contiendra un...

Une fenêtre de dialogue va apparaitre demandant de préciser pour quelle classe entité vous voulez une Data Table, indiquez la classe entité Customer, puis comme property indiquez le nom du managed bean pour les clients suivi du nom de la propriété correspondant à une liste de clients. Dans notre cas, il faut mettre customerMbean.customers.

Pourquoi ? Car lorsque dans une JSP ou dans une page JSF on référencera une "propriété" de lecture, cela appellera la méthode get correspondante. Par exemple, si on référence la propriété "customers" du managed bean customerMBean, cela ne fonctionnera que si il existe une méthode getCustomers() qui renvoie une Collection de Customer (en effet, la Data Table gère des Collection). Il n'est pas nécessaire qu'une variable "customers" existe dans le bean, il suffit que la méthode Collection<Customer> getCustomers() existe. La preuve :

```
@Named(value = "customerMBean")
@ViewScoped
public class CustomerMBean implements Serializable {
...
public Collection getCustomers() {
    return customerManager.getAllCustomers();
}
```

Voici donc la fenêtre avec les bonnes valeurs :



Ceci devrait insérer de nombreuses lignes dans la page JSF.

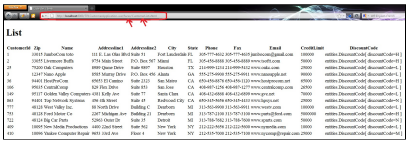
Exécution du projet et premier test

Pour exécuter le projet, faites clic droit sur le projet en forme de triangle (le projet "enterprise") et Run. Une page va s'afficher, modifiez l'URL pour afficher la bonne page :

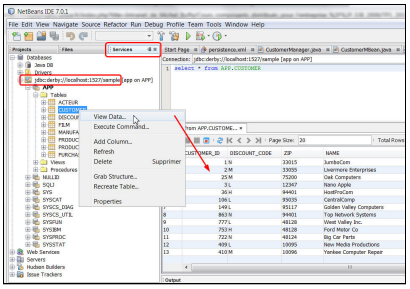
1. Ajoutez "faces" dans l'URL,
2. Indiquez le nom de la page JSF avec son suffixe .xhtml

Si vous voulez éviter d'avoir à modifier l'URL à chaque lancement de l'application, vous pouvez aussi modifier le fichier web.xml : remplacer le welcome-file par celui-ci : faces/CustomerList.xhtml.

Vous devriez obtenir le résultat suivant :



Pour le moment ce tableau n'est pas très bien présenté car il n'y a aucune fioriture de mise en page. Les données proviennent de la base jdbc/sample. Vous pouvez vérifier que ces données sont les bonnes, allez dans l'onglet "Services" de NetBeans qui comprend un gestionnaire de base de données assez simple, mais très pratique. Ouvrez la vue sur les tables de la base jdbc/sample et faites clic droit/view data :



Vous pouvez vérifier que ce sont bien les mêmes données qui ont été affichées dans la page JSF.

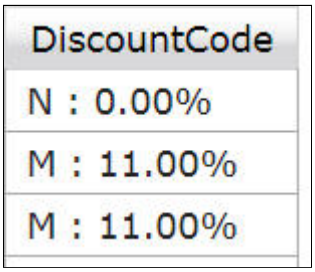
Maintenant on va modifier l'affichage du discountCode qui est une relation, en effet, voir "entities.DiscountCode[discountCode=M]" n'est pas très satisfaisant. Si vous avez compris le concept de "propriétés", vous pouvez modifier la ligne qui affiche le code. remplacez :

```
<h:outputText value="#{item.discountCode}"/>
```

par :

```
<h:outputText value="#{item.discountCode.discountCode} : #{item.discountCode.ra
```

Ce qui aura pour effet de remplacer la première expression par le résultat de l'appel de la méthode getDiscountCode() de la classe DiscountCode.java, et la seconde par la valeur renvoyée par getRate() de cette même classe. L'affichage sera bien meilleur :



Faites la même chose avec le zip code! Cherchez dans MicroMarket.java la propriété qui correspond au code postal (zip code) sous forme de S

Etudiez maintenant le code de la page JSF, du managed bean, du session bean de l'entity bean, et essayez de retrouver les morceaux que vous avez développés dans le schéma présenté dans la toute première illustration de cette page. Regardez également comment s'articulent les différentes parties et dans quel ordre elles sont exécutées.

Remplacement de la DataTable par une provenant de la librairie de composants JSF PrimeFaces

Ajout de la librairie PrimeFaces dans le projet

PrimeFaces propose des composants évolués/complémentaires pour JSF, le site web de référence est : http://www.primefaces.org

Cette librairie est déjà présente dans NetBeans 8.0.1, dans sa version 5.0 (prendre la version fournie avec votre NetBeans, qui peut être plus récente). Nous allons utiliser la version 5.0.

Si vous ne savez pas comment ajouter une librairie dans NetBeans : clic droit sur l'entrée Libraries du projet "war" et choisir "Add Library...". Choisissez la librairie PrimeFaces. Si vous voulez utiliser une version plus récente de PrimeFaces, que vous avez récupérée, cliquer sur le bouton "Create..." et nommer la librairie ("Primefaces 5.1" par exemple). Cliquer sur OK et cliquer sur "Add JAR/Folder..." pour ajouter le jar que vous avez récupéré.

Modification de la page JSF

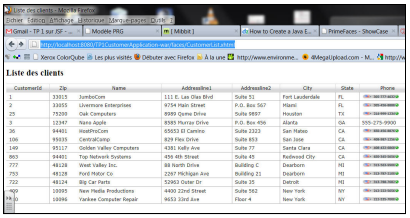
Pour pouvoir utiliser des tags provenant de PrimeFaces dans une page JSF il faut ajouter le namespace suivant : (cf http://primefaces.org/gettingStarted.html)

```
xmlns:p="http://primefaces.org/ui"
```

A partir de là on pourra utiliser des tags PrimeFaces avec le préfixe p:

- Remplacez simplement tous les tags <h:dataTable> et </h:dataTable> par <p:dataTable> et </p:dataTable>
- Idem pour les tags <h:column> et </h:column>, remplacez les h: par p:

Déployez le projet (clic droit sur le projet avec le triangle/deploy) puis exécutez à nouveau la page qui affiche la liste des clients. Il est nécessaire de déployer car il faut que la librairie que nous venons d'ajouter au projet soit déployée dans le serveur. Il suffit de le faire une fois pour le projet, ensuite le déploiement incrémental (avec save ou ctrl-s) suffira. Vous devriez obtenir le résultat suivant :



C'est déjà mieux présenté non ? Bon, ce qui est intéressant, c'est que le composant dataTable de PrimeFaces possède de nombreuses options pour la pagination, rendre les colonnes triables, éditables, etc. Allez donc voir les démos/sources sur : http://www.primefaces.org/showcase/u...le/basic.xhtml

1. Par exemple, ajoutez dans le tag <p:dataTable ...> les attributs paginator="true" et rows="10", sauvez, rechargez la page, ça y est, les résultats sont paginés (en mémoire, nous verrons plus tard comment faire des requêtes paginées sur la BD),
2. Ensuite, faites en sorte que certaines colonnes soient triables: attribut sortBy="#{item.propriétéDeLaColonne}" dans le tag p:column des colonnes triables. Rappel : ça ne marche que si on a bien une variable d'instance pour conserver la liste des clients dans le backing bean et il vous faudra donc modifier le code de la méthode getCustomers(),
3. Enfin, pour installer quelques filtres et champs de recherche sur la table regardez comment faire pour installer un champ de recherche sur certaines colonnes (regarder la démo "filter" sur le site de PrimeFaces).

Affichage des détails d'un client lorsqu'on clique sur une ligne

Maintenant nous allons voir comment afficher dans une autre page les détails d'un client lorsqu'on clique sur une ligne.

Ajout d'un lien dans le tableau pour déclencher l'affichage des détails d'un client

Modifiez la page CustomerList.xhtml de manière à ce que lorsqu'on clique sur la colonne Id on affiche le détail d'un client, comme à la ligne 18 du listing ci-dessous.

```
<p:dataTable value="#{customerMBean.customers}" var="item"
             emptyMessage="No customer found with given criteria"
```

```
            widgetVar="customerTable"
            paginator="true"
            rows="10">

 <f:facet name="header">
    <p:outputPanel>
        <h:outputText value="Search all fields:" />
        <p:inputText id="globalFilter" onkeyup="customerTable.filter()" style="
    </p:outputPanel>
</f:facet>

<p:column headerText="CustomerId"
          sortBy="#{item.customerId}"
          filterBy="#{item.customerId}"
          filterMatchMode="contains">
    <h:link outcome="#{customerMBean.showDetails(item.customerId)}" value="#{i
</p:column>
...
```

la ligne ajoutée ajoutera un lien hypertexte dans la colonne ; le texte du lien sera l'id du client (attribut "value=...") et lorsque l'utilisateur cliquera sur la colonne, la méthode showDetails du managedBean sera appelée avec l'id du client en paramètre. La page affichée par le lien est indiquée par la valeur de retour de la méthode (dont le listing suit) : CustomerDetails?customerId=2 (si l'id du client est 2). JSF ajoutera automatiquement le suffixe ".xhtml" au fichier et ajoutera au début l'adresse de l'application (http://localhost:8080/...). Le source de la méthode showDetails de CustomerMBean.java :

```
/**
     * Action handler - appelé lorsque l'utilisateur sélectionne une ligne dans
     * la DataTable pour voir les détails
     * @param customerId
     * @return
     */
    public String showDetails(int customerId) {
        return "CustomerDetails?idCustomer=" + customerId;
    }
```

Si vous sauvegardez la page JSF et l'exécutez, **vous verrez que la colonne id affiche des erreurs, car la page CustomerDetails.xhtml n'existe pas encore**. Vous pouvez en créer une vide ou bien suivre le

TP qui va montrer comment la créer, mais avant, on va commencer par ajouter son backing bean. Une bonne pratique est de toujours créer le backing bean avant la page JSF qui lui sert de vue.

Ajout d'un backing bean pour la page des détails

1. Vous allez créer un autre JSF Managed Bean intitulé CustomerDetailsMBean,
2. Vous lui donnerez comme portée "View"
3. Il doit être Serializable
4. Vous insèrerez dedans les méthodes et propriétés du source suivant:

```
package managedbeans;

import entities.Customer;
import java.io.Serializable;
import java.util.List;
import javax.ejb.EJB;
import javax.faces.view.ViewScoped;
import javax.inject.Named;
import session.CustomerManager;

/**
 * Backing bean pour la page CustomerDetails.xhtml.
 */
@Named
@ViewScoped
public class CustomerDetailsMBean implements Serializable {
  private int idCustomer;
  private Customer customer;

  @EJB
  private CustomerManager customerManager;

  public int getIdCustomer() {
    return idCustomer;
  }

  public void setIdCustomer(int idCustomer) {
    this.idCustomer = idCustomer;
  }

  /**
   * Renvoie les détails du client courant (celui dans l'attribut customer de
   * cette classe), qu'on appelle une propriété (property)
   */
```

```
  public Customer getDetails() {
    return customer;
  }

  /**
   * Action handler - met à jour la base de données en fonction du client passe
   * en paramètres, et renvoie vers la page qui affiche la liste des clients.
   */
  public String update() {
    System.out.println("###UPDATE###");
    customer = customerManager.update(customer);
    return "CustomerList";
  }

  /**
   * Action handler - renvoie vers la page qui affiche la liste des clients
   */
  public String list() {
    System.out.println("###LIST###");
    return "CustomerList";
  }

  public void loadCustomer() {
    this.customer = customerManager.getCustomer(idCustomer);
  }
}
```

La classe CustomerDetailsMBean.java de ce bean contient;

- une propriété "readonly" details (on n'a que la méthode getDetails(), pas de setDetails(), c'est àa une propriété readonly, avec juste un get...) qui renvoie le client dont les détails sont affichés par la page CustomerDetails ;
- une propriété idCustomer pour récupérer le paramètre idCustomer de l'URL (CustomerDetails.xhtml?idCustomer=2) ;
- une méthode loadCustomer pour récupérer un client à partir de idCustomer (en utilisant l'EJB CustomerManager), ainsi qu'une propriété customer pour conserver ce client ;
- une méthode pour enregistrer les modifications apportées au client par l'utilisateur de l'application.

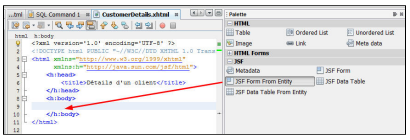
La méthode loadCustomer appelle une méthode dans l'EJB customerManager qu'il va falloir rajouter. Pour rechercher un Customer par son Id voici comment faire:

Dans CustomerManager.java:

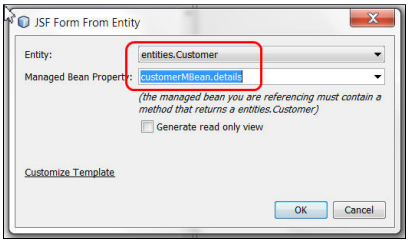
```
public Customer getCustomer(int idCustomer) {
        return em.find(Customer.class, idCustomer);
}
```

Ajout d'une page JSF pour afficher les détails d'un client

1. Faites sur le projet web clic droit/new/jsf page et créez une page CustomerDetails.xhtml,
2. Drag and droppez dans le body de la page "jsf form from entity".



Indiquez ensuite le nom de la classe entité dont vous voulez afficher les informations dans un formulaire, ainsi que le nom de la méthode du managedBean qui renvoie les détails d'un client :



Ici on a mis customerMBean.details qui signifie que c'est la méthode getDetails() qui sera appelée.

Une fois qu'on a validé, la page JSF se remplit avec des lignes qui sont en fait un formulaire qui sera pré-rempli dès l'affichage s'il existe un objet "customer" dans la session (ce qui sera le cas, voir ce que nous avons expliqué dans la section précédente).

Pour que cette page affiche bien les détails sur le client dont l'id est passé en paramètre de la requête HTTP GET, il reste à récupérer l'id dans le backing bean et à lancer la méthode loadCustomer. Pour cela, il faut modifier le code de la page CustomerDetails en ajoutant ceci juste avant la balise <h:head> et après la balise <html xmlns=...> :

```
<f:metadata>
  <f:viewParam name="idCustomer" value="#{customerDetailsMBean.idCustomer}"
               required="true"/>
  <f:viewAction action="#{customerDetailsMBean.loadCustomer}"/>
</f:metadata>
```

La section <f:metadata> peut servir à plusieurs choses. Ici elle sert à indiquer ce qui sera exécuté juste avant que la page ne soit affichée.

<f:viewParam> indique à JSF de récupérer la valeur du paramètre de nom "idCustomer" et de la mettre dans la propriété idCustomer du backing bean customerDetailsMBean (avec le setter setIdCustomer).

<f:viewAction> indique à JSF de lancer la méthode loadCustomer du backing bean customerDetailsMBean.

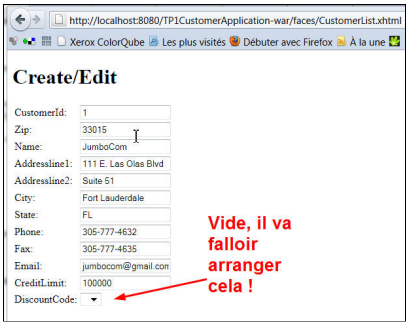
Ces 2 actions sont effectuées dans cet ordre lorsque la requête GET arrive sur le serveur. Tout se passe donc comme on veut ; par exemple, si l'URL contient CustomerDetails.xhtml?idCustomer=2, la valeur 2 est mise dans la propriété idCustomer du backing bean et ensuite, le client d'id 2 est chargé depuis la base de données et mis dans la variable customer du backing bean. La page des détails affichera les informations sur ce customer ; on peut le voir en lisant le code généré par NetBeans pour la page des détails ; par exemple <h:inputText id="name" value="#{customerDetailsMBean.**details**.name}" title="Name" /> avec la propriété details qui correspond à la variable customer dans le backing bean.

Il reste à ajouter le code pour passer de la page des détails sur un client à la page de la liste des clients mais vous pouvez tester l'affichage de la page des détails à partir de la liste de tous les clients.

Avec le code que l'on vient d'ajouter on peut même modifier l'URL de l'affichage de la page des détails à partir de la liste des clients en changeant "à la main" la valeur du paramètre idCustomer, et en faisant ré-afficher la page. Par exemple en changeant idCustomer=2 en idCustomer=36 (il faut choisir un id de client qui existe dans la base de données).

Exécutez le projet et testez que les détails s'affichent bien

Maintenant vous pouvez exécuter le projet et voir si lorsque vous cliquez sur le lien Id dans la liste ça affiche bien le détail d'un client. Vous devriez obtenir ceci pour le client d'id=1 :



Voilà, les détails du client numéro 1 s'affichent bien, cependant le code de discount pour ce client n'est pas correctement affiché.

Nous allons voir maintenant comment arranger ce petit problème, comment ajouter des boutons de navigation et comment tant qu'à faire permettre une modification des données.

Régler l'affichage des DiscountCodes dans le détail d'un client

Dans la base de données la table des DISCOUNT_CODE contient juste 4 valeurs correspondants aux quatre types de réductions possibles. Nous allons utiliser un Converter, qui permet de transformer un objet de type entities.DiscountCode en String, et vice versa.

On va commencer par rajouter au projet EJB une facade pour la table DISCOUT_CODE. Une façade est un session bean stateless. Dans ce bean on ajoutera une méthode List<DiscountCode> getAllDiscountCodes(). Inspirez-vous du gestionnaire de client et de la méthode qui renvoie tous les clients. Plutôt que copier/coller le code ci-dessous, essayez de refaire les étapes que nous avions faites lors de l'écriture du gestionnaire de clients (avec insert code/call enterprise bean, insert code/user Entity Manager etc...)

Vous devriez avoir un code comme ceci :

```
package session;

import entities.DiscountCode;
import java.util.List;
import javax.ejb.Stateless;
import javax.ejb.LocalBean;
import javax.persistence.EntityManager;
```

```
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

@Stateless
@LocalBean
public class DiscountCodeManager {

    @PersistenceContext(unitName = "TP1CustomerApplication-ejbPU")
    private EntityManager em;

    public List<DiscountCode> getDiscountCodes() {
        Query query = em.createNamedQuery("DiscountCode.findAll");
        return query.getResultList();
    }

    public void persist(Object object) {
        em.persist(object);
    }
}
```

Donc, maintenant on a un EJB qui renvoie toutes les valeurs possibles de DiscountCode.

On va ajouter dans le managed bean du projet web un objet de type javax.faces.convert.Converter, qui nous servira à transformer un objet de type entities.DiscountCode en String, et vice versa. On utilisera une représentation en string de la forme "L : 7.00%", comme dans la colonne du tableau de détail des clients. Pour cela, commencez par indiquer au bean qu'il va appeler un EJB : DiscountCodeManager, celui que nous venons d'écrire. Ensuite, pour utiliser ce converter dans la page JSF .xhtml, on a besoin de définir une méthode get. Voici le code à ajouter dans le managed bean CustomerDetailsMBean:

```
@EJB
private DiscountCodeManager discountCodeManager;

public Converter getDiscountCodeConverter() {
    return discountCodeConverter;
}

private Converter discountCodeConverter = new Converter() {

    @Override
    public Object getAsObject(FacesContext context, UIComponent component, Str
```

```
            return new ConverterException("On verra la conversion String->Objet plu
        }
        @Override
        public String getAsString(FacesContext context, UIComponent component, Obje
            DiscountCode dc = (DiscountCode) value;
            return dc.getDiscountCode()+" : "+dc.getRate()+"%";
        }
    };
```

Finalement, on peut définir une méthode qui servira à remplir le menu déroulant de la page JSF CustomerDetails.xhtml. Procédez comme dans le début du TP (insert code/call enterprise bean), puis insérez dans le même bean CustomerDetailsMBean, la méthode suivante :

```
    /**
     * renvoie une liste de DiscountCode pour affichage dans le menu déroulant
     * de la page des détails d'un client
     * @return
     */
    public List<DiscountCode> getAllDiscountCodes() {
        return discountCodeManager.getDiscountCodes();
    }
```

On va pouvoir maintenant modifier la partie "menu" de la page CustomerDetails.xhtml afin que la méthode que nous venons d'écrire soit appelée. Voici le code qu'il faut modifier dans CustomerDetails.xhtml :

```
<h:selectOneMenu id="discountCode" value="#{customerDetailsMBean.details.discou
                 title="DiscountCode" required="true" requiredMessage="The Disco
                 converter="#{customerDetailsMBean.discountCodeConverter}">
        <f:selectItems value="#{customerDetailsMBean.allDiscountCodes}"
                       var="item"
                       itemLabel="#{item.discountCode} : #{item.rate} %" itemVa
</h:selectOneMenu>
```

Ce code ajoute un menu déroulant dans lequel l'utilisateur peut choisir une seule option. Ce choix est rangé dans la variable définie par l'attribut value de <h:selectOneMenu>. Les choix sont définis par la balise <f:selectItems> : les choix sont définis par l'attribut value de <f:selectItems>. Ce qui est affiché est

déterminé par l'attribut itemLabel et la valeur qui correspond au choix et qui est envoyée au serveur est déterminée par l'attribut itemValue (le convertisseur sera utilisé pour passer de DiscountCode à String et de String à DiscountCode à String (pour ranger le choix de l'utilisateur comme un DiscountCode dans l'entité Customer).

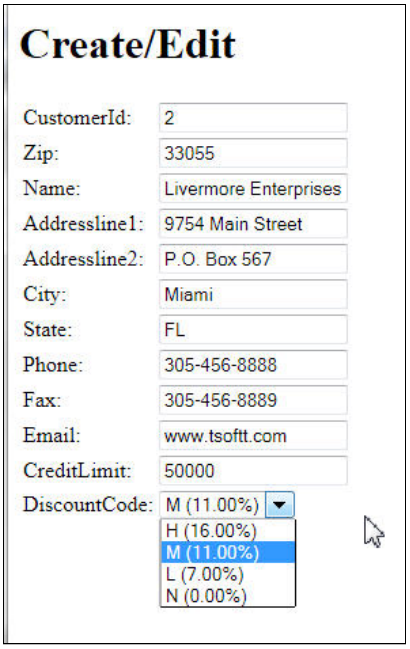
Remarquons plusieurs choses :

1. Le contenu du menu est invoqué par value="#{customerDetailsMBean.allDiscountCodes}" du tag f:selectItems, cette expression (il s'agit du langage EL de JSF) correspond à un appel à getDiscountCodes() sur le bean de nom customerDetailsMBean, c'est la méthode que l'on vient d'écrire.
2. L'attribut qui correspond à la propriété du "modèle associé" au menu, c'est à dire la propriété dont la valeur sera affichée dans le menu comme choix par défaut (appel au getter), mais c'est aussi la propriété qui sera modifiée (par un appel à son setter) lorsque la valeur sera modifiée par un nouveau choix dans le menu. Comme les menus renvoient une String lorsqu'on leur demande la valeur choisie, c'est le Converter qui se charge de transformer un Objet en string et vice-versa. Ici:
    1. le modèle associé au menu est spécifié par l'attribut value="#{customerDetailsMBean.details.discountCode}" du tag h:selectOneMenu (appel de la méthode getDiscountCode() de l'objet renvoyé par la méthode getDetails() du customerDetailsMBean
    2. l'instance de Converter à utiliser est définie par l'attribut converter="#{customerDetailsMBean.discountCodeConverter}"du tag h:selectOneMenu (appel de la méthode getDiscountCodeConverter() de du customerDetailsMBean.

On peut utiliser le Converter dans la page JSF CustomerList.xhtml également :

```
<p:column headerText="DiscountCode"
          sortBy="#{item.discountCode.discountCode}"
          filterBy="#{item.discountCode.rate}%"
          filterMatchMode="contains">
    <h:outputText value="#{item.discountCode}" converter="#{customerDetailsMBea
</p:column>
```

Voilà, sauvegardez, testez, ça doit fonctionner :



Voilà ! Maintenant il ne reste plus qu'à ajouter des boutons de navigation pour mettre à jour les données ou revenir à la liste des clients.

Ajout de boutons pour la mise à jour et retour à la liste

Il suffit de rajouter deux lignes dans le fichier CustomerDetails.xhtml ! Ajoutez ces deux lignes :

```
<h:commandButton id="back" value="Back" action="#{customerDetailsMBean.list}"/
 <h:commandButton id="update" value="Update" action="#{customerDetailsMBean.up
```

juste après le menu déroulant pour les discountCodes et avant la fin du formulaire (avant le </h:form>)... Vous comprenez maintenant ce que cela signifie :

- Si on clique sur le premier bouton, on appelle list() dans CustomerMBean.java,
- Si on clique sur le second bouton on appelle update()...

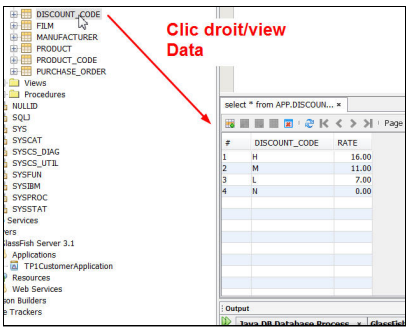
Regardons la version actuelle de la méthode update() :

```
public String update() {
        System.out.println("###UPDATE###");
```

```
        customer = customerManager.update(customer);
        return "CustomerList";
}
```

Cette méthode prend la propriété "customer", de type Customer et appelle une méthode métier dans le gestionnaire de clients qui va mettre à jour le client en base. Les attributs de cette propriété (nom, etc...) ont pour valeurs celles du formulaire de la page de détails. Si on a modifié ces valeurs avant de cliquer sur le bouton "update", les attributs de la propriété ont été modifiés. Donc on a bien dans "customer" un client mis à jour (en mémoire pour le moment, regardez ce que fait customerManager.update() pour voir. On verra en cours que c'est là que se fait la modification en base de données).

Il existe en effet une relation entre les clients et les codes de remise. Regardez attentivement la classe entité Customer.java, vous verrez que le code de remise est un attribut discountCode de type DiscountCode. Or le menu déroulant dans la page de détails, lorsqu'on modifie le code de remise, nous renvoie une String. C'est le travail du Converter de transformer cette String en objet de type DiscountCode !



Observez la requête nommée DiscountCode.findByDiscountCode définie dans un attribut de la classe bean entité DiscountCode. Cette requête possède un paramètre noté :discountCode. Quand on donne la valeur d'un code de réduction à ce paramètre, on peut récupérer l'objet de type DiscountCode correspondant. On propose d'utiliser une nouvelle méthode public DiscountCode getDiscountCodeByDiscountCode(char code) dans le bean session session.DiscountCodeManager. Voici cette méthode:

```
public DiscountCode getDiscountCodeByDiscountCode(char code) {
        Query query = em.createNamedQuery("DiscountCode.findByDiscountCode");
        query.setParameter("discountCode", "" + code);
        return (DiscountCode) query.getSingleResult();
 }
```

Observez: on associe le caractère code au paramètre de requête "discountCode", et on récupère un unique résultat de type DiscountCode: query.getSingleResult().

Voici finalement ce qu'on vous propose comme transformation String->DiscountCode par le Converter:

```
private Converter discountCodeConverter = new Converter() {

        @Override
        public Object getAsObject(FacesContext context, UIComponent component,
            char code = value.charAt(0);
            DiscountCode dc = discountCodeManager.getDiscountCodeByDiscountCod
            return dc;
       }
        @Override
        public String getAsString(FacesContext context, UIComponent component,
            DiscountCode dc = (DiscountCode) value;
            return dc.getDiscountCode()+" : "+dc.getRate()+"%";
        }
    };
```

Supprimez dans la page des détails le dernier menu à propos des zip codes). Testez, exécutez, testez, amusez-vous à modifier les valeurs des clients, vérifiez dans le gestionnaire de base de données de netbeans que les modifications ont été prises en compte...

Vous pourrez à titre d'exercice faire la même chose (convertisseur etc) pour le menu zip code.

Problèmes devant encore être réglés :

- Lorsque l'on passe de la page des détails à la liste des client, les URLs ont un temps de retard, la page de la liste affiche l'URL de la page des détails. Pourquoi ?

- Plus gênant, si on modifie les informations sur un client et qu'on recharge tout de suite après la page qui affiche la liste de tous les clients, on reçoit un message demandant de confirmer l'envoi des modifications et ensuite on a un message d'erreur. Vous comprenez pourquoi ?

Dans le prochain TP et en cours nous verrons comment corriger ces problèmes avec le modèle PRG.
