Exemple10_J2EE

Un exemple de développement J2EE

Avec Eclipse Eclipse 2019-06 et WindFly (JBoss)

Exemple de l'utilisation de :

- Un **EJB Session Stateful**
- Un **EJB Sesssios Stateless**
- Des **Entity** utilisés en persistence par les EJB
- Des **DrivenMessage**
- Des **Servlets** qui utilisent les EJB
- Des pages **JSP** qui réalisent l'interface IHM et qui utilisent aussi les EJB
- **JQuery** pour faire les requêtes sur le serveur

1. Introduction

Cet exemple consiste à développer une grande partie de l'exemple de la librairie « cultura » vu dans le cours d'architecture.

Le paragraphe §2 suivant est un aide mémoire pour moi-même.

2. Exécution de cet exemple

Lancer Eclipse :
C:\jacques\LOGICIELS\jee-2019-06\eclipse\eclipse.exe

Le workspace à utiliser :
C:\jacques\04_DEVELOPPEMENT_JEE\eclipse-workspace2020

Lancer EasyPHP et configurer le serveur apache sur le port 80.
Vider la base de données «exemple10 ».

Dans l'onglet Servers, lancer le serveur : start (Le premier)

Dans le navigateur :
L'adresse URL pour se connecter à cette petite application Web est :
http://localhost:8080/Exemple10Web/

Pour consulter le contenu de la BD en H2 :
- Arréter le serveur
- Data Source Explorer > H2DS3 > clic-droit > Connect > aller dans l'arborescence > tables > clic-droit > Data
- Disconnect (on ne peut pas avoir 2 connexions en même temps)

Pour consulter le contenu de la BD en MySQL :
dans EasyPHP

Le DrivenMessage n'étant pas en "Statefull" il écoute le canal et écrit en BD l'information lue dans le canal.

L'IHM Admin (page du navigateur) refresh 500ms et interroge la BD via un stateless

Cet exemple contient un exemple de WebService et un client pur java qui utilise ce WebService : Exemple10_ClientWS.

3. Les cas d'utilisation

Voici les cas d'utilisation que nous allons vérifier ensemble en exécutant cet exemple :

- Création de 2 comptes clients en base de données : LAFONT et DUPONT
- Initialisation des livres dans la base de données (initialisation d'un stock de livre en dur)

- Connexion d'un client sur un browser et connexion d'un autre client dans un autre browser ou en navigation privée dans le même browser
- Réservation de livre en // par les 2 clients : gestion du panier
- Validation des commandes et vérifier que ces commandes ont bien été passées
- Afin de vérifier l'exclusivité en même temps de la réservation d'un même livre par deux clients en même temps, l'utilisateur "TORTUE" attend 10 secondes dans le code de réservation. Avec "synchronized" on montre que cela marche très bien. En modifiant le code sans "synchronized" on montre que l'on peut réserver le même livre en même temps.
- Envoi de messages successifs dans la file afin de vérifier que chacun des clients se partagent bien la file et consomme les messages à tour de rôle
- Envoi d'un message (en Ajax) afin de vérifier le déclenchement du DrivenMessage de chacun des utilisateurs. Le rôle de la requête est de lire dans la base de données.
- Création d'un compte utilisateur en utilisant un Webservice.

4. L'environnement de développement utilisée

Pour réaliser cet exemple, j'ai utilisé :

- Eclipse IDE for Entreprise JavaDevelopers. Version : 2019-06 (4.12.0)
- Dans lequel, j'ai installé le serveur JBoss Wildfly 17
- Le Java utilisé est le JSE 1.8
- La norme EJB utilisée est EJB 3.x
- Le ORM utilisée est Hibernate
- La datasource utilisée est une jta-data-source H2 (local au Serveur) avec une connexion JDBC
- La configuration du serveur utilisé est standalone-full.xml, indispensable pour utilisée les Driven Message.

Le IDE Eclipse se trouve à http://www.eclipse.org/downloads/packages/file/55190

Plusieurs étapes de configuration ont été nécessaires :

- Installation et téléchargement le serveur d'application (File>New>Other>Server>Server>Next>Wildfly17>Next>Local) Le téléchargement prend plusieurs minutes.
- JAVA_HOME (pour utiliser le add-user.bat afin de créer un compte utilisateur)
- Sous Chrome mais pas sous Firefox, http://localhost:9990/console/index.html afin d'accéder à l'ihm de configuration du serveur d'application pour créer le data-source pour la BD en local : (Subsystems>Datasource>Add Datasource)

  JNDI : java:/**H2DS3**
  Driver : H2
  jdbc:h2:~/test_db;DB_CLOSE_DELAY=0

- Configurer le lancement du serveur d'application à partir de Eclipse (Servers>Wildfly) :

  -mp "C:\Users\jlaforgu\wildfly-17.0.1.Final\modules" org.jboss.as.standalone -b localhost --server-config=standalone-full.xml -Djboss.server.base.dir=C:\Users\jlaforgu\wildfly-17.0.1.Final\standalone -Dlogging.configuration=file:C:/Users/jlaforgu/wildfly-17.0.1.Final/standalone/configuration/logging.properties

- Préciser dans tous les projets créés la version 1.8 de Java (BuildPath>JavaCompiler+Project Facets)

- Préciser dans tous les projets créés le SA à utiliser (BuildPath>Traget Runtime>Wildfly 17/0 Runtimes

5. Les projets de l'exemple

Pour réalise un SI basé sur la technologie J2EE, il est nécessaire de créer sous Eclipse plusieurs projets :

DrivenMessageEJB	les driven messages doivent être créés dans un projet à part des EJB
Exemple10EAR	le projet de déploiement Enterprise Application ARchive de notre exemple
Exemple10EJB	le projet contenant l'implémentation des EJB
Exemple10EJBClient	le projet contenant les interfaces distantes des EJB
Exemple10JPA	le projet contenant les données dont celles devant persister une base de données (Java Persistence API)
Exemple10Web	le projet contenant la couche de présentation et de contrôle (JSP et Servlet)

L'adresse URL pour se connecter à cette petite application Web est :
http://localhost:8080/Exemple10Web

Pour le projet JPA de l'exemple, il faut configurer le persistance.xml (Exemple10JPA>JPA Content>persistence.xml :

- General> Mettre les classes de données devant être gérées en persistence dans "Managed Classes"

- Connection> JTA data source: java:/**H2DS3**

Pour le projet Exemple10EAR > Properties >Deployment Assembly> Add..
On y ajoute les projets devant être déployés sur le serveur d'application qui sont, pour notre exemple :

- DrivenMessageEJB
- Exemple10EJB
- Exemple10EJBClient
- Exemple10JPA
- Exemple10Web

Sur chaque projet, (BuildPath>Project, il faut préciser les dépendances entre les projets.
Dans notre exemple :

Exemple10Web dépend de **Exemple10EJBClient** et **Exemple10JPA** car les servlets utilisent les EJB à travers les interface Remote définies dans Exemple10EJBClient et les données utilisées dans les interfaces.

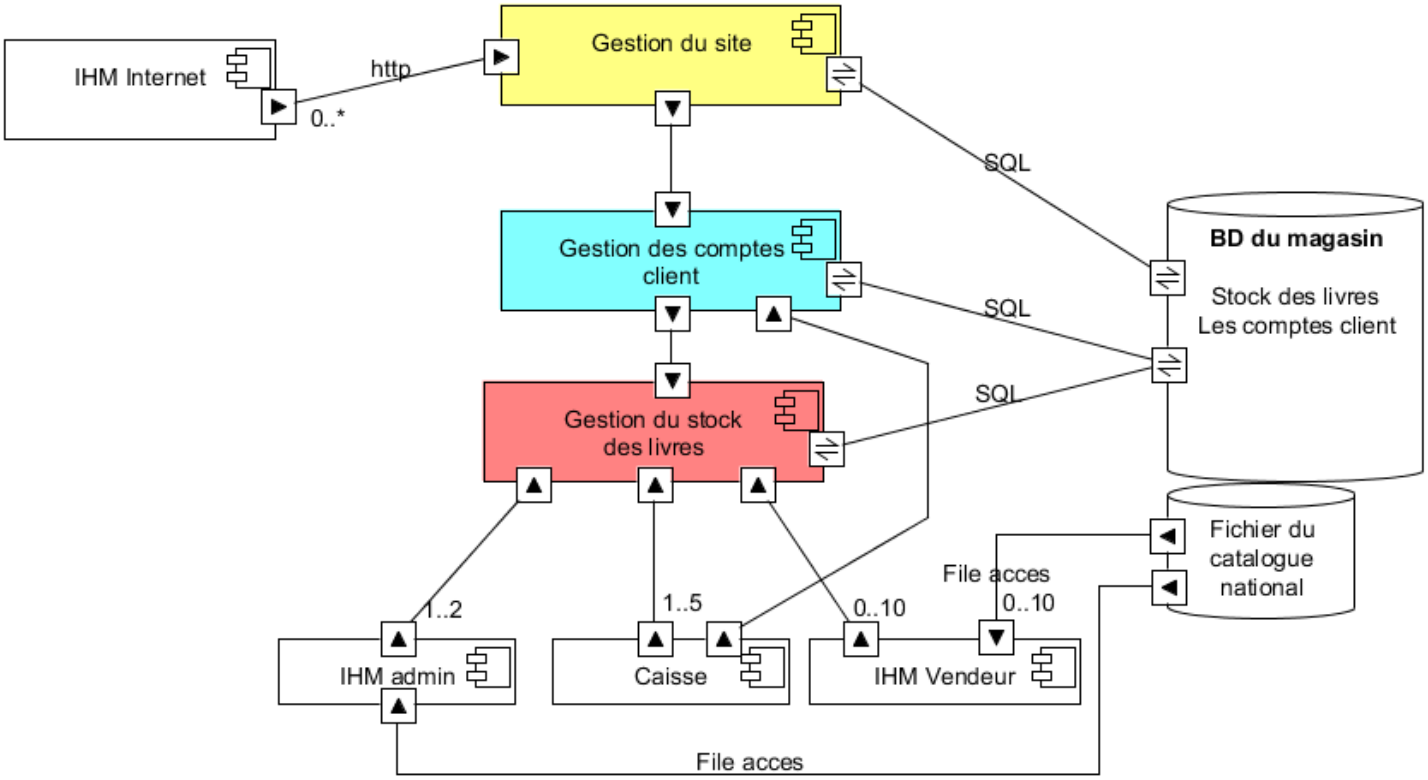
Exemple10JPA n'est dépendant d'aucun projet.

Exemple10EJBClient est dépendant de **Exemple10JPA** contenant les données utilisées dans les interfaces.

Exemple10EJB est dépendant de **Exemple10JPA** contenant les données utilisées dans le code métier.

DrivenMessageEJB est dépendant de **Exemple10JPA** pour mettre à jour la BD.

6. Configuration architecturale



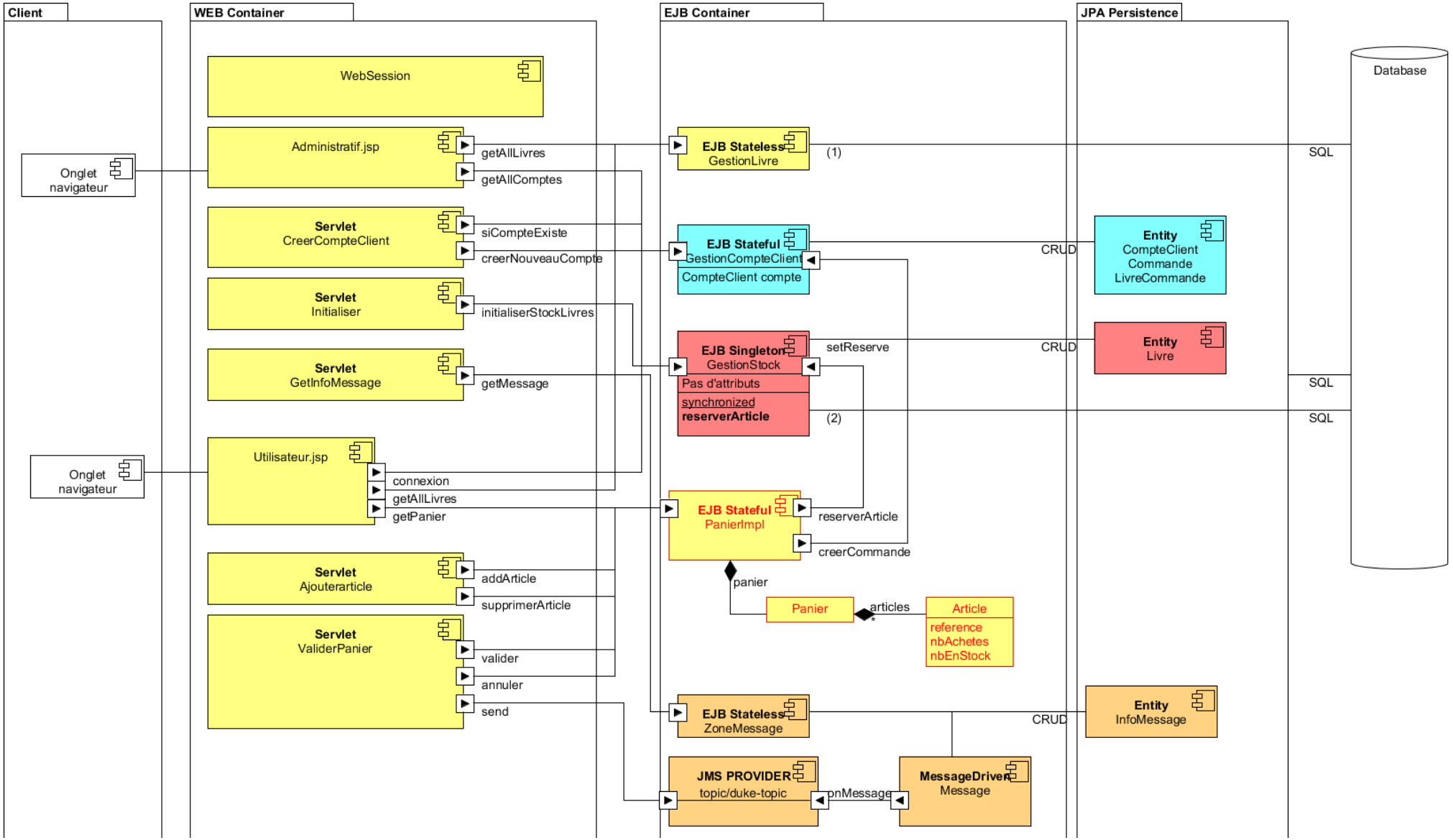
Afin de limiter le développement de ce SI, nous faisons les choix suivants :

- Le stock des livres en BD est initialisé avec l'ihm d'administration.
- Le livre est caractérisé par un id unique, une référence, un titre, et son état de réservation.
- Le compte client est caractérisé par un id unique, son nom, et ses commandes.
- Une commande est caractérisée par un id unique, un numéro, et les livres commandés.
- L'ihm d'administration permet de :
  - Créer un compte client
  - Afficher le contenu de tous les comptes clients
  - Initialiser la base de données avec un stock de livres
  - L'ihm est notifié à chaque fois qu'un utilisateur valide une commande
- L'ihm de chaque utilisateur permet de :
  - Se connecter au serveur en s'authentifiant avec son nom
  - De réaliser une commande de réservation de livre

- D'afficher l'historique des commandes de l'utilisateur

- On ne fait pas de développement pour la caisse, et l'ihm vendeur.

7. Architecture technique


Client
WEB Container
EJB Container
JPA Persistence
Database
WebSession
Administratif.jsp
getAllLivres
getAllComptes
Onglet navigateur
Servlet CreerCompteClient
siCompteExiste
creerNouveauCompte
Servlet Initialiser
initialiserStockLivres
Servlet GetInfoMessage
getMessage
Utilisateur.jsp
connexion
getAllLivres
getPanier
Onglet navigateur
Servlet Ajouterarticle
addArticle
supprimerArticle
Servlet ValiderPanier
valider
annuler
send
EJB Stateless GestionLivre
(1)
SQL
EJB Stateful GestionCompteClient
CompteClient compte
CRUD
Entity CompteClient Commande LivreCommande
EJB Singleton GestionStock
Pas d'attributs
synchronized reserverArticle
setReserve
(2)
CRUD
Entity Livre
SQL
SQL
EJB Stateful PanierImpl
reserverArticle
creerCommande
panier
Panier
articles
Article
reference
nbAchetes
nbEnStock
EJB Stateless ZoneMessage
CRUD
Entity InfoMessage
JMS PROVIDER
topic/duke-topic
onMessage
MessageDriven Message

COMMENTAIRES DU SCHEMA EN COURS

8. Commentaires du code

Les EJB sont utilisés dans les pages JSP avec les balises d'utilisation d'un bean :

```
  <jsp:useBean id="compte" scope="session"
class="fr.cnam.ejb.GestionCompteClient"/>
  <jsp:useBean id="zonemessage" scope="session"
class="fr.cnam.ejb.ZoneMessage"/>
  <jsp:useBean id="gestionlivre" scope="session"
class="fr.cnam.ejb.GestionLivre"/>
  <jsp:useBean id="stock" scope="application"
class="fr.cnam.ejb.GestionStock"/>
```

Cette balise jsp garde en mémoire de la session la connexion avec les EJB Stateful

Toutes les actions réalisées dans une page se fait en utilisant des **servlets**. Ces servlets accèdent au EJB à travers les Session http gérée par Serveur d'Application, propre à chaque utilisation si le scope est "session", commun si le scope est "application".

```
  @WebServlet("/ValiderPanier")
  public class ValiderPanier extends HttpServlet {
  […]
  HttpSession session = request.getSession(true);
     PanierImplRemote panier =
(PanierImplRemote)session.getAttribute("panier");
     GestionCompteClientRemote compte =
(GestionCompteClientRemote)session.getAttribute("compte");
     GestionStockRemote stock = (GestionStockRemote)
(getServletContext().getAttribute("stock"));

     if ((String)(request.getParameter("Valider"))!=null) {
      boolean valide = panier.valider(compte,stock);

      if (valide) {
       String texte = "COMMANDE de "+compte.getCompte().getNom();
      }
     }
     if ((String)(request.getParameter("Annuler"))!=null) {
      panier.annuler(compte,stock);
     }

     // On recharge la page
     response.sendRedirect("Utilisateur.jsp");
```

Exemple d'un EJB Stateful (le panier pour valider une commande)

```
  @Stateful
  @LocalBean
  public class PanierImpl implements PanierImplRemote {

   private Panier panier;
```

```
      public PanierImpl() {panier = new Panier();}

      public void addArticle(String reference) {
       panier.addArticle(reference);
      }

      public void supprimerArticle(String reference) {
       panier.supprimerArticle(reference);
      }

      public boolean valider(GestionCompteClientRemote
compte,GestionStockRemote stock) {
       return stock.reserverArticle(compte,panier);
      }

      public void annuler(GestionCompteClientRemote
compte,GestionStockRemote stock) {
       panier.clear();
      }
  }
```

Dans ce cas le EJB Stateful PanierImpl appelle le EJB Singleton GestionStock pour réaliser la réservation des livres commandés.

Dans le EJB SIngleton GestionStock la méthode reserverArticle est synchronized pour éviter le bug de réservation en même temps d'un livre par deux utilisateurs :

```
  synchronized public boolean reserverArticle(GestionCompteClientRemote
compte,Panier panier)
  […]
```

Dans cette méthode il y a du code en paramètre que l'on peut activer pour démontrer l'utilité de cette synchronisation (décommenter le code et enlever *synchronized*).

Les données **Livre** et **CompteClient** sont des **Entity**. Ceci permet la création automatique des tables en BD et permet de ne pas écrire d'instruction SQL pour réaliser leurs créations et leurs mises à jour en Base de Données.

Exemple d'une Entity, **CompteClient** :

```
  @Entity
  public class CompteClient implements Serializable {

   @Id
   @GeneratedValue
   private Integer id;

   @Column
   private String nom;   // Identifiant du client

   @OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER,orphanRemoval =
true)
   @JoinColumn(name="idCompteClient")
```

```
  private Set<Commande> commandes;  // Les commandes réalisées par le
client
```

Persistence.xml :

```
  <?xml version="1.0" encoding="UTF-8"?>
  <persistence version="2.1"
xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
   <persistence-unit name="Exemple10JPA">
    <jta-data-source>java:/H2DS3</jta-data-source>
   <class>fr.cnam.data.Livre</class>
   <class>fr.cnam.data.CompteClient</class>
   <class>fr.cnam.data.Commande</class>
   <class>fr.cnam.data.LivreCommande</class>
   <class>fr.cnam.data.InfoMessage</class>
   <properties>
         <property name="hibernate.dialect"
value="org.hibernate.dialect.HSQLDialect" />
         <property name="hibernate.show_sql" value="false" />
         <property name="hibernate.format_sql" value="false" />
         <property name="hibernate.hbm2ddl.auto" value="none" />
         <property name="hibernate.search.default.directory_provider"
value="ram" />
         <property name="hibernate.search.indexing_strategy"
value="manual" />
      </properties>
   </persistence-unit>
  </persistence>
```

La 1ère fois afin de créer les tables automatiquement :
<property name="hibernate.hbm2ddl.auto" value="**create**" />

Schémas des tables créées :

```
  CREATE TABLE LIVRE (
    ID INTEGER NOT NULL,
    REFERENCE VARCHAR(255),
    RESERVE BOOLEAN,
    TITRE VARCHAR(255)
  );

  CREATE TABLE COMPTECLIENT (
    ID INTEGER NOT NULL,
    NOM VARCHAR(255)
  );

  CREATE TABLE COMMANDE (
    ID INTEGER NOT NULL,
```

```
    NUMERO INTEGER,
    IDCOMPTECLIENT INTEGER
  );

  CREATE TABLE LIVRECOMMANDE (
    ID INTEGER NOT NULL,
    NUM INTEGER NOT NULL,
    REFERENCE VARCHAR(255),
    IDCOMMANDE INTEGER
  );

  CREATE TABLE INFOMESSAGE (
    ID INTEGER NOT NULL,
    MESSAGE VARCHAR(255)
  );
```

Pour mettre en place d'un mécanisme de MOM, on crée les files de message et le contexte d'utilisation de JMS. Pour cela on crée un singleton, **ConfigurerJMS**, avec la création des "destinations" de JMS :

```
  @JMSDestinationDefinitions(
          value = {
              @JMSDestinationDefinition(
                name = "java:jboss/exported/jms/topic/duke-topic",
                      interfaceName = "javax.jms.Topic",
                      destinationName = "jms/duke-topic"
              ),
              @JMSDestinationDefinition(
                name = "java:jboss/exported/jms/queue/duke-queue",
                      interfaceName = "javax.jms.Queue",
                      destinationName = "jms/duke-queue"
              )
          }
  )

  @Singleton
  @LocalBean
  public class ConfigurerJMS implements ConfigurerJMSRemote {
  […]
```

On crée un DrivenMessage **Messages** :

```
  @MessageDriven(
    activationConfig = {
  @ActivationConfigProperty(
      propertyName = "destination", propertyValue =
"java:jboss/exported/jms/topic/duke-topic"),
  @ActivationConfigProperty(
      propertyName = "destinationType", propertyValue = "javax.jms.Topic")
    },
    mappedName = "java:jboss/exported/jms/topic/duke-topic")
  public class Messages implements MessageListener {
  [..]
```

```
  public void onMessage(Message message) {
          // TODO Auto-generated method stub

          String mess="*******";
          TextMessage textMessage =(TextMessage)message;
          try{
           mess = textMessage.getText();
           }catch(Exception ex) {}
          System.out.println("MESSAGE : "+mess);

          InfoMessage info = new InfoMessage(mess);
          info.sauver();
      }
```

Ce DrivenMessage se comporte comme un Stateless. Il stocke le message dans une table en base de données.

Ce message s'affichera automatiquement dans la page de l'administration.

Pour réaliser un rafraichissement de la page (Administration.jsp) dynamiquement en fonction de la modification de la couche métier, on utilise le javascript et jquery :

```
  function refreshMessage()
  {
  $.ajax({
          url : 'GetInfoMessage?'+new Date().getTime(),
          type : 'GET',
          dataType : 'html',
          success : function(code_html, statut){ // success est toujours en
place, bien sûr !

             document.getElementById("MessageText").value = code_html;
            }
       });
  }
  window.setInterval("refreshMessage();",500);
```

```
  @WebServlet("/GetInfoMessage")
  public class GetInfoMessage extends HttpServlet {
  […]
  HttpSession session = request.getSession(true);

    ZoneMessageRemote zonemessage =
(ZoneMessageRemote)session.getAttribute("zonemessage");
    if (zonemessage==null)response.getWriter().println("zonemessage est
NULL");
    else {
     String s = zonemessage.getMessage();
     response.getWriter().println(s);
    }
```

ZoneMessage est un Stateless qui va récupérer le message stocké en BD.

Pour le WebService :

Création du web service dans Exemple10Web :

Java Resources / src / org.example.www.CompteClient / **ComteCllientSOAPImpl.java**