EJB 3.1

pour JEE 6, 7

(avec JPA,CDI, RMI et JMS)

Table des matières

I - Présentation (EJB 3 et JEE)

1. EJB (Enterprise JavaBean) - présentation

Fonctionnalités des EJB *(valeurs ajoutées)*

- ***Appels distants possibles*** (via RMI-over-IIOP *ou SOAP*)
- ***Bon support*** pour ***transactions distribuées***
- ***Contrôle d'accès (authentification, ...)***
- ***Module*** réutilisable d'***objets "métier"***
- ***Standard J2EE/JEE5*** supporté par beaucoup de serveurs (WebSphere, WebLogic, JBoss, Jonas, ...)

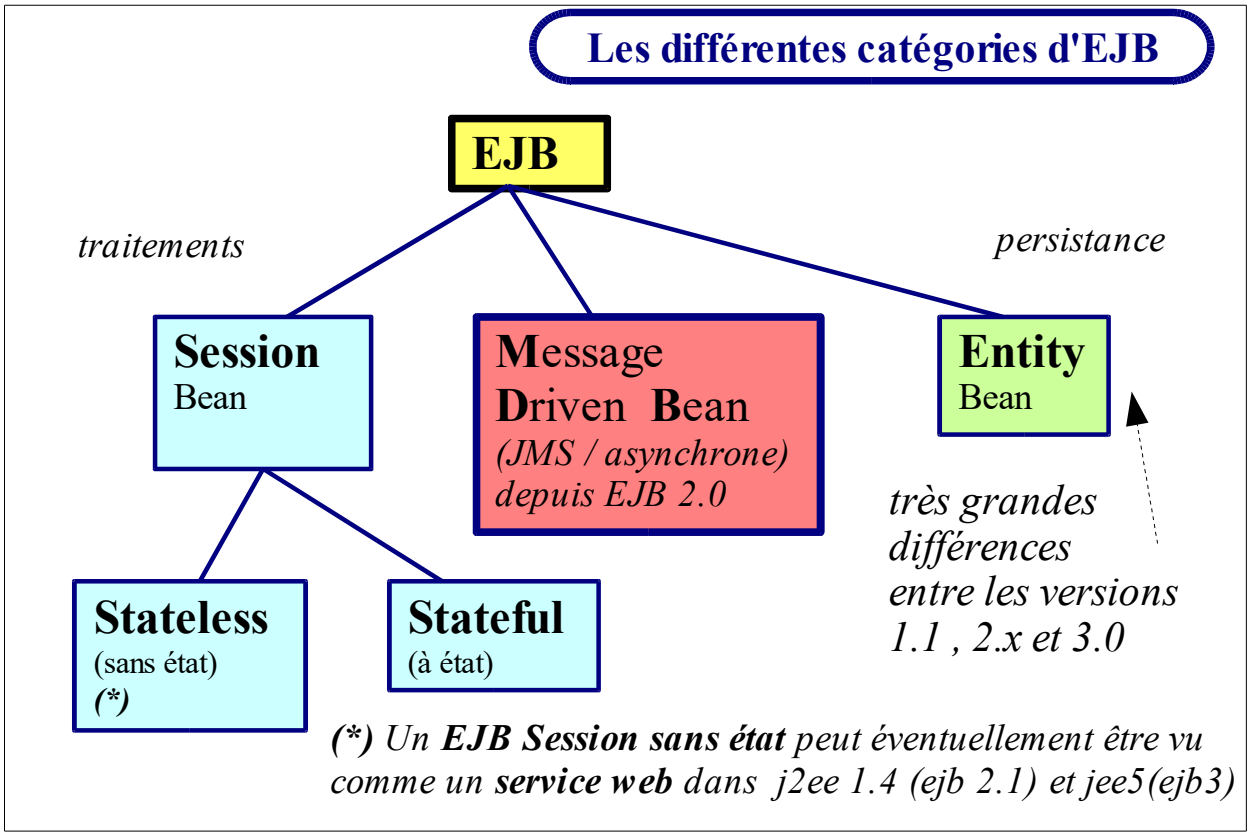
Présentation des **EJB**

Un **EJB** correspond essentiellement à un **objet métier** (***traitement applicatif*** ou ***entité persistante***).

Le sigle **EJB** désigne avant tout une **spécification de composant métiers**: *comment ils doivent être écrit* et *le contrat qu'ils doivent respecter avec le conteneur EJB*

Principal objectif du framework "EJB":

-ne programmer que l'aspect "métier" des composants.
le serveur d'application (avec son container d'EJB) **prend en charge les aspects techniques** (sécurité, multi-tâches, transactions).



Différents types d'EJB

• **EJB Session** : Ces composants correspondent aux points d'entrée en mode synchrone des "***traitements métiers et applicatifs***". Ils sont étroitement lié à la session d'un utilisateur.

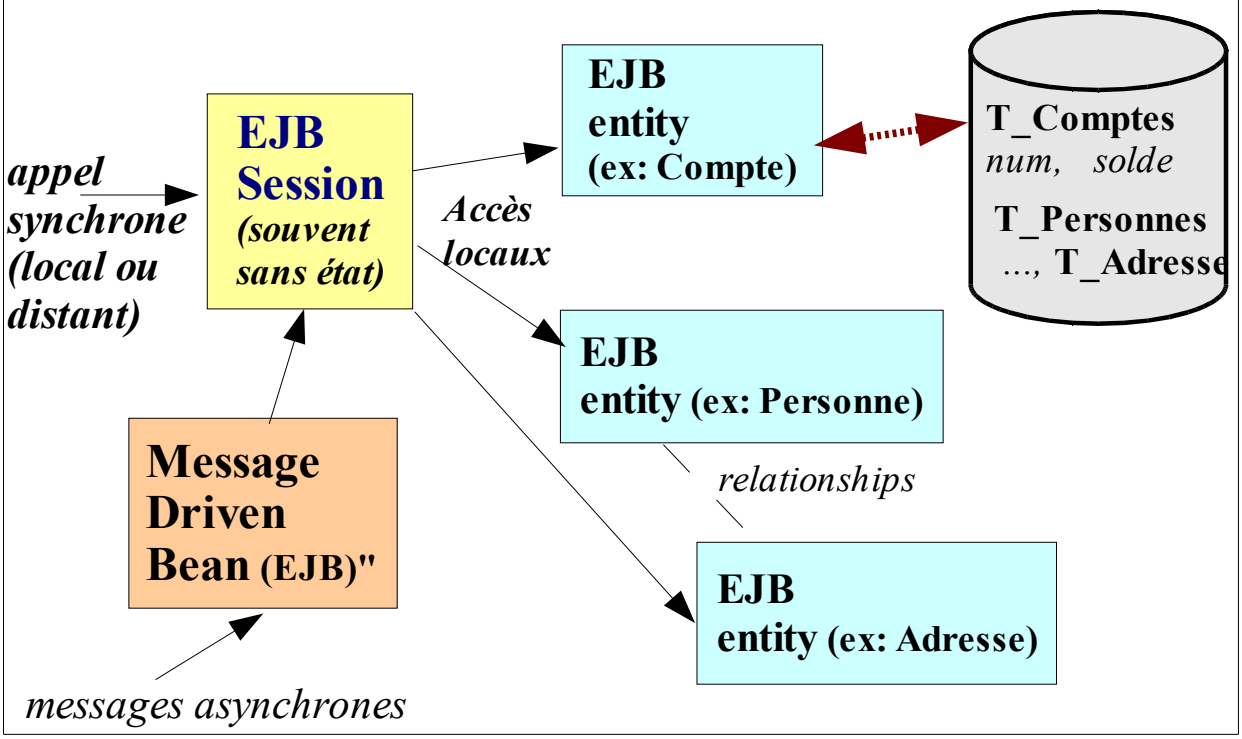
---> temporaires et non-partagés

•**EJB Entity** : Ces composants correspondent aux "***entités fondamentales du métier***". Ils représentant des données partagées par tous les utilisateurs du système

---> persistants et partagés

•**EJB Piloté par Messages** : Ces composants correspondent aux points d'entrée en mode **asynchrone** des "traitements applicatifs". Ils sont généralement déclenchés suite à la réception d'un message JMS ou SOAP. ---> temporaires & asynchrones

Relations classiques entre EJB d'un même module



Stateless *(sans état)* vs. Stateful *(à état)*

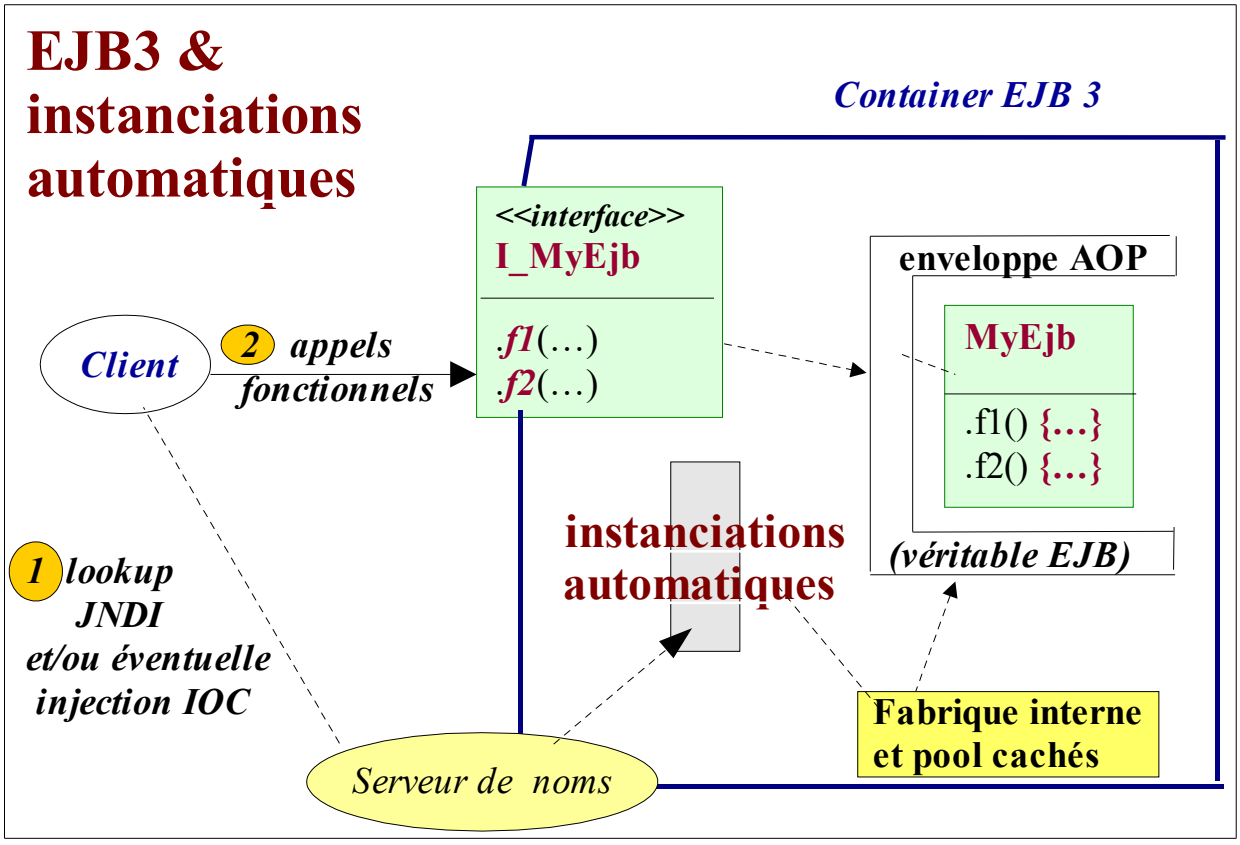
- (Ejb Session) **stateless** :
  - sans état, aucune donnée métier n'est conservée dans la mémoire de l'EJB entre deux appels (successifs) émanant d'un même client .
  - sert à factoriser des traitements atomiques entre les clients . Tous les paramètres nécessaires doivent être précisés d'un coup lors d'un appel d'une méthode autonome.
  - très bien optimisés par les serveurs d'application ➔ bonnes performances.
- (Ejb Session) **stateful** :
  - avec état conversationnel, les données internes du bean sont conservées entre deux appels (ex: caddy électronique, ...).
  - sert à gérer une session utilisateur.

1. Fabrication automatique

L'essentiel de la configuration s'effectue via des annotations Java>=5.

Les descripteurs de déploiement (fichiers de configuration XML / ejb-jar.xml , ...) peuvent se trouver très simplifiés si on ne les utilise que pour **compléter la configuration**.

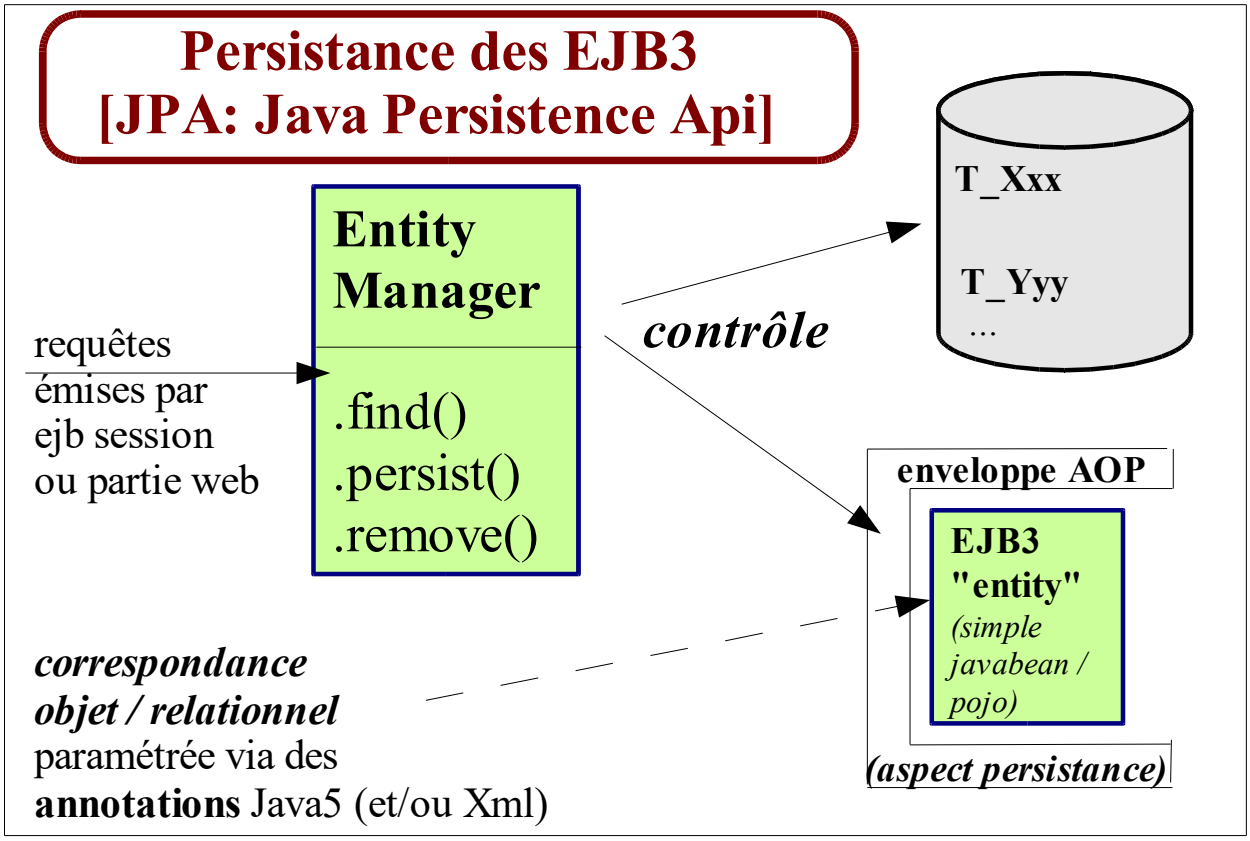
Les EJB entités ont été complètement chamboulés ==> nouvelle Api ***JPA*** *(Java Persistence Api)* très proche de Hibernate .



NB:

- L'**instanciation automatique des EJB3** est très pratique.
- Sa réelle mise en oeuvre au sein d'un serveur d'application est cependant un point crucial qui demande à être **bien optimisé**. .

2. Persistance des EJB "entity"



3. Grands traits des EJB3

3.1. Terminologie

client local ==> situé dans la ***même JVM*** (Machine Virtuelle Java) que l'EJB considéré.

	... EJB2.1	.... EJB3
Vue cliente d'un	**Local** or **Remote** ***interface***	**Business** ***interface*** (local or remote)

3.2. Catégorisation via annotations java5

La **vue cliente d'un EJB3** (appelée "***Business interface***") n'est en fait qu'une ***simple interface java*** qui n'a pas besoin d'hériter d'une interface spécifique aux EJB (telle que EJBHome ou EJBObject des EJB2) .
D'autre part, cette "*business interface*" des *EJB3* n'est pas tenue de remonter explicitement des exceptions de type *java.rmi.RemoteException*.

De façon à spécifier le type d'un EJB3 , on utilise une des **annotations** suivantes:

@Stateful	EJB3 **session à état**
@Stateless	EJB3 **session sans état**
@Entity	EJB3 **entité** *(lié à JPA et contrôlé par PersistenceManager)*
@MessageDriven	EJB3 **piloté par messages asynchrones**

Ces annotations sont à placer au niveau de la classe d'implémentation.

En outre, certaines **annotations** permettent de préciser quels sont les **modes d'invocation possibles** d'un EJB session **:**

@Remote	**accessible à distance (via RMI-over-IIOP)**
@Local	**accessible localement (depuis même JVM)**
@WebService	**accessible à distance (via SOAP) en tant que service Web**

Les annotations @Remote et @Local peuvent être placées sur l'interface ou bien sur la classe d'implémentation de l'EJB session.

Les spécifications EJB3 indiquent qu'on ne peut pas placer @Remote et @Local au même endroit. Par conséquence, si un EJB doit pouvoir être accessible à la fois localement et à distance, on doit alors configurer plusieurs interfaces (l'une avec @Local , l'autre avec @Remote avec héritage possible). Ceci permet de différentier la liste des méthodes exposées localement et à distance.

3.3. Un petit exemple (très simple):

"Business" interface de l'EJB :

```
package myejb;

public interface Calculator {
        public int add(int x, int y);
        public int subtract(int x, int y);
        public int divide(int x, int y);
}
```

Classe interne de l'EJB:

```
package myejb;
import javax.ejb.Remote;
import javax.ejb.Stateless;

@Stateless
@Remote
public class CalculatorBean implements Calculator
{
  public int add(int x, int y)   {     return x + y;   }
  public int subtract(int x, int y)   {     return x - y;   }
  public int divide(int x, int y)   {     return x / y;   }
}
```

code client (testé avec JBoss 4)

```
public static void test_calculator() throws Exception
  {
    System.out.println("********** test_calculator *********");
    InitialContext ctx = new InitialContext();
    Calculator calculator = (Calculator) ctx.lookup("test_ejb3/CalculatorBean/remote ou ...");

    System.out.println("1 + 1 = " + calculator.add(1, 1));
    System.out.println("1 - 1 = " + calculator.subtract(1, 1));
  }
```

NB:

- Bien qu'ayant la sémantique d'un appel distant , l'interface de l'EJB (vue cliente) est une simple interface locale .
- Un lookup suivit d'un casting Java suffit (plus besoin de PortableRemoteObject.narrow() pour les EJB3).
- Les noms jndi (à passer en paramètre de lookup()) ont été normalisés depuis JEE6 (Jboss7 , …) et ont maintenant la forme suivante :

  "java:app/<module-name>/<bean-name>[!<fully-qualified-interface-name>]"

  depuis l'application ou bien

"ejb:<app-name>/<module-name>//<bean-name>!<fully-qualified-classname-of-the-remote-interface>"

en accès distant (@Remote).

4. JEE en tant qu'ensemble d'API & conteneur JEE

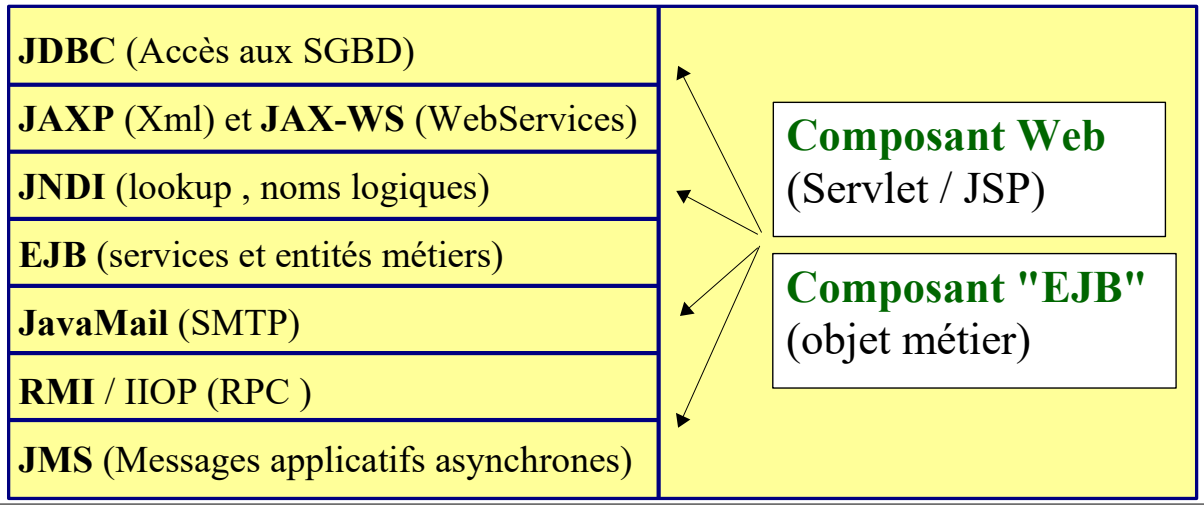
JEE (*signifiant Java Enterprise Edition*) peut être vu comme un ensemble d'API permettant de développer des applications évoluées à déployer sur un serveur d'entreprise.
Les API de JEE se rajoutent à celles du JDK (base JSE) . Elles concernent essentiellement les aspects "présentation WEB" , "EJB" , ... et "Services Web" .

JEE & API

Java EE (+ EJB,JSP,Servlet,JMS, ….)

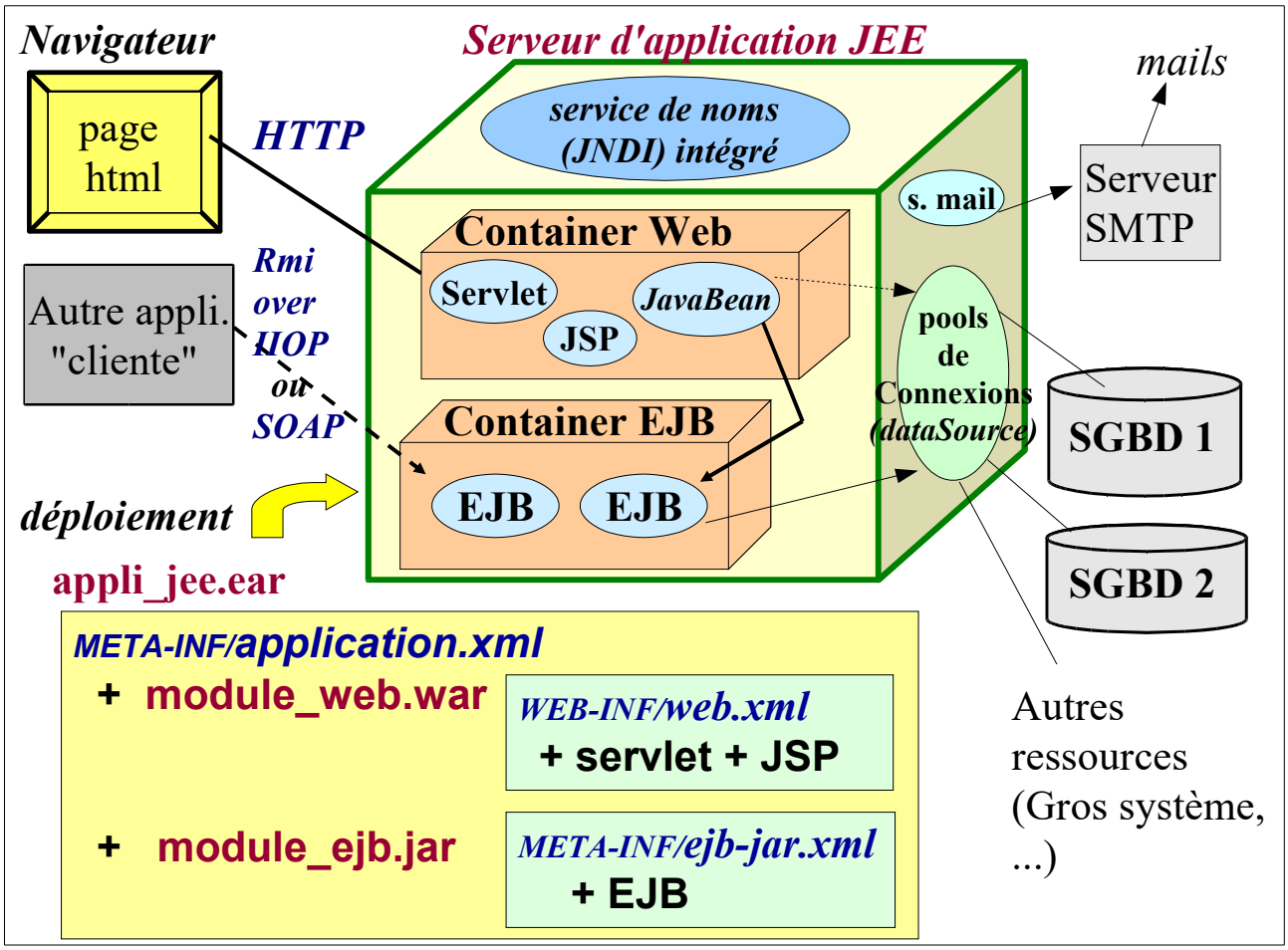
Java SE (Java Standard Edition – jdk 1.4 , 1.5 , 1.6, 1.7 , 1.8)

(JSE + **JEE)** peut être vu comme un modèle d'architecture basé sur des "**container**" qui offrent des **services techniques orthogonaux** aux ***composants métiers***:



5. Structure d'un serveur JEE

JEE peut également être vu comme un **modèle d'architecture** pour les **serveurs d'applications**.
Les **spécifications JEE** indiquent clairement le rôle des "**container**" : Ceux-ci doivent offrir aux composants applicatifs qu'ils hébergent un accès normalisé aux API standards de JEE .
Autrement dit , un **composant JEE** (*ex*: servlet , EJB, ...) fonctionne exactement de la même manière au sein des serveurs WebLogic , JBoss ou WebSphere car il peut appeler les mêmes fonctionnalités (mêmes API) et qu'il expose lui même les mêmes points d'entrées pour la gestion de son cycle de vie.



Depuis J2EE 1.2 , le **déploiement d'une application JEE est standardisé**:

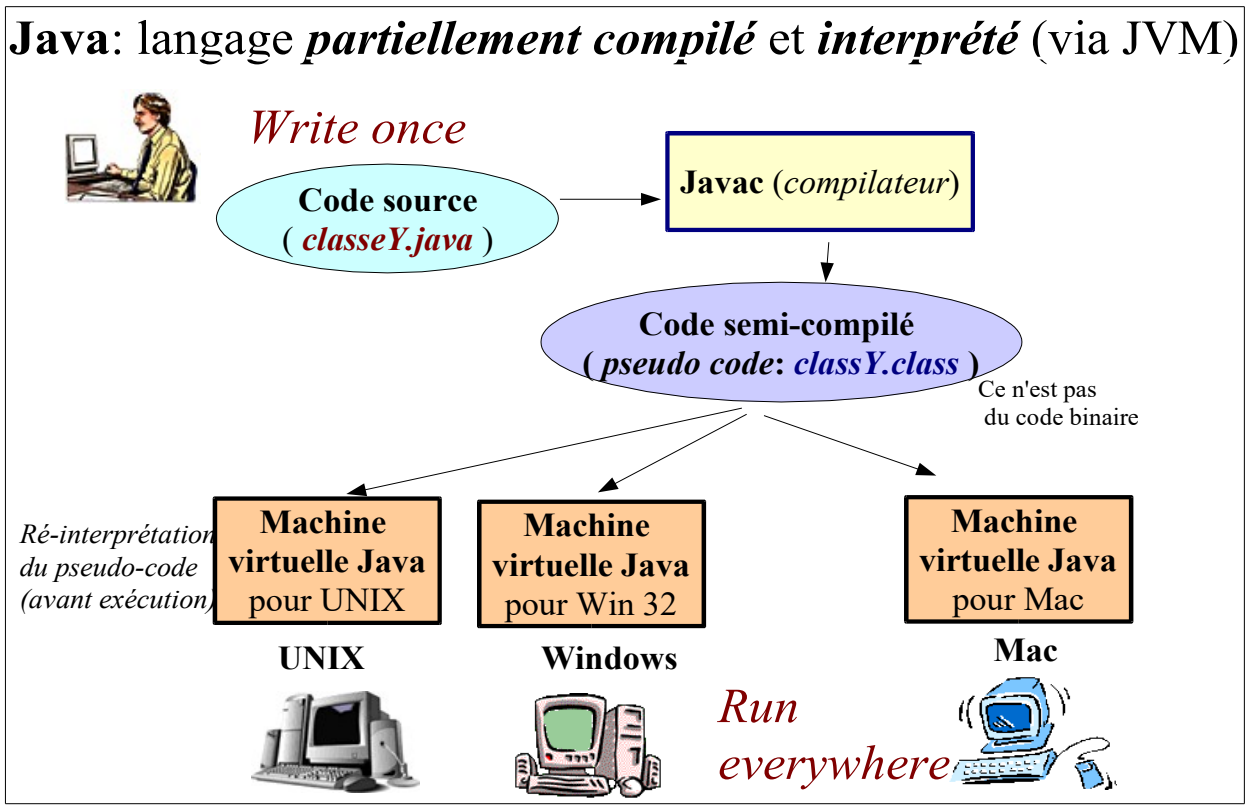
- Un fichier **".war"** (pour ***Web ARchive***) contient tous les composants "**web**" et les fichiers de configurations associés (***WEB-INF/web.xml*** , ...).
- Un fichier **".jar"** (pour ***Java ARchive***) contient tous les composants "**EJB**" et les fichiers de configurations associés (***META-INF/ejb-jar.xml*** , ...).
- Un fichier "**.ear**" (pour *Enterprise ARchive*) regroupe différentes sous archives (".war", ".jar" , ...) et un fichier de configuration globale : ***META-INF/application.xml*** dont la balise **context-root** de l'**application WEB** indique l'**URL** relative de celle-ci.

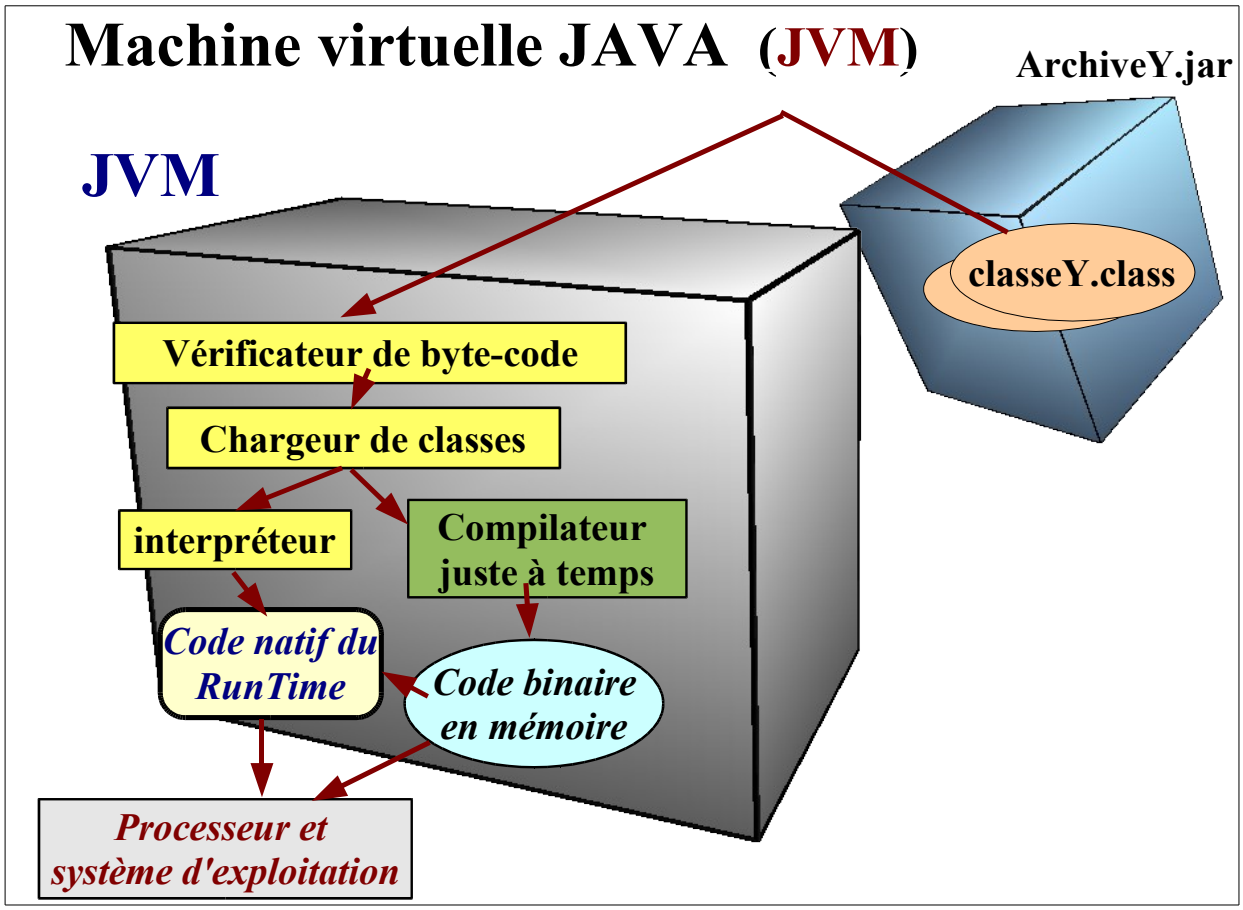
Au lieu de parler de J2EE 1.5 , Sun/JavaSoft a préféré baptiser **JEE5,6,7** les nouvelles versions des spécifications de sa plate-forme Java de niveau entreprise .

Les principaux apports de ces nouvelles versions sont les suivants:

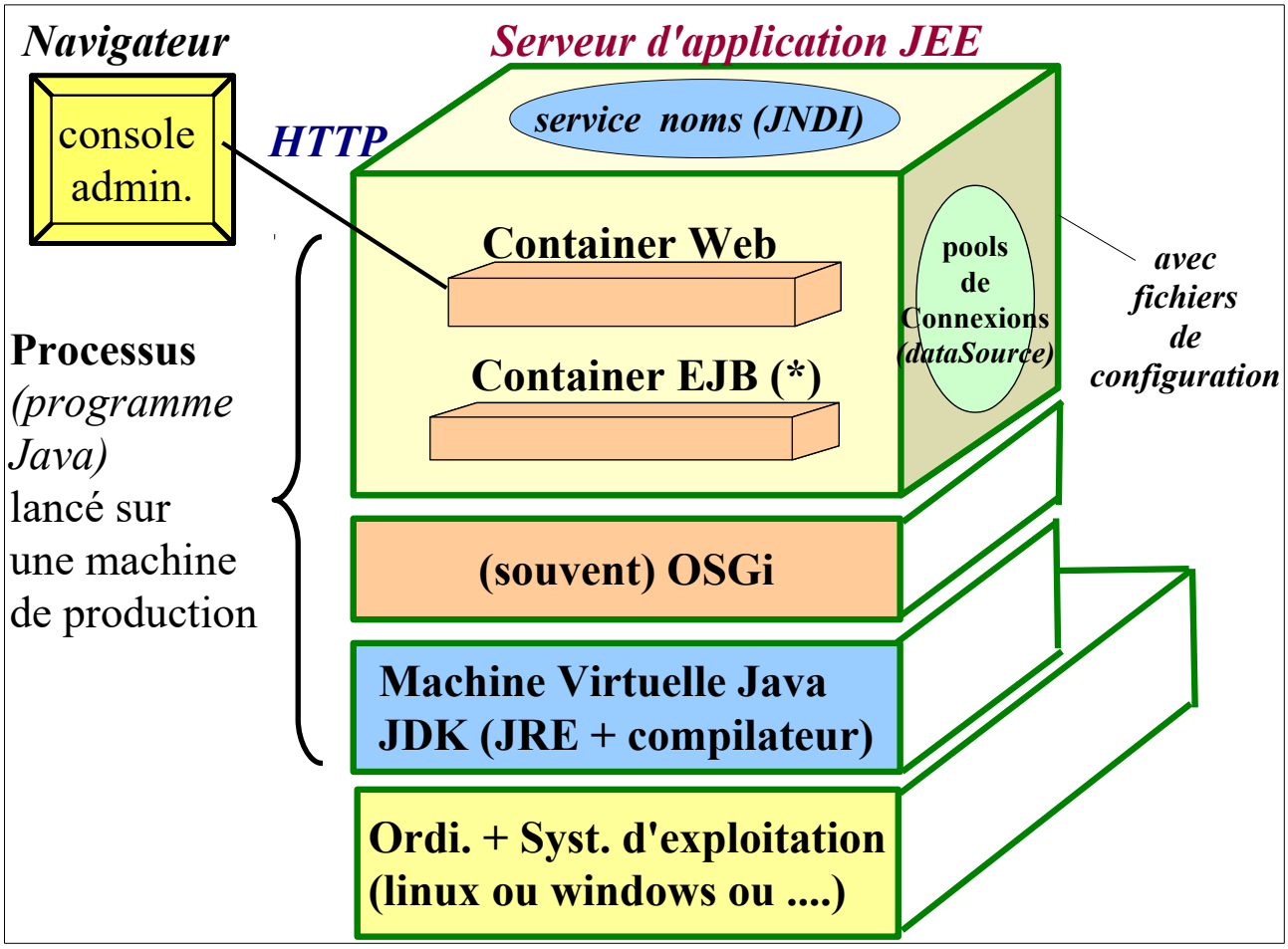
- **EJB3 , 3.1 , 3.2** (avec api **JPA 1 , 2.0 , 2.1** pour la persistance des données).
- Nouveau support des ***services WEB "Soap"*** via l'api **JAX-WS** (mieux que JAX-RPC)
  Support des **services WEB "REST"** avec l'api **JAX-RS** (v1.0 pour JEE6 v2.x pour JEE7)
- Intégration du framework **JSF** (1 puis **2**) dans la partie WEB
- **Partie Web *supportant l'injection de dépendances (@Ejb , @Resource , @Inject CDI)***
- **...**

6. Un Serveur JEE fonctionne avec une "JVM"





Un Serveur d'application JEE est avant tout un cas particulier de programme écrit en java et qui s'exécute à l'aide d'une machine virtuelle Java (JVM) .



Certains mécanismes internes des serveurs d'applications JEE ont besoin de déclencher des compilations (ex: pages JSP transformées en Servlet à compiler) . Il faut donc s'appuyer sur le JDK complet (compilateur + JRE = Java Runtime Environment) .

La version du JDK (1.4 , 1.5 , 1.6 , 1.7 ou 1.8) a une très grande importance car elle conditionne les possibilités/fonctionnalités du serveur.

Les serveurs JEE très simples (ex : TomcatEE) ne s'appuient pas sur OSGi.

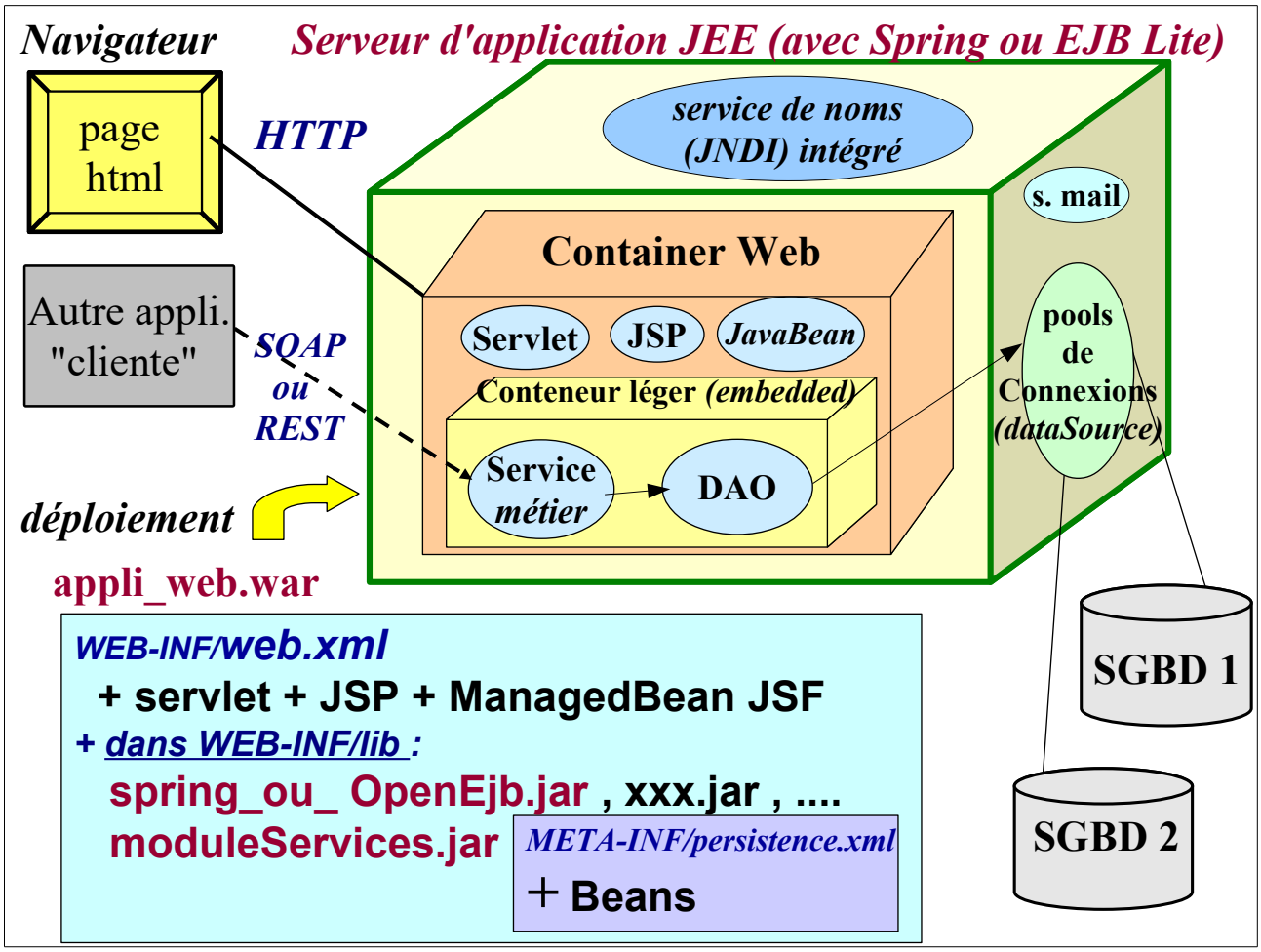
Par contre, la plupart des serveurs JEE sophistiqués (Glassfish >= 3 , Jboss >= 7 , …) s'appuient en interne sur OSGi de façon à pouvoir :

- mieux gérer des (re-)déploiements à chaud (sans arrêter / relancer l'ensemble du serveur).
- mieux gérer les inter-dépendances entre modules (partage de librairies , ...)

7. Utilisation de Spring ou d' EJB-Lite ou CDI , ...

L' utilisation d'un conteneur d'EJB (en version 2 ou 3) n'est pas du tout obligatoire.

On peut préférer utiliser un **conteneur embarqué** tel que **Spring** ou **EJB-Lite** (ex : OpenEJB) ou ...



==> principales différences:

- utilisation possible de serveurs très simples (Tomcat , TomcatEE , …)
- on ne déploie pas un ".ear" mais un ".war" qui comporte lui même des ".jar" dans sa partie WEB-INF/lib .

==> concrètement :

- Déploiement de "JSF_ou_autre + Spring_3_ou_4 + JPA/Hibernate" dans Tomcat
- Déploiement de "codeJSF_ou_autre + EJB" dans TomcatEE (avec OpenEjb / OpenJpa)
- ….

8. Evolution de JEE

Evolutions de J2EE, JEE5,JEE6

J2EE 1.0 à 1.2

Socle architecture = Servlet/JSP + JNDI + RMI + EJB
Formalisation des archives (.war, .jar, .ear)

J2EE 1.3 à 1.4 *(WebSphere 5,6 , WebLogic 7,8,9, Jboss 3.2,4)*

EJB 2.x (***MDB***, ..., Interfaces locales, ...) ,Connecteurs **JCA** (et .rar)

JEE5 *(Jdk >= 1.5 , WebSphere7 , Jboss 4.2 ou 5 , ...)*

EJB3 (config. via annotations , **J**ava **P**ersistence **A**pi)
Framework web ***JSF 1*** (Java Server Faces) , ***IOC*** , ...
JAX-WS (Api simple et efficace pour Services Web)

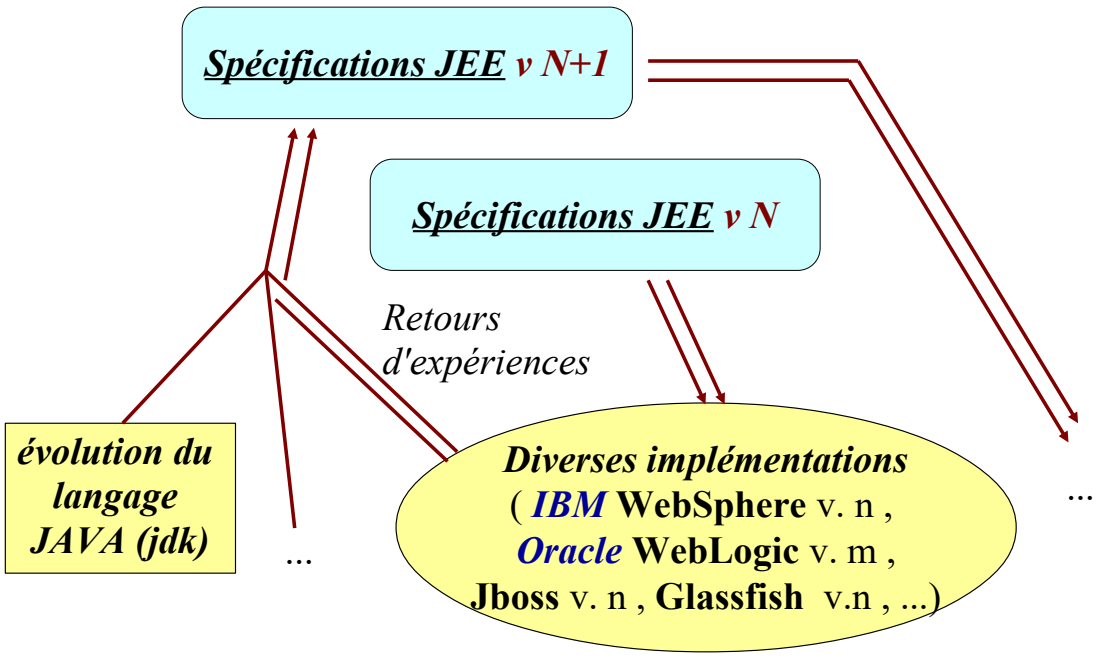
JEE6(jdk >=1.6 , Jboss 7 , Glassfish 3 , ..) :

EJB3.1 , JSF2 , annotations coté Web, JAX-RS 1 ,CDI (@Inject),...

JEE7(Glassfish 4 , Jboss WildFly 9 , ...) :

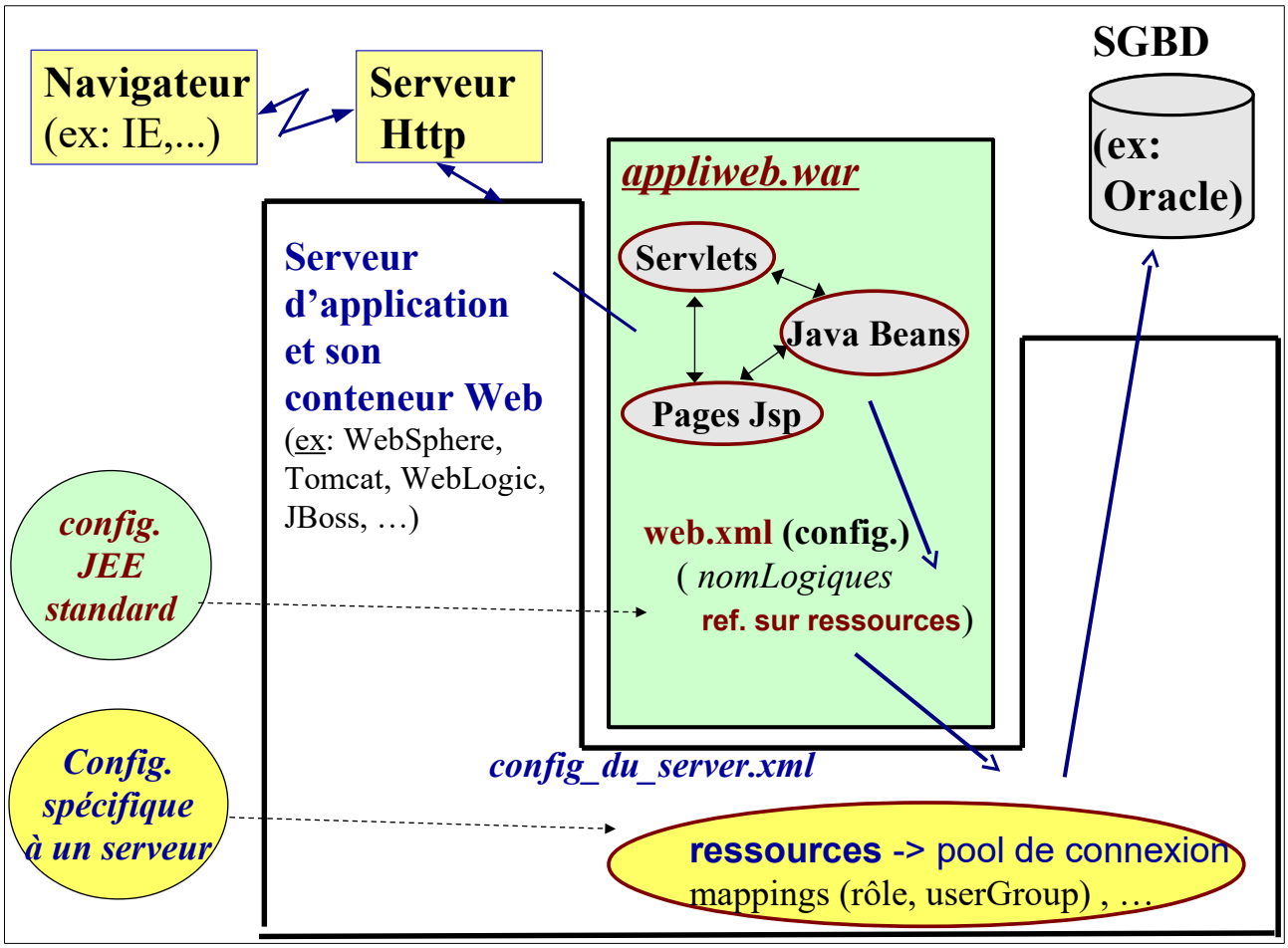
EJB3.2 , JSF2.1, JAX-RS2, @Transactional sans EJB, ...

JEE : une ***spécification*** d'architecture qui évolue



Sun/Oracle (inventeur du langage Java) pilote l'évolution de JEE

9. Spécificités selon le serveur d'application JEE



Chaque serveur d'application se configure à sa façon (avec des fichiers de configuration différents) et quelquefois avec l'aide d'une console d'administration.

Exemples : ...\glassfish4\glassfish\domains\domain1\config**domain.xml** pour glassfish

….\wildfly-9.0.2.Final\standalone\configuration**standalone-full.xml** pour Jboss

…

AU FINAL , on se retrouve "DEPENDANT DU SERVEUR d'APPLICATION" pour des CONFIGURATIONS SPECIFIQUES sur les points suivants :

- DataSource JDBC (accès au bases de données)
- Session de mail (config URL SMTP pour envoyer un mail)
- configurations JMS (Queue , ..) et JNDI (noms logiques des ressources locales ou distantes)
- sécurité JEE (realms , ...) , ….

10. Principaux serveurs d'applications (JEE)

Serveurs d'applications	*Marques/Editeurs*	*Caractéristiques*
WebSphere	**IBM**	- Produit commercial avec le support d'une grande marque. - Serveur assez sophistiqué (très paramétrable et avec une bonne console d'administration). - surtout utilisé dans les grandes entreprises (banques, assurances, ...)
WebLogic	**BEA --> Oracle**	- Autre bon produit commercial (à peu près aussi sophistiqué que WebSphere)
Jboss (4.2 , 5.1 , 7.1) puis **EAP** (payant en prod) et **wildFly** (purement Open Source)	**Jboss / Red Hat**	- Open source , existe depuis longtemps - Souvent innovant sur les technologies java (jmx , ejb3, ...) - Utilisation très simple pour les tests durant la phase de développement - console d'administration rudimentaire.
...		
Tomcat (*)	**Apache Group**	- Open source faisant office de référence sur la partie "conteneur Web". - Serveur JEE simplifié (partie "conteneur web" seulement (sans EJB)).
TomcatEE	**Apache Group**	Assemblage de "Tomcat" + "OpenEJB/OpenJPA" + "OpenWebBean" (CDI) .
GlassFish	**SUN --> Oracle**	- Serveur JEE de référence (en partie open source) assez complet et assez innovant sur certaines technologies (BPEL, ESB/JBI , ...) - Moins sophistiqué de "WebLogic" de la même marque

(*) NB: Tomcat peut être utilisé :
-soit de façon autonome en tant que mini serveur JEE (sans partie EJB)
-soit en tant que partie "conteneur web" intégrée dans un autre serveur JEE plus complet .

- Certains anciens serveurs JEE (tels que "Jonas" de OW2) ne sont plus maintenus (faute de budget).

11. Descripteur de déploiement de l'application

META-INF/application.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee"
```

```
  xmlns:application="http://java.sun.com/xml/ns/javaee/application_6.xsd"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/application_6.xsd"
    id="Application_ID" version="6">
 <display-name>myJeeApp</display-name>
 <module>
  <ejb>myJeeAppEJB.jar</ejb>
 </module>
 <module>
  <java>myJeeAppClient.jar</java>
 </module>
 <module>
  <web>
   <web-uri>myJeeAppWeb.war</web-uri>
   <context-root>myJeeAppWeb</context-root>
  </web>
 </module>
</application>
```

La valeur du paramètre "**context-root**" détermine une partie de l'URL menant à l'application :
==> *http://localhost:8080/**myJeeAppWeb***

Rappel: La partie web nécessite un fichier ***WEB-INF/web.xml*** de ce type:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
id="WebApp_ID" version="3.0">
   <display-name>myJeeAppWeb</display-name>
    ...
        <servlet>
                <servlet-name>Faces Servlet</servlet-name>
                <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
                <load-on-startup>1</load-on-startup>
        </servlet>

        <servlet-mapping>
                        <servlet-name>Faces Servlet</servlet-name>
                <url-pattern>*.faces</url-pattern>
    </servlet-mapping>
</web-app>
```

12. Eventuel fichier "META-INF/ejb-jar.xml"

Indispensable au sein des anciennes versions des EJB (1.1 , 2.0 , 2.1) , le descripteur de déploiement standard "***META-INF/ejb-jar.xml***" **n'est plus strictement obligatoire pour les EJB3**.

Pour les EJB3, le fichier "***META-INF/ejb-jar.xml***" ne vient que compléter la configuration encodée sous forme d'***annotations Java5***.

META-INF/ejb-jar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar
        xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                            http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
        version="3.0">
   <description>Test EJB3</description>
   <display-name>Test EJB3</display-name>

</ejb-jar>
```

Dans l'exemple ci-dessus quasiment vide , ce fichier peut être éventuellement complété en y ajoutant (entre autres) des sous blocs de ce genre:

```
<enterprise-beans>
<session>
...
<ejb-name>EmployeeService</ejb-name>
<ejb-class>xxx.yyy.EmployeeServiceBean</ejb-class>
...
<resource-ref>
      <description>...</description>
      <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
      <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
...
</session>
</enterprise-beans>
```

13. Versions des principales API de JEE

En règle générale , un serveur d'une version N supporte du code standard (et des configurations "standards") en versions N-1 et N-2 .

Par contre un ancien code complémentaire (non standard) de l'époque N-1 est rarement parfaitement supporté par la version N (ex : extension richfaces3 pour JSF1 / JEE5 qui n'est plus supporté par JSF2 de JEE6) .

13.1. Tableau des principales versions

API	*Fonctionnalités*	*JEE5*	*JEE6*	*JEE7*
Servlet	Cœur web/http	2.5	3.0	3.1
JSP	Pages (coté serveur)	2.1	2.2	2.2
JSF	Framework Web sophistiqué (MVC, ...)	1.2	2.0	2.2
EJB	Objet/service métier , transaction , ...	3.0	3.1	3.2
JPA	Persistance / ORM java	1.0	2.0	2.1
JMS	Envoi et réception de messages (Queue)	1.1	1.1	2.0 et 1.1
JTA	Transaction ditribuée	1.1	1.1	1.2
JSTL	Standard TagLib	1.2	1.2	1.2
JavaMail	Envoi et réception de mail	1.4	1.4	1.4
JCA	Connecteur JEE (syst. propriétaire , ...)	1.5	1.6	1.6
JAX-WS	Api pour web-services Soap	2.0	2.2	2.2
JAX-RS	Api pour web-services REST		1.1	2.0
CDI	Injection de dépendance (inter-container)		1.0	1.1
Validation	Validation via annotations		1.0	1.1

Nouvelles API (en version 1.0) disponibles depuis JEE7 : Java Api for WebSocket , Java Api for JSON , Concurrency Utilities for JEE , Batch Applications for Java Platform

13.2. Namespaces XML pour les fichiers de configuration

Attention : La marque "SUN" a été reprise/rachetée il y a quelques années par la marque "Oracle" .

Les API les plus récentes ont maintenant des namespaces qui ont beaucoup évolués
("**http://java.sun.com**" devenu "**http://xmlns.jcp.org**")

Principales entêtes Xml des fichiers de configuration JEE :

web.xml (3.0 / JEE6)	<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee **http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd**" version="3.0">
web.xml (3.1 / JEE7)	<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee **http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd**" version="3.1">

faces-config.xml (2.0 / JEE6)	<faces-config xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd" version="2.0">
faces-config.xml (2.2 / JEE7)	<faces-config xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd" version="2.2">

beans.xml (1.0 / JEE6)	<beans xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee **http://java.sun.com/xml/ns/javaee/beans_1_0.xsd**" version="1.0">
beans.xml (1.1 / JEE7)	<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee **http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd**" version="1.1">

persistence.xml (2.0 / JEE6)	<persistence xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd" version="2.0">
persistence.xml (2.1 / JEE7)	<persistence 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" version="2.1">

II - EJB session sans état (et invocations)

1. EJB Session sans état apparent (stateless)

1.1. cycle de vie d'un EJB3 session sans état

S'il est sans état , l'EJB session ne comporte aucun champ qui doit être conservé entre 2 appels successifs vers 2 de ses méthodes "métier" (éventuellement différentes). Toutes les données nécessaires aux traitements "métier" sont passées en tant que paramètres des méthodes.

NB: Un EJB session sans état peut tout de même comporter des données internes techniques telles qu'une référence sur un "DataSource" ou un "Gestionnaire de persistance [Jpa/ejb3 entity] ".

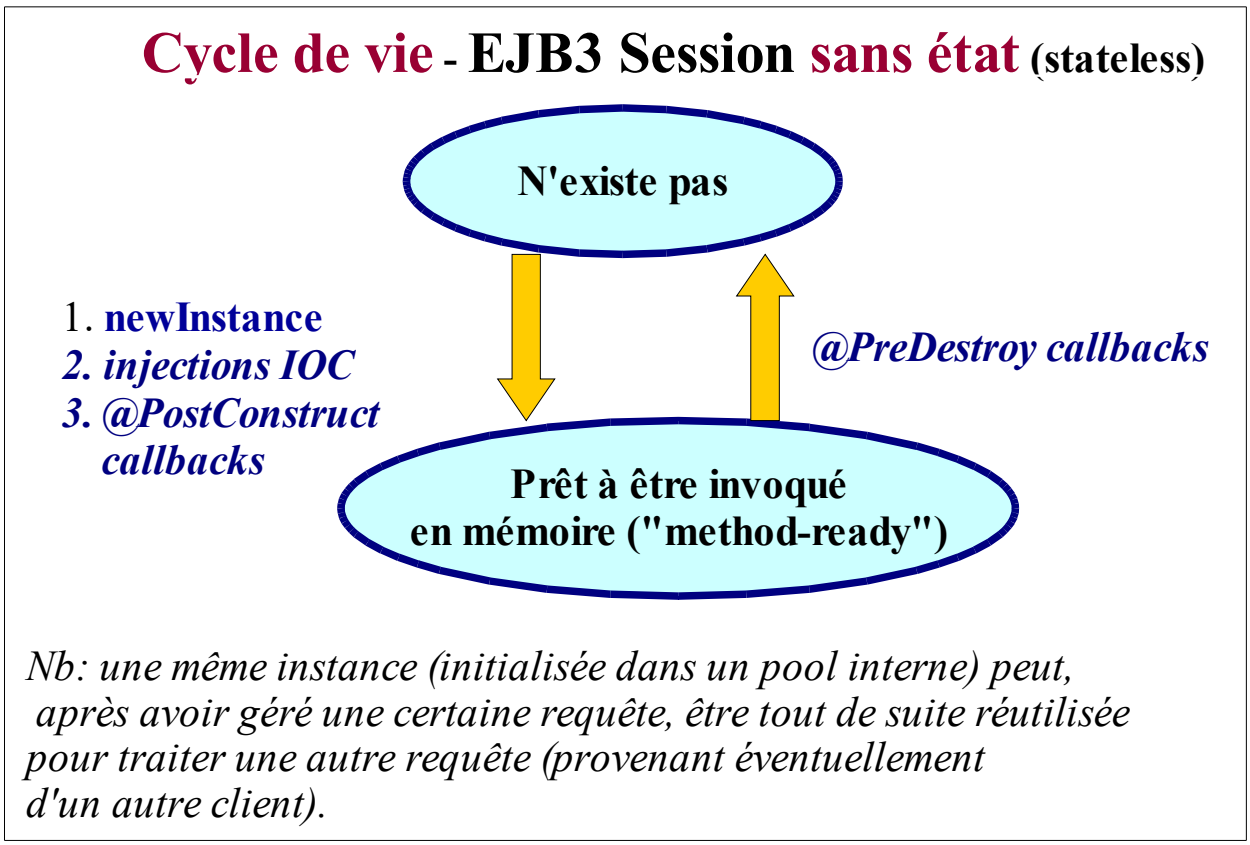
Ces références techniques internes devraient idéalement être initialisées via des injections IOC

Cette éventuelle présence d'éléments techniques (références de ressources , logique transactionnelle, contrôle d'accès selon authentification, ...) fait qu'un EJB Session sans état est :

- d'un point de vue "code des traitements" indépendant de toute instance
- d'un point de vue "état technique caché en interne" lié à une instance

D'autre part, un conteneur d'EJB maintient souvent un **pool d'EJB Session sans état**.

Une même instance **(sans état)** peut alors servir à traiter successivement plusieurs clients (lorsqu'une éventuelle transaction est entièrement terminée) :



NB: les **injections IOC** et les **méthodes "callbacks" mentionnées par @PostConstruct (après IOC) et @PreDestroy (avant destruction et sans transaction) sont facultatives** .

exemple:

Classe interne d'un l'EJB3 session sans état:

```
package myejb;
import javax.ejb.Remote;
import javax.ejb.Stateless;

@Stateless
@Remote   // ou bien @Local mais pas les deux avec une seule et même interface
public class CalculatorBean implements Calculator
{
   public int add(int x, int y)   {      return x + y;   }
   public int subtract(int x, int y)    {      return x - y;   }
   public int divide(int x, int y)   {      return x / y;   }
}
```

NB: si l'on souhaite avoir à la fois @Local et @Remote , on peut éventuellement structurer le code comme ci-après:

```
package myejb;

public interface Calculator {
      public int add(int x, int y);
      public int subtract(int x, int y);
      public int divide(int x, int y);
}
```

```
package myejb;
import javax.ejb.Local;

@Local
public interface CalculatorLocal extends Calculator {

}
```

```
package myejb;
import javax.ejb.Remote;

@Remote
public interface CalculatorRemote extends Calculator {

}
```

```
package myejb.impl;
import javax.ejb.Stateless;
import ...;
@Stateless
public class CalculatorBean implements Calculator , CalculatorLocal, CalculatorRemote {
...
@PostConstruct
   protected void myInitCallback(){     System.out.println("ejb initialised " + this);   }

   @PreDestroy
   protected void myEndCallback(){     System.out.println("end of Ejb " + this);      }
}
```

sachant que d'autres combinaisons sont encore possibles.

2. Conventions de noms JEE6 (depuis EJB 3.1)

NB1 : depuis JEE6 (et EJB3.1) , les noms JNDI doivent suivre des formats normalisés :

```
// jndi external names (from external EJB client app):
//ejb:<app-name>/<module-name>/<distinct-name>/<bean-name>!<fully-qualified-classname-of
                                                          -the-remote-interface>
```

```
Le "distinct-name" peut être vide
Exemple :
"ejb:minibank-jee6/minibank-jee6EJB//GestionComptesImpl!
tp.myapp.minibank.impl.domain.ejb.itf.GestionComptesRemote"
```

Attention : un chemin jndi de type
`"java:global/minibank-jee6/minibank-jee6EJB/GestionComptesImpl!`
`tp.myapp.minibank.impl.domain.ejb.itf.GestionComptesRemote"` **utilisé depuis un lookup déclenché par un programme externe à Jboss** mène à l'**exception** suivante :
`"javax.naming.NoInitialContextException: Need to specify class name in environment or system property" .`

==> il faut donc absolument que le chemin JNDI soit au format attendu "**ejb:....**" mentionné précédemment !

Attention :
Les conventions de noms qui suivent ont été vérifiées avec **JBoss 7.1.1**
Des tests et/ou des ajustements avec d'autres serveurs peuvent s'avérer nécessaires.

NB2 (pour Jboss7):

- La plupart des **noms JNDI** des composants "EJB" pris en charge par le serveur Jboss_AS_7 sont **affichés dans la console** de "jboss" au démarrage du serveur.
- Dans la cas où un EJB3 "stateless" comporte une annotation **@WebService** , l'**URL de sa description WSDL** s'affiche également parmi les **traces de démarrage** (dans la console ou dans un fichier de log).

3. Client distant et externe vis à vis d'un EJB3

L'accès externe et distant au serveur Jboss 7 via le registre des noms logiques JNDI a été complètement chamboulé (comparé aux anciennes versions 3, 4.2 ,et 5.1 de Jboss).
Le numéro de port (par défaut) de la partie JNDI est 4447 (et plus 1099) .

3.1. librairie nécessaire (coté client)

La librairie suivante doit être à recopiée dans le classpath de l'application cliente externe :

- **jboss-client.jar** (à récupérer dans **JBOSS7_HOME/bin/client**)

Eventuellement , junit-4.8.1.jar peut être ajouté (à des fins de tests unitaires).

3.2. Configuration et code d'accès distant à un ejb "@Remote"

Solution 1 (pour Jboss AS 7.1.1.Final) :

```
package myclient;

import javax.naming.InitialContext;
import myejb.GestionComptes;   // "business interface" de l'EJB3

public class Client
{
public static void main(String[] args) throws Exception {
java.util.Properties jndiProps = null;
jndiProps= new java.util.Properties();
jndiProps.setProperty(Context.URL_PKG_PREFIXES,
                      "org.jboss.ejb.client.naming");
try {
Context ic = new InitialContext(jndiProps);
serviceGestionComptes=(GestionComptes)
    ic.lookup("ejb:minibank-jee6/minibank-jee6EJB//GestionComptesImpl!
tp.myapp.minibank.impl.domain.ejb.itf.GestionComptesRemote");
Compte cpt = serviceGestionComptes.getCompteByNum(1L);
System.out.println("compte 1 = " + cpt.toString());
//Assert.assertTrue(cpt.getNumero()==1L);
}
catch (NamingException e){
e.printStackTrace();
}
}//end of main
}//end of class
```

NB1:Le simple fait d'avoir préciser la propriété ***URL_PKG_PREFIXES à la valeur "org.jboss.ejb.client.naming"*** couplé au chargement des bonnes librairies en mémoire provoque une ***lecture et interprétation automatique*** d'un fichier **jboss-ejb-client.properties** qui doit être présent à la racine du classpath du client.

Ce fichier **jboss-ejb-client.properties** doit ressembler au suivant :

```
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
remote.connections=default
remote.connection.default.host=localhost
remote.connection.default.port=4447
remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
```

et **META-INF/MANIFEST.MF**

```
Manifest-Version: 1.0
```

```
Main-Class: myclient.Client
```

si myclient.Client est une classe de démarrage avec une méthode main()

NB : On peut également envisager des **classes** de test "JUnit4" (sans main() et avec @Test)

Solution2 pour Jboss 7.1 et Jboss wildfly 9.2 :

```
props.put(Context.INITIAL_CONTEXT_FACTORY,
          "org.jboss.naming.remote.client.InitialContextFactory");
props.put(Context.PROVIDER_URL, "remote://localhost:4447");
          //   remote://localhost:4447 for Jboss7.1 , http-remoting://localhost:8080 for wildfly 8,9
props.put(Context.SECURITY_PRINCIPAL, "guest"); // username : "admin" , "guest" , "..."
props.put(Context.SECURITY_CREDENTIALS, "guest007"); //password : "pwd", "guest007"
        //avec utilisateur ajouté via la commande JBOSS_7_HOME/bin/add-user
        //et roles associés admin,guest,....  sur partie "ApplicationRealm" .
props.put("jboss.naming.client.ejb.context", true); //indispensable pour accès @Remote

Context jndiContext = new InitialContext(props);

String jndiName = "my-jee-app/my-jee-app-ejb-impl/ConvertisseurBean"
                  +"!tp.myapp.ejb.itf.IConvertisseur";
// sans "ejb:" et sans // pour version de jndiName sans jboss-ejb-client.properties et sans
// props.setProperty(Context.URL_PKG_PREFIXES,"org.jboss.ejb.client.naming");
ejbConvertisseur = (IConvertisseur) jndiContext.lookup(jndiName);
System.out.println("30 euros= "  + ejbConvertisseur.euroToFranc(30) + " francs");
```

Cette autre solution à le mérite d'être plus classique (proche de ce qui se fait avec d'autres serveurs tels que "glassfish" ou "websphere") et est compatible au besoins de JMS .

NB: il est quelquefois nécessaire de rendre disponible le fichier "jboss-client.jar" comme une dépendance maven :

install_jboss_client_in_maven_local.bat

```
set MVN_HOME=C:\Prog\apache-maven-3.3.9
set JBOSS_PATH=C:\Prog\jboss-as-7.1.1.Final
cd /d %~dp0
"%MVN_HOME%\bin\mvn" deploy:deploy-file
                -Dfile="%JBOSS_PATH%\bin\client\jboss-client.jar" -DpomFile=jboss-client.pom
                -DrepositoryId=local    -Durl=file:///C:\Users\didier\.m2\repository
pause
```

jboss-client.pom

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.jboss.as</groupId>
  <artifactId>jboss-client</artifactId>
  <version>7.1</version>    < !-- ou bien 9 ou 9.2 pour jboss wildfly 9.2 -->
</project>
```

D'autre part , au sein du projet "client-externe-vis-a-vis-des-ejb" , la dépendance maven vers "jboss-client" doit idéalement être placée AVANT la dépendance vers javaee-api 6 ou 7 .

4. Client Web/J2EE vis à vis d'un EJB3

Si le client d'un EJB3 est situé au sein de la partie web de l'application J2EE , les paramètres techniques JNDI n'ont alors pas besoin d'être précisés car la partie web s'exécute elle aussi au sein du serveur J2EE.
D'autre part, un accès local à l'EJB est souvent suffisant (tant qu'il s'exécute dans la même JVM).

```
InitialContext ctx = new InitialContext();
Calculator calculator = (Calculator) ctx.lookup(
                              "java:app/myJeeAppEJB/CalculatorBean!myejb.CalculatorLocal");
int res= calculator.add(1, 1);
```

Equivalent paramétré par annotations (fonctionnant au sein d'un composant pris en charge par un conteneur Web , Jsf , Ejb ou autre):

```
@EJB(mappedName="java:app/myJeeAppEJB/CalculatorBean!myejb.CalculatorLocal")
private Calculator calculator;

ou bien plus simplement (lorsqu'il n'y a qu'un seul EJB existant compatible avec l'interface) :
@EJB
private Calculator calculator;
```

4.1. Noms JNDI normalisés pour EJB 3.1 (depuis JEE6)

Les noms JNDI des EJB 3.0 dépendaient du serveur d'application hôte (Jboss, WebLogic, WebSphere, …) et n'étaient donc pas portables.

Les spécifications EJB 3.1 (JEE6) précisent (enfin) des noms JNDI portables (et de niveau global) :

Le nom JNDI d'un EJB3.1 doit être au format suivant:

```
"java:global/<app-name>/<module-name>/<bean-name>[!<fully-qualified-interface-name>]"
```

Ce nom complet global sera accompagné des deux alias suivants

```
"java:app/<module-name>/<bean-name>[!<fully-qualified-interface-name>]"
"java:module:<bean-name>[!<fully-qualified-interface-name>]"
```

utilisables depuis la même application (ou depuis le même module).

La partie !<fully-qualified-interface-name> est quelquefois facultative (lorsqu'il n'y a qu'une seule interface au niveau d'un EJB?).

NB: Tous ces noms "jndi globaux" (au niveau d'un serveur d'application) ne sont pas vus tels quels depuis l'extérieur (depuis une autre JVM/autre serveur) .

Par exemple au niveau du serveur jboss 7 , un accès externe à un ejb 3.1 doit se faire via un nom jndi du type :

```
"ejb:<app-name>/<module-name>/<distinct-name>/<bean-name>!<fully-qualified-classname-of-the-remote-interface>"
```

et lorsque distinct-name est vide comme souvent cela donne

```
"ejb:<app-name>/<module-name>//<bean-name>!<fully-qualified-classname-of-the-remote-interface>"
```

==> à priori , il vaudrait mieux utiliser des accès distants SOAP plutôt que RMI lorsque c'est possible .

4.2. Eventuelles références de ressources

NB : Bien que techniquement encore possible , cet ancien style de paramétrage/configuration (datant de J2EE) est de plus en plus considéré comme "has been" et remplacé par des injections de dépendances (avec @EJB ou @Inject) .

Une référence de ressource correspond à une **indirection** dans les ***noms JNDI*** qui servent à référencer les ressources.
C'est un peu comme un *raccourcis windows* ou un *lien symbolique unix*.
Pour être utile une référence doit rediriger vers une vraie ressource (pool de connexions, ...).

Dans certains cas , les références de ressources sont automatiquement reliées au ressources adéquates (c'est le cas de la plupart des liaisons internes [composant vers composant]) .

Dans d'autre cas, une référence qui part du code de l'application J2EE doit être mappée/associée vers une ressource globale ou externe (tel qu'un pool de connexions qui est lié/spécifique au serveur J2EE).

Les liaisons (mapping) entre les références de ressources (liées à l'application) et les véritables ressources (spécifiques au serveur) sont généralement renseignées au sein d'un fichier de configuration spécifique au serveur (ex: jboss.xml/jbossweb.xml , ...).

Certains serveurs (tel que WebSphere d'IBM) sont capables de générer automatiquement des liaisons/mappings par défaut en se basant (de façon heuristique) sur des correspondances de noms [ex: référence "java:comp/env/jdbc/Xxx" ===> ressource "jdbc/Xxx"] .
Ceci ne fonctionne malheureusement pas toujours avec JBoss qui impose des noms JNDI de type "java:Xxx"aux pools de connexions.

4.3. Référence vers ejb (depuis client web)

WEB-INF/web.xml

```
...
<ejb-local-ref>
      <ejb-ref-name>calculatorEjb</ejb-ref-name>
      <ejb-ref-type>Session</ejb-ref-type>
      <local>myejb.Calculator</local>
      <mapped-name>myJeeApp/CalculatorBean/local ou ...</mapped-name>
      <!-- NB: l'information de liaison (mapped-name) peut être redéfinie
               dans un fichier complémentaire spécifique au serveur
               (WEB-INF/jbossweb.xml , ....) -->
  </ejb-local-ref> ...
```

avec

```
private void initEjb(){
      try {
            InitialContext ctx = new InitialContext();
            this.calculator = (Calculator)
                    ctx.lookup("java:comp/env/" + "calculatorEjb");
            } catch (NamingException e) {
                  e.printStackTrace();
```

```
            }
        }
```

ou bien

```
@EJB(name="calculatorEjb")
private Calculator calculator=null;
```

5. Liens entre différents EJB

Un EJB de type "MDB" peut utiliser un EJB de type "session/stateless"
Un EJB de type "session/stateless" (de niveau service métier) peut éventuellement utiliser un autre EJB de type "session/stateless" (jouant le rôle d'un DAO).

Il ne faut surtout pas écrire directement "refEjb = new YYYBean();" car l'EJB instancié serait mal initialisé (pas pris en charge par le conteneur d'EJB3).

La bonne solution consiste à paramétrer une injection de dépendance vers un autre EJB:

```
...
public class XXXBean implements XXX{

@EJB(name="YYYBean")
private YYY  yyyEjb;
... }
```

NB : S'il existe (comme souvent) une seule implémentation (ex : YYYBean) de l'interface YYY alors l'injection peut être effectuée avec une annotation sans paramètre : **@EJB()**

III - EJB vu/invoqué comme un service web

1. EJB3 session sans état vu comme un service WEB

1.1. Paramétrage de l'aspect "service web" d'un EJB Session

interface du service web

```
package myejb;

import javax.jws.WebParam;
import javax.jws.WebService;
import data.Prix;

/* * NB: @WebService est indispensable
 * et sans @WebParam les noms des paramètres deviennent "arg0" , "arg1" , ... dans le WDSL */

@WebService ( )
public interface Convertisseur {
        public double euroToFranc(@WebParam(name="ve") double ve);
        public double francToEuro(@WebParam(name="vf")double vf);
        public Prix getConvertPrice(@WebParam(name="p")Prix p);
}
```

classe de l'EJB vu comme un service web:

```
package myejb;

import javax.ejb.Stateless;
import javax.jws.WebService;
import data.Prix;
/* WSDL_URL ===> (pour Jboss 5.1)
                http://localhost:8080/myJeeApp-myJeeAppEJB/ConvertisseurBean?wsdl */
@Stateless
@Remote
@WebService(endpointInterface="myejb.Convertisseur")
public class ConvertisseurBean implements Convertisseur {
        public double euroToFranc(double ve) { return ve * 6.5957; }
        public double francToEuro(double vf) { return vf / 6.5957; }
        public Prix getConvertPrice(Prix p) { Prix res=null;
            if(p.getMonnaie().equals("Euro")) res= new Prix(p.getMontant() * 6.5957,"Franc");
             else if(p.getMonnaie().equals("Franc"))
                                                res= new Prix(p.getMontant() / 6.5957,"Euro");
            return res;
        }
}
```

NB: lors du déploiement dans le serveur d'application , les annotations (@WebService , ...) de l'api standard JAX-WS seront interprétées/analysées et le serveur mettra alors en oeuvre tous les éléments nécessaires (point d'accès SOAP , description WSDL , ...) .

NB: Le programme utilitaire "soap-ui" (à télécharger) offre une bonne interface graphique pour tester un service web (sans programmer).

NB: Si Jboss est lancé depuis eclipse (en mode développement/test), il faut alors configurer le serveur Jboss de la façon suivante:

Dans Server/Open/Pen Launch config:

-b 0.0.0.0 (prog arg)

et (VM args):

-Djava.endorsed.dirs=C:\...\jboss-5.1.0.GA\lib\endorsed
en plus de -Dprogram.name=run.bat -Xms128m -Xmx512m -XX:MaxPermSize=256m
(en une seule ligne avec les différentes options séparées par des " ")

1.2. Client externe JAX-WS (java6)

Mode opératoire:

1. **générer (à partir du fichier WSDL)** toutes les classes nécessaires au "***proxy JAX-WS***"
2. écrire le code client utilisant les classes générées.

Génération d'un proxy d'appel avec wsimport du jdk 1.6 :

lancer_wsimport.bat

```
set JAVA_HOME=C:\Prog\java\jdk\jdk1.6.0_19
set WSDL_URL=http://localhost:8080/myJeeApp-myJeeAppEJB/ConvertisseurBean?wsdl
set DEST_DIR=D:\tp\JAVA_EE\back\ejb3\wksp_ejb3\myJeeAppClient\appClientModule
"%JAVA_HOME%/bin/wsimport"  -keep -d %DEST_DIR%  %WSDL_URL%
```

sans l'option -keep ==> proxy au format compilé seulement

avec l'option ***-keep*** ==> code source du proxy également

l'option **-d** permet d'indiquer le répertoire *destination* (ou le proxy sera généré).

D'autres options existent (à lister via ***-help***)

NB: wsconsume (spécifique Jboss) peut éventuellement être utilisé à la place de wsimport .

Exemple de code "client" invoquant un service web:

```
package myclient;


import javax.xml.ws.BindingProvider;
import xxx.data.Prix;
import xxx.serv.ConvertisseurBeanService;
```

```
public class ClientApp{

public static void main (String[] args) {
  try {
     ConvertisseurBeanService service = new ConvertisseurBeanService();

    xxx.serv.Convertisseur conv = (xxx.serv.Convertisseur)
            service.getConvertisseurBeanPort();
    /*
    javax.xml.ws.BindingProvider bp = (javax.xml.ws.BindingProvider) conv;
    Map<String,Object> context = bp.getRequestContext();
    context.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
                          "http://127.0.0.1:8080/myJeeApp-myJeeAppEJB/ConvertisseurBean");
    */
    /* context.put(BindingProvider.USERNAME_PROPERTY,"userNameSiBasicHttpAuth");
       context.put(BindingProvider.PASSWORD_PROPERTY, "pwdSiBasicHttpAuth"); */

    System.out.println("100 F = " + conv.francToEuro(100) + " E");
    System.out.println("15 E = " + conv.euroToFranc(15) + " F");

     } catch (Exception ex) {
       ex.printStackTrace();
     }
  }

}
```

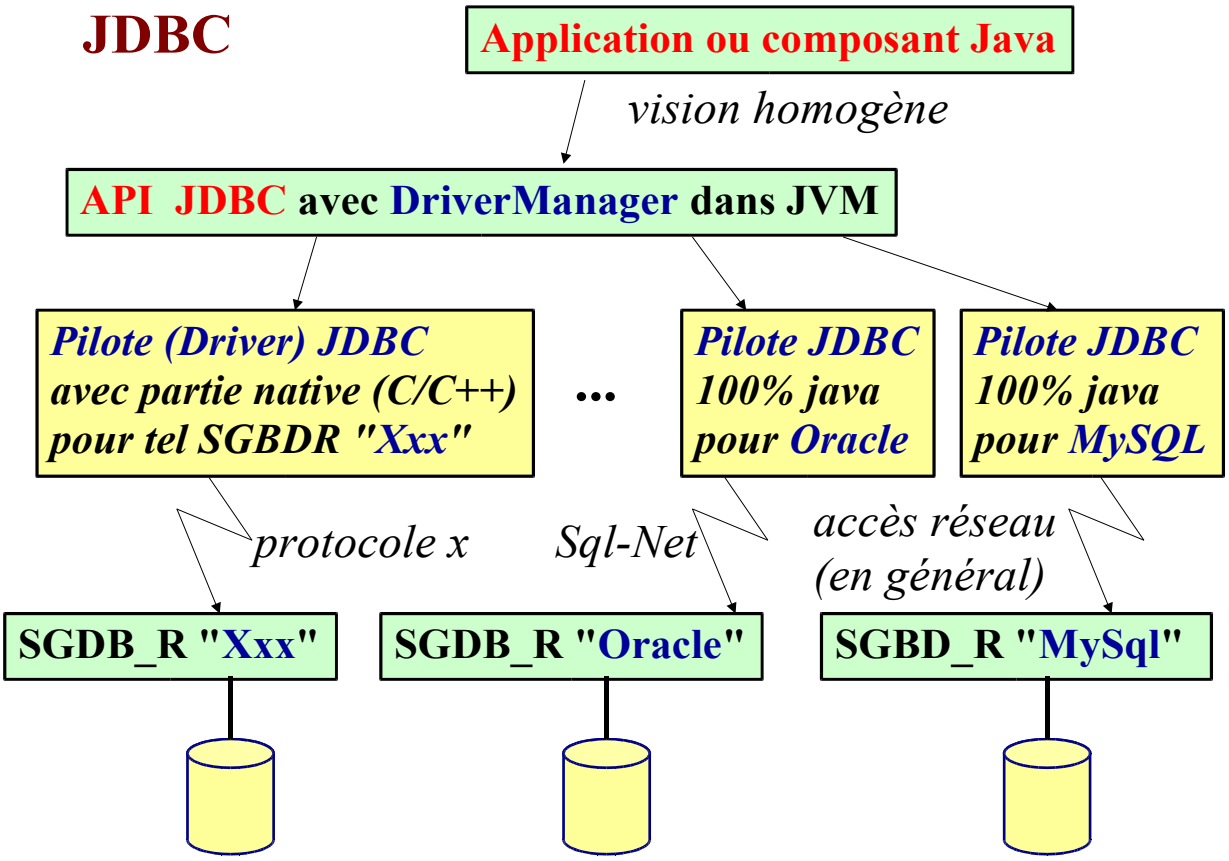
NB: Les EJB3 (et les serveurs d'application "JEE5 , JEE6 , JEE7") utilisent en interne les API JAX-WS et JAXB2 .

==> Etudier si besoin l'api JAX-WS pour approfondir le sujet.

IV - Source de données JDBC (et EJB)

1. Sources de données JDBC

1.1. Api JDBC (Java DataBase Connectivity)



1.2. Pool de connexions et DataSource

Pool de connexions vers SGBDR

Rôles (utilités) des pools de connexions:

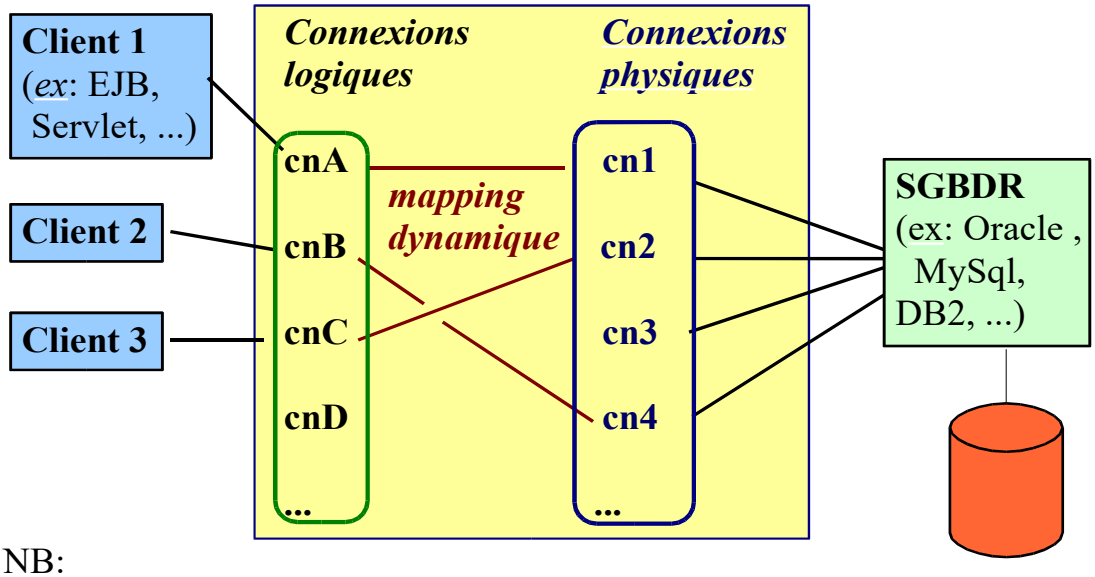
Recycler et ***partager*** *(par différentes attributions successives)* **un ensemble de connexions physiques vers un certain SGBDR**.

Ceci permet d'éviter les 2 écueils suivants:

- *Ouvrir, fermer et ré-ouvrir , ... des connexions vers le SGBDR* (opérations longues répétées ➔ mauvaises performances).

- *Utiliser simultanément une même connexion pour effectuer de multiples traitements* ➔ mauvaise gestion des concurrences d'accès et des transactions (joyeux mélanges)

Pool de connexions



NB:

Dès d'un client ferme une connexion logique , la connexion physique associée est considérée comme libre et peut alors être recyclée de façon à ce qu'un autre client puisse obtenir une nouvelle connexion logique.

Remarque: Etant donné qu'une connexion libérée (via close) par un composant n'est pas vraiment fermée mais peut être tout de suite réutilisée par un autre composant, chaque traitement (à l'intérieur d'une méthode d'un composant) doit:

- demander une connexion disponible dans le pool
- l'utiliser brièvement
- rapidement la libérer

Vue du pool par le client java - DataSource

Un client java voit un pool de connexions JDBC comme un objet de type *javax.sql.**DataSource***.
L'accès à cette source de données découle d'une **recherche JNDI** à partir d'un nom convenu (à paramétrer):

```
InitialContext ic = new InitialContext();
String dsName="java:comp/env/jdbc/dsBaseX"
DataSource ds = (DataSource) ic.lookup(dsName);
```

L'objet *DataSource* permet alors de **récupérer de nouvelles connexions logiques**:

```
Connection cn =  ds.getConnection();

// ... utilisation classique d'une connexion JDBC  ...

cn.close(); // fermeture de la connexion logique
```

2. Architecture JCA (Connecteurs)

JCA (**J**ava **C**onnector **A**rchitecture) est une API assez générique permettant à un composant déployé dans un serveur d'application J2EE de communiquer avec des **s**ystèmes d'**i**nformations très divers d' une **e**ntreprise (**E.I.S.)** :

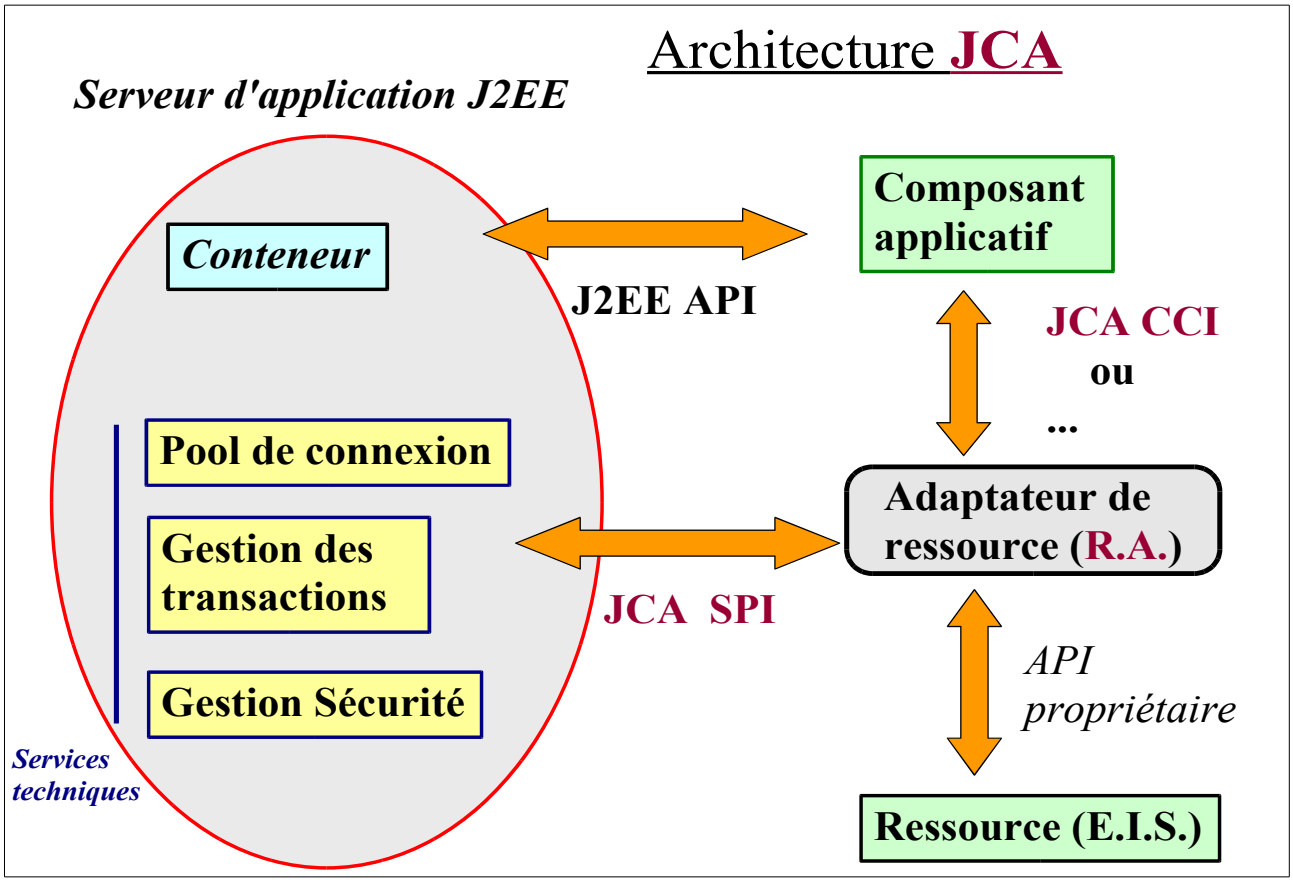
- **SGBDR** (Oracle, Sybase, DB2, SQL-server, ...) – via JDBC.
- **Moniteur transactionnel** (Tuxedo, CICS, ...).
- **Système propriétaire** (SAP, ..., ERP divers, ...).

Le principal objectif de **JCA** consiste à bien **découpler les différents aspects suivants**:

- **Services techniques géré par le serveur d'application** (contrôle supervisé des **transactions**, gestion de la **sécurité**, gestion des **pools de connexions**, ...).
- Appels effectués par le composant java (ex: EJB) ==> Api particulière (JDBC, ...) ou bien interface "client" commune (générique) **CCI** de JCA.
- Fonctionnalités propres au système d'information mis en jeu.

Les spécifications de **JCA** permettent (via des interfaces contractuelles) de bien délimiter les rôles des différents intervenants (Composant, EIS, Serveur d'application) devant bien collaborer.

- L'interface JCA **SPI** (**S**ervice **P**rovider **I**nterface) permet de faire en sorte que le serveur d'application puisse contrôler certains services techniques de l'EIS via le connecteur.
- L'interface JCA **CCI** (**C**ommon **C**lient **I**nterface) permet à un composant applicatif de communiquer de façon relativement standard avec l'adaptateur de ressources.



3. Accès à un DataSource JDBC depuis un Ejb session

3.1. Récupération d'une source de données via recherche JNDI

```
@Resource
private SessionContext ctx;
```

ou bien

```
InitialContext ctx = new InitialContext();
```

puis

```
DataSource ds = (DataSource) ctx.lookup("java:/MyDS");
```

3.2. injection directe d'une dépendance vers une ressource

```
@Stateless
public class MySessionBean implements MySession {
...
@Resource(mappedName="java:/myDS")
public DataSource customerDS;
...
public void myMethod1(String myString){
        try {    Connection cn = customerDS.getConnection();     ...}  catch (Exception ex){...  }
    } ....}
```

4. éventuelle utilisation coté web (DataSource)

WEB-INF/web.xml

```
... <!-- JDBC DataSources (java:comp/env/jdbc)
        ENC lookup("java:comp/env/" +"jdbc/TestEjb3DbDataSource") -->
    <resource-ref>
        <description>TestEjb3DB DS</description>
        <res-ref-name>jdbc/TestEjb3DbDataSource</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
        <!-- <mapped-name>java:/TestEjb3DbDataSource</mapped-name>
             (info dans jboss-web.xml) -->
    </resource-ref>...
```

NB: Le nom JNDI d'une ressource du serveur peut éventuellement être renseignée dans un fichier spécifique au serveur : jboss.xml (ou jboss-web.xml) à la place d'utiliser la balise <mapped-name>.

Exemple (pour un "DataSource" de Jboss que l'on souhaite utiliser coté web):

WEB-INF/jboss-web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>
   <resource-ref>
        <res-ref-name>jdbc/TestEjb3DbDataSource</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <jndi-name>java:/TestEjb3DbDataSource</jndi-name>
    </resource-ref>
</jboss-web>
```

5. Configuration d'une source de données (pool)

La configuration d'une source de données (pool de connexions JDBC) s'effectue à partir d'un fichier de configuration spécifique à un serveur d'application .
Cette configuration n'est pas la même au sein de Tomcat , Jboss ou WebSphere.

Dans le cas des anciennes versions des serveurs JBoss (3, 4 , 5) , une source de données JDBC se configurait via un fichier "**xxx-ds.xml**" qu'il fallait placer dans le répertoire **JBoss/server/default/deploy** .

Au sein des versions récentes de Jboss (7.x , EAP 6.x, wildfly) , la configuration d'un datasource s'effectue maintenant dans le fichier JBOSS7_HOME/standalone/configuration/**standalone.xml .**

5.1. Configuration d'un "dataSource" jdbc (jboss7)

Une source de données JDBC se configure dans Jboss 7 comme une **nouvelle partie** (à déclarer) dans le **sous système "urn:jboss:domain:datasources:1.0"** du fichier JBOSS7_HOME/standalone/configuration/**standalone.xml** et comme un **nouveau module (jboss7/osgi)** comportant **le driver JDBC** *(ex: mysqlconnector...jar)* **.**

Les blocs de configuration à ajouter à **standalone.xml** sont les suivants :

```
...
<subsystem xmlns="urn:jboss:domain:datasources:1.0">
        <datasources>
          <datasource ...>  ...  </datasource>
          <datasource jndi-name="java:/MinibankDS" pool-name="MinibankDS-Pool"
                      enabled="true" jta="true" use-java-context="true" use-ccm="true">
            <connection-url>jdbc:mysql://localhost:3306/minibank_db_ex1</connection-url>
            <driver>mysql</driver>
            <transaction-isolation>TRANSACTION_READ_COMMITTED</transaction-isolation>
            <pool>
              <prefill>true</prefill> <use-strict-min>false</use-strict-min>
              <flush-strategy>FailingConnectionOnly</flush-strategy>
            </pool>
            <security>
              <user-name>root</user-name>   <password>root</password>
            </security>
          </datasource>
          <drivers>
            <driver name="h2"...>..... </driver>
            <driver name="mysql" module="com.mysql">
              <driver-class>com.mysql.jdbc.Driver</driver-class>
              <xa-datasource-class>
                 com.mysql.jdbc.jdbc2.optional.MysqlXADataSource
              </xa-datasource-class>
            </driver>
          </drivers>
        </datasources>      </subsystem>...
```

5.2. Configuration générale d'un module osgi pour jboss7 (exemples : "dozer" et/ou "mysql")

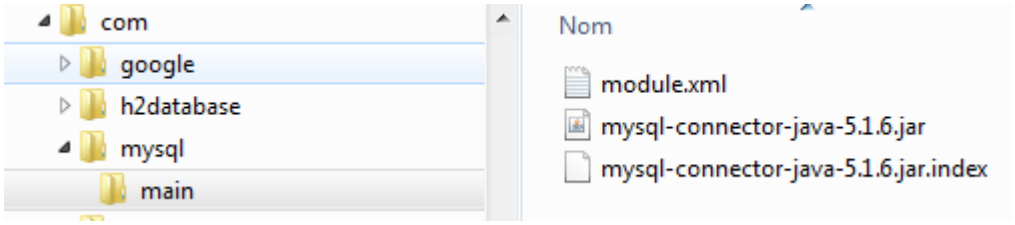
Un nouveau **module (jboss7/osgi)** se configure en créant de **nouveaux répertoires et fichiers** dans la branche *JBOSS7_HOME/**modules***.

Par exemple le nouveau module "*org.dozer*" peut se configurer de la façon suivante :

1) création du nouveau répertoire "*dozer*" dans *modules/org* .
2) création du sous répertoire "**main**" dans *modules/org/dozer*.
3) création (par copie à adapter) du fichier de configuration main/**module.xml**
4) **placer** (un ou plusieurs) **fichier(s) "xyz.jar" dans le répertoire "main"**
   (ex: *dozer-5.3.1.jar*)
5) éditer le contenu de **main/module.xml** pour préciser les **dépendances :**

```
<?xml version="1.0" encoding="UTF-8"?>
<!--  org.dozer module  -->
<module xmlns="urn:jboss:module:1.1" name="org.dozer">
  <resources>
        <resource-root path="dozer-5.3.1.jar"/>  <!-- Insert other resources here -->
  </resources>
  <dependencies>
    <module name="javax.api"/>    <module name="org.apache.commons.beanutils"/>
    <module name="org.apache.commons.lang"/>
    <module name="org.slf4j"/>    <module name="javax.management.j2ee.api"/>
  </dependencies>
</module>
```

Exemple de configuration du module "**com.mysql**" (avec diver jdbc) :



avec modules/com/mysql/main/**module.xml** contenant :

```
<module xmlns="urn:jboss:module:1.1" name="com.mysql">
  <resources>
        <resource-root path="mysql-connector-java-5.1.6.jar"/>
  </resources>
  <dependencies>
    <module name="javax.api"/>    <module name="javax.transaction.api"/>
  </dependencies>
</module>
```

NB : le fichier ….jar.index sera créé automatiquement.
urn:jboss:module:1.1 pour **Jboss 7.1** et **urn:jboss:module:1.3** pour **Jboss wildfly 9.2**

V - Ejb "Entity" & JPA (présentation)

1. Problématique "O.R.M."

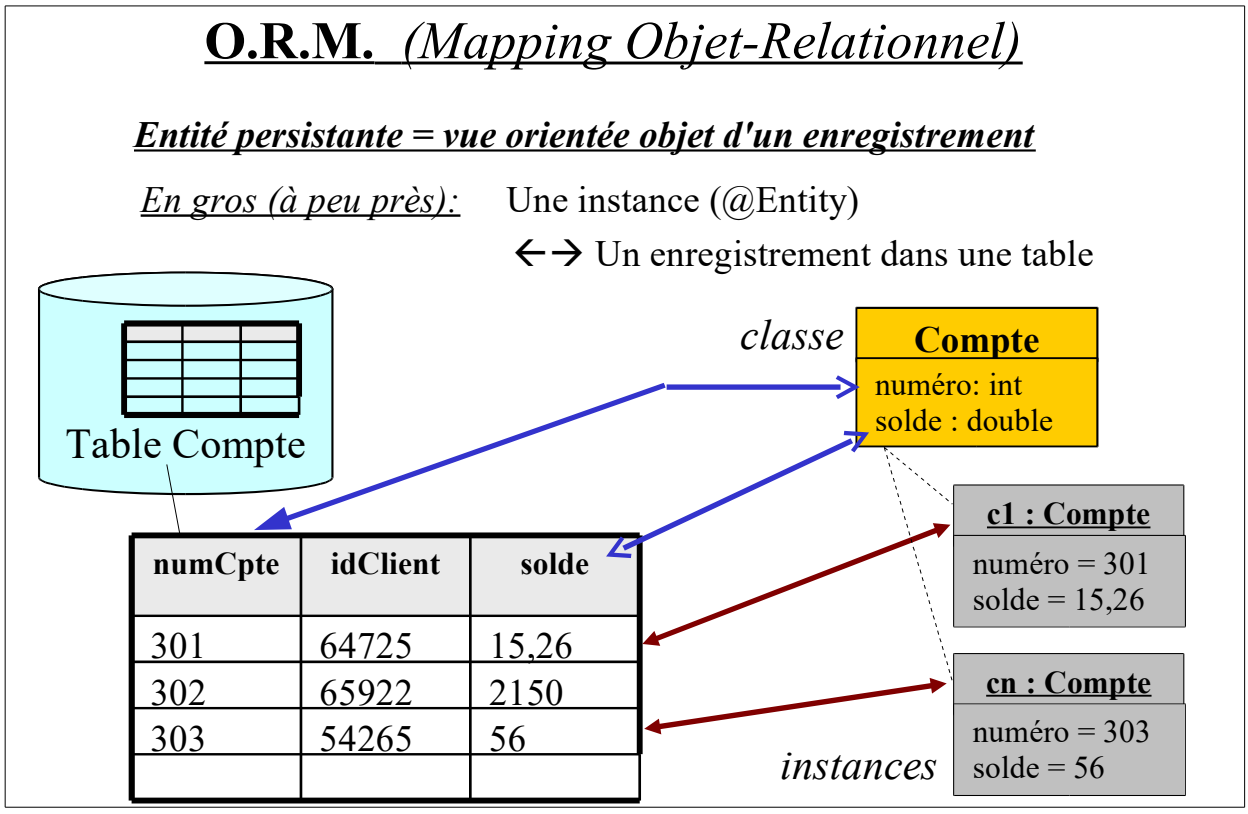
1.1. Objectif & contraintes:

L'objectif principal d'une technologie de **mapping objet/Relationnel** est d'établir une **correspondance relativement transparente et efficace** entre :

un **ensemble d'objets en mémoire**

et

un **ensemble d'enregistrements d'une base relationnelle.**



Une telle technologie doit permettre à **un programme orienté objet de ne voir que des objets** dont certains sont des **objets persistants**. ***Le code SQL est en très grande partie caché*** car les objets persistants sont automatiquement pris en charge par la technologie "O.R.M."

Autrement dit , **le code SQL n'est plus (ou très peu) dans le code "java" mais est généré automatiquement à partir d'une configuration de mapping** (*fichiers XML* , annotations, ...) .

Contraintes : pour être **exploitable** , une **technologie "O.R.M."** se doit d'être :

- **simple** (à configurer et à utiliser) et **intuitive**
- **portable** (possibilité de l'intégrer facilement dans différents serveurs d'application)
- **efficace** (bonnes performances, fiable , ...)
- capable de gérer les **transactions** ou bien de participer à une transaction déjà initiée
- ...

1.2. Eléments techniques devant être bien gérés

Au delà des simples associations élémentaires du type
"1 enregistrement d'une table <---> 1 instance d'une classe" ,
une **technologie "O.R.M." sophistiquée** doit savoir gérer certaines **correspondances évoluées** :

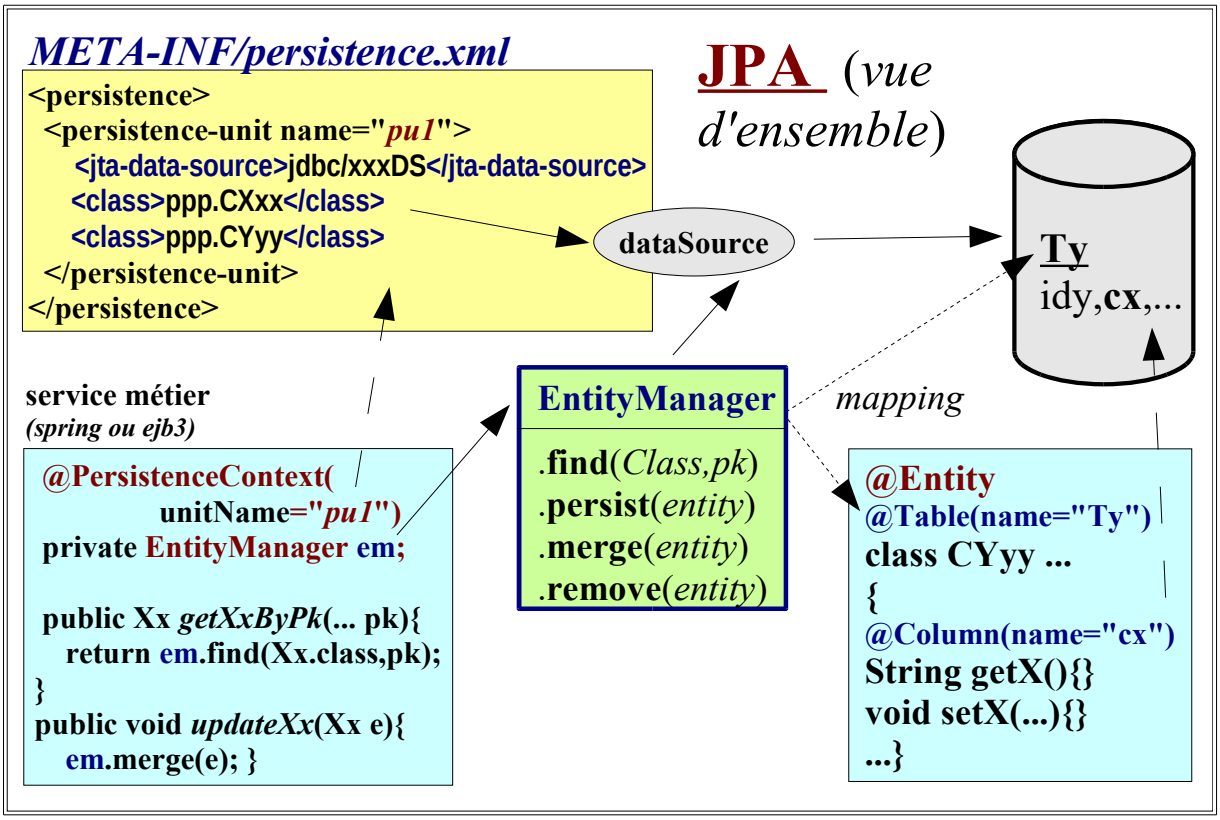
- **Associations "1-1" , "1-n" , "n-n"** (jointures entre tables <---> relations entres objets)
- **Objets "valeur"** (sous parties d'enregistrements , tables secondaires , ..)
- **Généralisation/Héritage/Polymorphisme** (<---> schéma relationnel ?)
- ...

Une technologie "O.R.M." doit sur ces différents points être **flexible et efficace** .

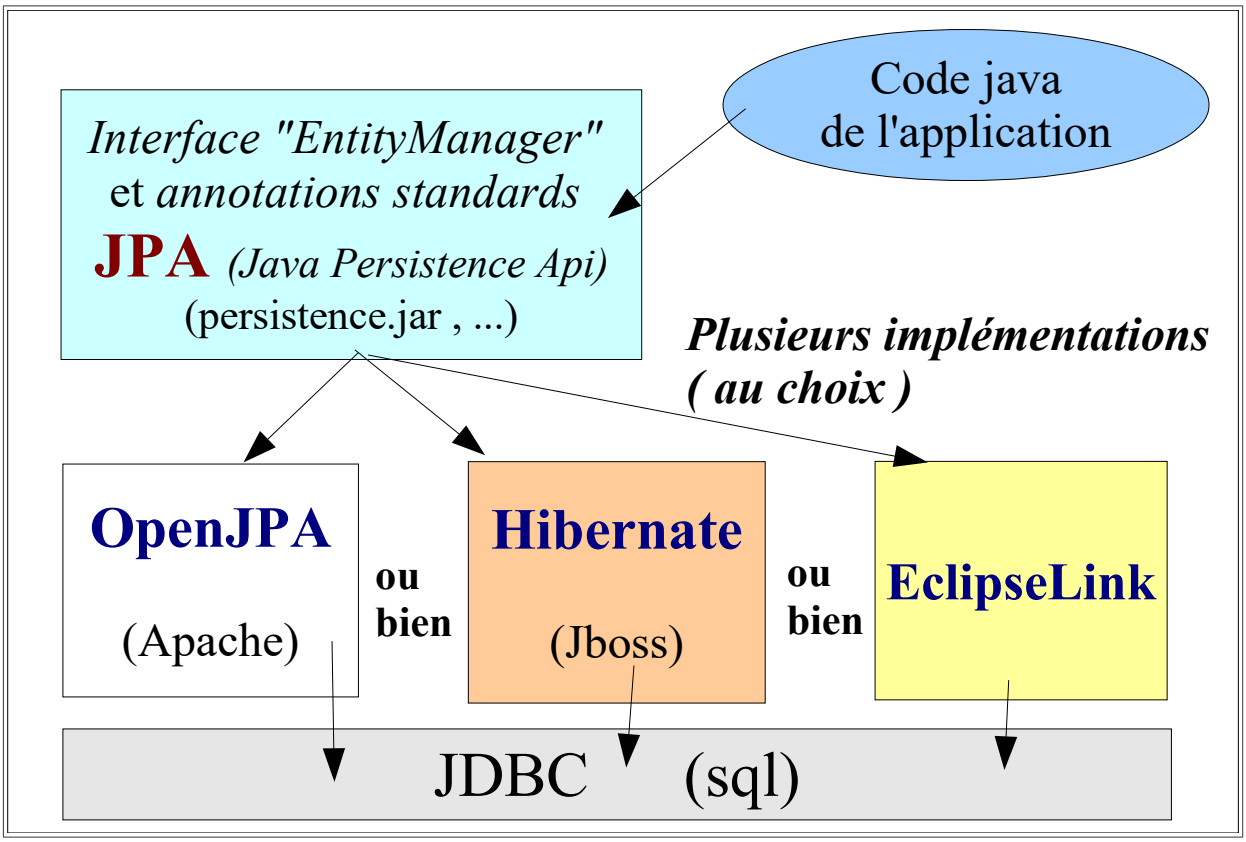
2. JPA (Java Persistance Api)

JPA (Java Persistence Api) et @Entity

- **API O.R.M. officielle** de **Java EE**
- **Configuration** basée sur des **annotations** *(avec configuration xml possible : orm.xml).*
- Utilisable dans les contextes suivants:
  - dans **module d'EJB3**
  - dans **module Spring**
  - de façon **autonome** (java sans serveur)
- Plusieurs **implémentations** internes disponibles (***OpenJPA*** , ***Hibernate*** depuis version 3.2 , ...)



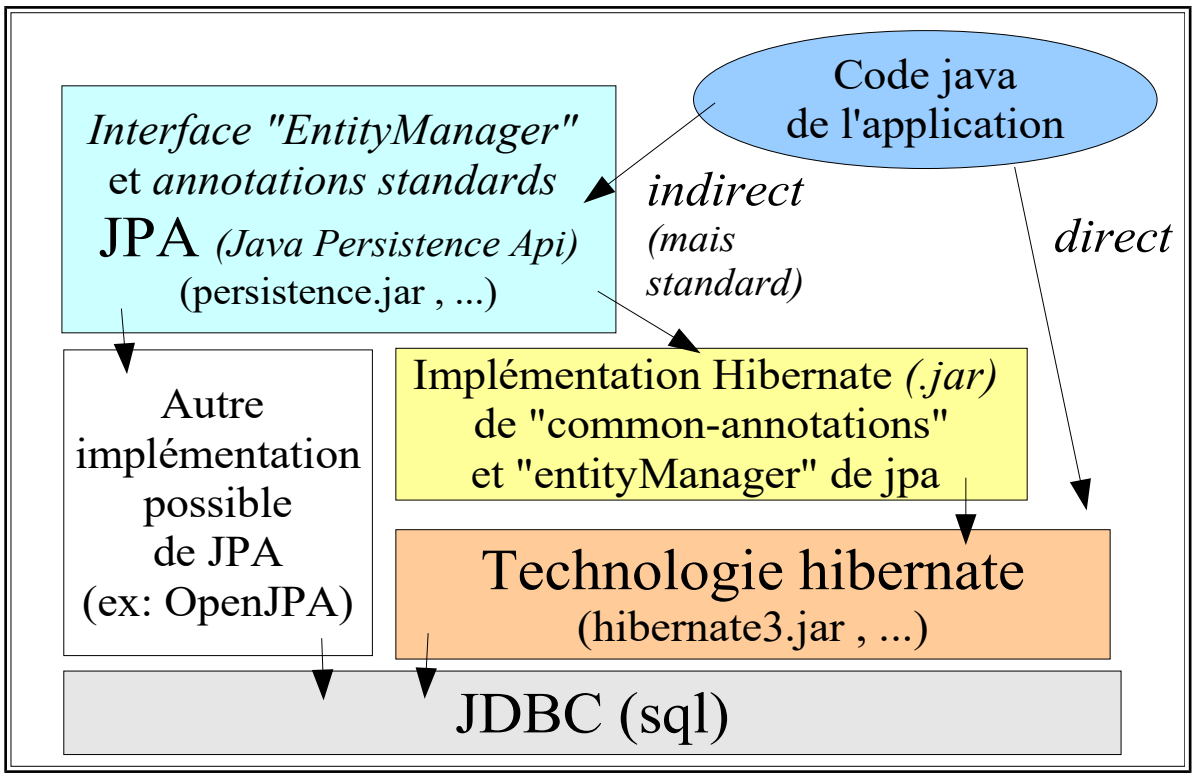
2.1. Plusieurs implémentations disponibles pour JPA



Selon le contexte, l'implémentation de JPA sera libre ou bien quasi imposée par le serveur d'applications (ex : JBoss Application Serveur utilise en interne Hibernate pour implémenter JPA).

Hibernate est quelquefois utilisé en tant qu'implémentation interne de JPA.

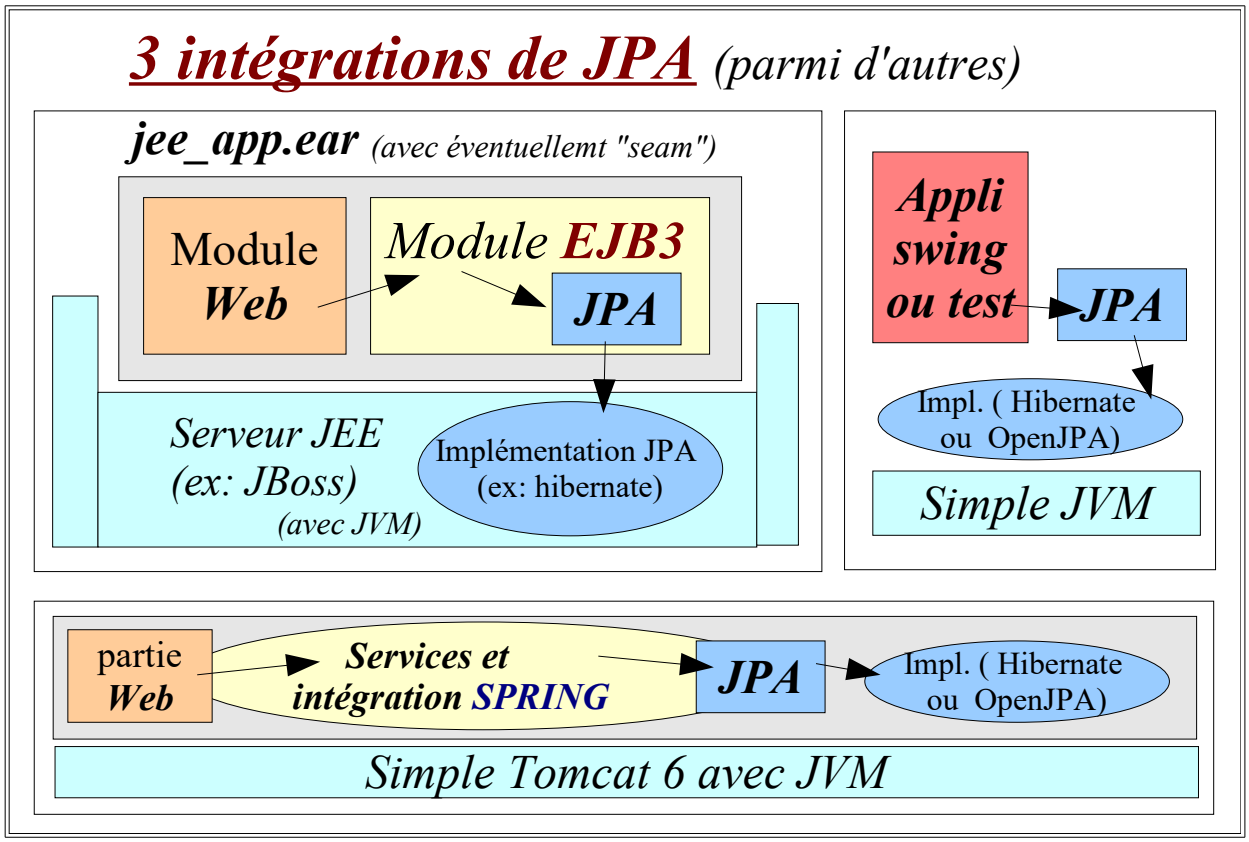
C'est le cas au sein du serveur Jboss :



VI - JPA : architecture & configuration

1. Présentation de JPA (Java Persistence Api)

L'api JPA peut être utilisée de façon très variable (dans EJB3 , dans Spring ou seul)



Une intégration "EJB3" s'appuie essentiellement sur ce qui est offert par le serveur d'applications (Jboss, WebSphere, ...). Par exemple JBoss 4.2.3 ou 5.1 comporte déjà en lui toutes les librairies (.jar) correspondant à l'API JPA (et son implémentation basée sur Hibernate).

Une intégration "Spring" s'appuie par contre généralement sur des librairies choisies (Hibernate ou OpenJPA ou ....) et qui sont quelquefois livrées et déployées avec l'application (dans WEB-INF/lib par exemple) .

Une classe de Test "Junit" exécutée avec une simple JVM peut très bien utiliser directement l'API JPA (avec ou sans Spring) en s'appuyant sur une des implémentations disponibles (Hibernate ou OpenJPA ou ...) .

2. Unité de Persistance (configuration+packaging)

Une **unité de persistance** (***Persistent Unit***) est une sorte de **module (ou sous module) applicatif** ou une sorte d'**unité de configuration** qui permet de rassembler/packager les éléments complémentaires suivants:

- ***gestionnaire de persistance*** (EntityManagerFactory et EntityManager)
- ***contexte de persistance (ensemble bien délimité d'objets persistants (@Entity) à prendre en charge).***
- Métadonnées pour le mapping ORM (**configuration** sous formes d'annotations et/ou de fichier Xml)

Concrètement, une **unité de persistance** possède un **nom unique** (permettant de la référencer) et ***se configure au sein d'un fichier de configuration standard*** (**META-INF/persistence.xml**) .

META-INF/persistence.xml à placer dans une des archives suivantes:

- ***module d'EJB*** (ejb-jar file)
  ou
- sous module d'un module WEB (.jar dans WEB-INF/lib d'un .war)
  ou
- ***module public/partagé dédié à la persistance JPA*** (.jar à la racine de l'ear ou dans le répertoire lib d'un EAR) ou Module **Spring** ou ....

META-INF/persistence.xml peut comporter **un ou plusieurs "PersistentUnit"** identifié(s) par un (des) nom(s) unique(s).
NB:

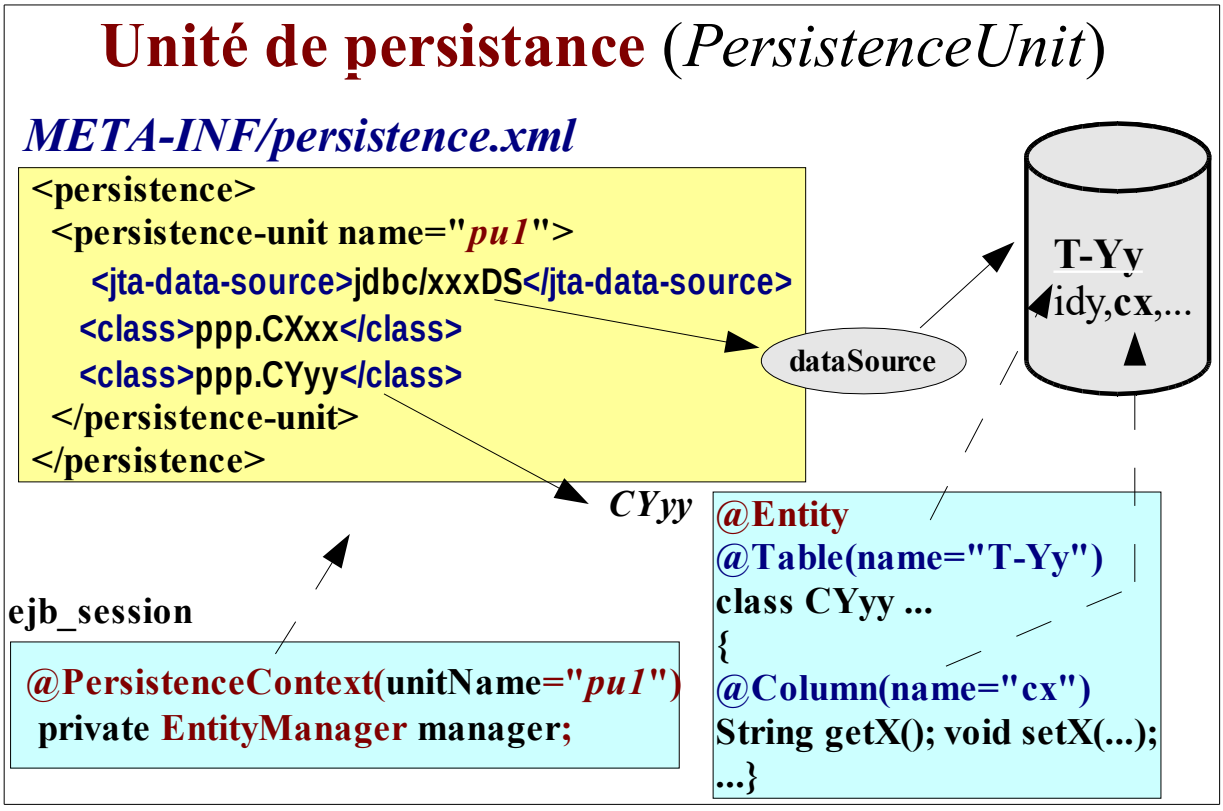
- Lorsqu'une unité de persistance est incorporée dans un module d'EJB , celle ci est alors considérée comme "privée" et ne peut pas être utilisée par d'autres modules. D'autre part, ci celle-ci est unique, il n'est pas indispensable de préciser son nom.

- Lorsqu'une unité de persistance est packagée dans un module public de niveau global ('.jar" dans l'EAR) ***[ce qui est assez rare]*** , il faut alors explicitement déclarer son existence au sein du fichier META-INF/**application.xml** :

```
<?xml version="1.0" encoding="UTF-8"?>
<application ....>
     <display-name>test_ejb3</display-name>
     <module>
          <!-- <persistence>test_ejb3_jpa.jar</persistence> -->
          <ejb>test_ejb3_jpa.jar</ejb>
     </module>
     <module>
          <ejb>test_ejb3_ejb.jar</ejb>
     </module> ...
</application>
```

<persistence> ... </persistence> est prévu pour **JEE5** (ex: Jboss5)
<ejb> .... </ejb> est temporairement accepté par J2EE1.4 / Jboss4

3. Unité de Persistance (META-INF/persistence.xml)



META-INF/persistence.xml (exemple)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"  version="1.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" >
    <persistence-unit    name="OrderManagement">
       <description>....</description>
          <jta-data-source>java:/MyOrderDS</jta-data-source>
          <!-- <mapping-file>orm.xml</mapping-file> -->
          <class>com.widgets.Order</class>
          <class>com.widgets.Customer</class>
          <properties>
             <!--  <property name="hibernate.hbm2ddl.auto" value="create-drop"/> -->
              <property  name="hibernate.dialect"  value="org.hibernate.dialect.MySQLDialect"/>
               <property  name="hibernate.show_sql"  value="true"/>
           </properties>
        </persistence-unit>
</persistence>
```

<persistence-unit> comporte un attribut optionnel "***transaction-type***" dont les valeurs possibles sont "***JTA***" ou "***RESOURCE_LOCAL***".
La valeur par défaut est "JTA" dans un env. EE (ex : EJB3) et "RESOURCE_LOCAL" dans un env. SE.

La valeur de **<jta-data-source>** ou **<non-jta-data-source>** correspond au ***nom JNDI global de la source de données SQL*** (pool de connexions permettant d'atteindre une base de données précise).
[Rappel : le serveur Jboss A.S. ajoute le préfixe "java:/" sur les noms JNDI des "dataSources"]

L'ensemble des classes (avec annotations @Entity) devant être prises en compte et prises en charge par le contexte de persistance se définit via une (ou plusieurs et complémentaires) indication(s) suivante(s):

- **Une liste explicite de classes** (balises ***<class>***)
- un ou plusieurs **fichier(s) xml de mapping "objet/relationnel"** (balises ***<mapping-file>***)
- ... .

NB: Dans le cas particulier d'un module EJB3 , les classes comportant des annotations @Entity seront automatiquement détectées et il n'est donc pas absolument nécessaire de les déclarées dans le fichier META_INF/persistence.xml . Par contre avec Spring , les <class>p.EntityClass</class> doivent être explicitées.

NB: un fichier xml de type **orm.xml** constitue soit une alternative vis à vis d'une configuration basée sur des annotations Java5 , soit un mécanisme de redéfinition.

Dans le cas où JPA est implémenté via la sous couche Hibernate , on peut éventuellement fixer la propriété ***hibernate.hbm2ddl.auto*** à l'une des valeurs suivantes :
* **create-drop** [créer les tables au démarrage de l'application et les supprime à l'arrêt]
* **update** [mise à jour des tables sans suppression]
* **none (ou le tout en commentaire)**

===> Les "create table" sont générés à partir des informations @Table , @Column , ...
===> Cette fonctionnalité peut être pratique en mode développement mais est très dangereuse en mode production !!!

Variante de META-INF/persistence.xml pour intégration dans Spring :

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0" >
<persistence-unit name="myPersistenceUnit"
                  transaction-type="RESOURCE_LOCAL">
 <provider>org.hibernate.ejb.HibernatePersistence</provider>
      <class>entity.persistance.jpa.Compte</class>
      <class>entity.persistance.jpa.XxxYyy</class>
 <properties>
    <property name="hibernate.dialect"
        value="org.hibernate.dialect.MySQLDialect" />
     <!--  <property name="hibernate.hbm2ddl.auto" value="update" /> -->
</properties>
</persistence-unit>
</persistence>
```

NB : **<provider>**.....HiernatePersistence**</provider>** permet de préciser que la technologie Hibernate est choisie pour implémenter JPA .

4. Configuration du mapping JPA via annotations

Le mapping de JPA se configure génralement en insérant quelques annotations JAVA dans le code JAVA des entités.

4.1. Exemple d'entité persitante (@Entity)

```
@Entity
@Table(name = "T_CUSTOMER")
public class Customer implements Serializable {

@Id
private Long number;

@Column(name = "customer_name")
private String name;
private Address address;

@OneToMany(...)
private List<Order> orders = new ArrayList<Order>();

@ManyToMany(...)
private List<PhoneNumber> phones = new ArrayList<PhoneNumber>();

public Customer() {} // No-arg constructor

public Long getNumber() {return number;}  // pk
public void setNumber(Long number) {this.number = number;}

public String getName() {return name;}
public void setName(String name) {this.name = name;}
public Address getAddress() {return address;}
public void setAddress(Address address) {this.address = address;}

public List<Order> getOrders() {return orders;}
public void setOrders(List<Order> orders) {
                        this.orders = orders;}
public List<PhoneNumber> getPhones() {return phones;}
public void setPhones(List<PhoneNumber> phones) {
                       this.phones = phones;}

// Business method to add a phone number to the customer
public void addPhone(PhoneNumber phone) {
this.getPhones().add(phone);
// Update the phone entity instance to refer to this customer
phone.addCustomer(this);}
```

```
}
```

Dans l'exemple précédent :

- l'annotation **@Entity** permet de marquer la classe Customer comme une classe d'objets persistants
- l'annotation **@Table(name="T_CUSTOMER")** permet d'associer la classe java "Customer" à la table relationnelle "T_CUSTOMER" .
- l'annotation **@Column(name="customer_name")** permet d'associer l'attribut "name" de la classe "Customer" à la colonne "customer_name" de la table relationnelle.
- l'annotation **@Id** permet de marquer le champ "number" comme **identifiant de l'entité (et comme clef primaire de la table)**.
- les autres annotations (**`@OneToMany`** , ...) seront approfondies dans un chapitre ultérieur.

NB: **Lorsqu'une annotation** (@Column ou ...) **n'est pas présente , les noms sont censés coïncider** (sachant que les minuscules et majuscules ont quelquefois de l'importance du coté SQL/relationnel selon le système d'exploitation hôte (ex: linux)).

Remarques importantes:

- Les annotations paramétrant le mapping objet/relationnel (@Id , @Column, @OneToMany, ...) peuvent soit être placées au dessus de l'attribut privé soit être placées au dessus de la méthode en "get".
  [au dessus des "private" --> plus lisible car dans le haut de la classe]
  [au dessus des "get" --> plus clair et/ou plus fonctionnel en cas d'héritage]

  Attention: il faut absolument garder un style homogène (si quelques annotations au dessus des "privates" et quelques annotations au dessus des "getXxx()" --> ça ne fonctionne pas !!! sauf si annotation spéciale **@Access** de JPA 2 avec *AccessType.FIELD* ou *PROPERTY*)

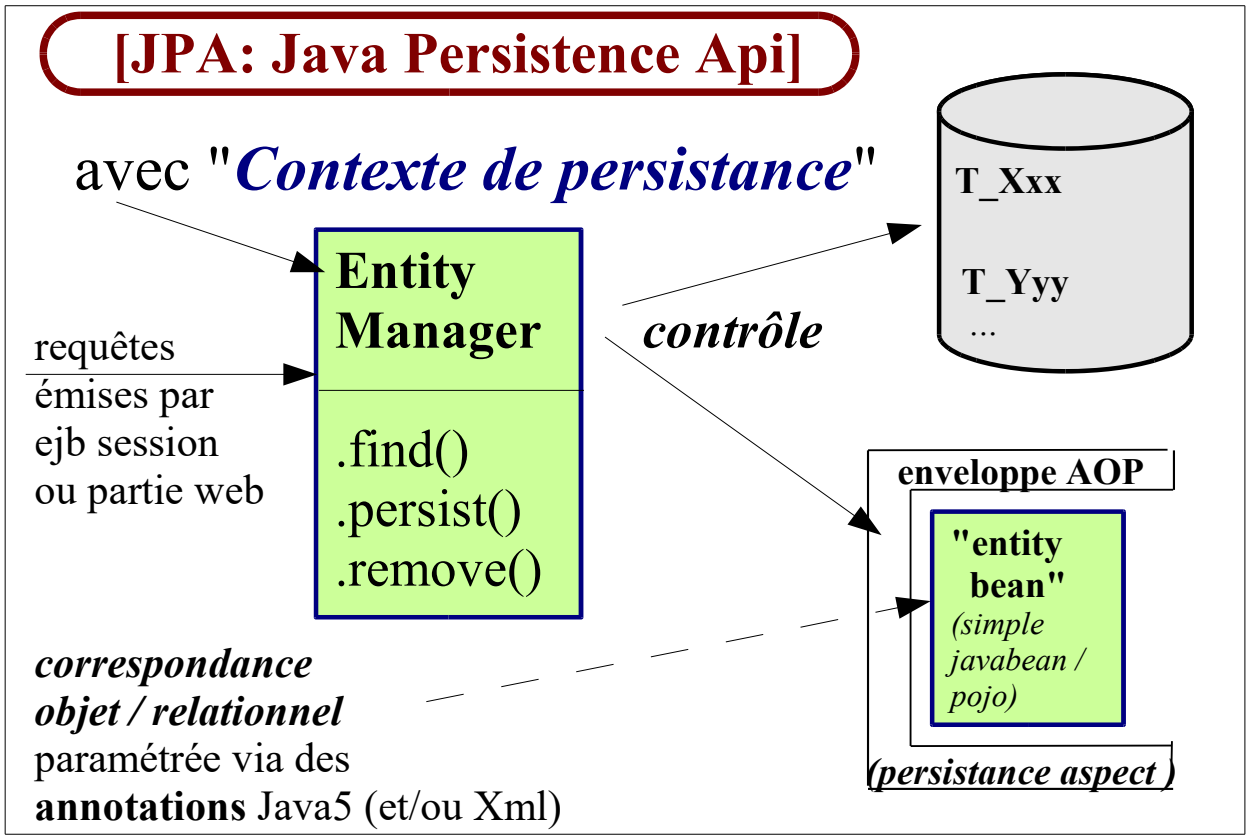
- Les annotations paramétrant des injections de dépendances (@Resource , @PersistenceContext , ...) peuvent soit être placées au dessus de l'attribut privé soit être placées au dessus de la méthode en "set".

- Même si une propriété (avec private +get/set) n'est marquée par aucune annotation , elle sera tout de même prise en compte lors du mapping objet-relationnel.
- Pour éventuellement (si nécessaire) désactiver le mapping d'une propriété d'une classe persistante, il faut utiliser l'annotation "**@Transient**" (au dessus du "private" ou du "getter"de la propriété).
- **@Enumerated(*EnumType.STRING*)** permet se stocker la valeur d'une énumération sous forme de chaîne de caractères dans une colonne d'une table relationnelle.
- **@Temporal(*TemporalType.DATE*** ou ***TemporalType.TIME*** ou ***TemporalType.TIMESTAMP***) permet de préciser la valeur significative d'une date java (java.util.Date) .

- ...

VII - EntityManager et entités persistantes

1. Entity Manager et son contexte de persistance



Un objet persistant (@Entity avec toutes ces annotations) doit être pris en charge par un **gestionnaire d'objets persistants** (implicitement associé à un "***contexte de persistance***") de façon à ce que la liaison avec la base de données soit bien gérée.

Ci après , figure l'interface prédéfinie "**EntityManager**" .

Celle-ci comporte tout un tas de **méthodes fondamentales** permettant de **déclencher/contrôler des opérations classiques de persistance (*recherches, mises à jour , suppressions , ...*)**.

```
package javax.persistence;

/*Interface used to interact with the persistence context. */
public interface EntityManager {

public void persist(Object entity); /* Make an instance managed and persistent.*/

public <T> T merge(T entity); /* Merge the state of the given entity
                                 into the current persistence context. */
public void remove(Object entity); /* Remove the entity instance. */

public <T> T find(Class<T> entityClass, Object primaryKey); /* Find by primary key. */
public <T> T getReference(Class<T> entityClass, Object primaryKey);
                                 /* Get an instance, whose state may be lazily fetched. */
```

```
public void flush(); /* Synchronize the persistence context to the underlying database.*/
public void setFlushMode(FlushModeType flushMode);
public FlushModeType getFlushMode();
public void lock(Object entity, LockModeType lockMode);
            /* Set the lock mode for an entity object contained in the persistence context.*/

public void refresh(Object entity); /* Refresh the state of the instance from the database */

public void clear(); /* Clear the persistence context */
public boolean contains(Object entity);
                    /* Check if the instance belongs to the current persistence context*/
public Query createQuery(String qlString);
                     /* Create an instance of Query for executing a JPQL statement */
public Query createNamedQuery(String name); /* in JPQL or native SQL */
public Query createNativeQuery(String sqlString);
public Query createNativeQuery(String sqlString, Class resultClass);
public Query createNativeQuery(String sqlString, String resultSetMapping);
public void joinTransaction(); /* join JTA active transaction */
public Object getDelegate(); /* implementation specific underlying provider */

public void close(); /*  Close an application-managed EntityManager */
public boolean isOpen();
public EntityTransaction getTransaction();
}
```

Un **contexte de persistance** (associé à *EntityManager*) peut:

- soit être **créé et géré par le conteneur** (ex: EJB , éventuellement WEB)
  puis injecté via l'annotation "**@PersistenceContext**" ou bien récupéré via un lookup JNDI .
- soit être explicitement créé via **createEntityManager**() de **EntityManagerFactory**().
  Son cycle de vie est alors à gérer (==> ***.close()*** à explicitement appeler)

2. Transaction JPA

L'interface "**EntityManager**" d'un **contexte de persistance** comporte une méthode ***getTransaction***() retournant de façon abstraite un objet permettant de contrôler (totalement ou partiellement) la transaction en cours:

```
public interface EntityTransaction {

public void begin(); /* Start a resource transaction. */
public void commit(); /* Commit the current transaction */
public void rollback(); /* Roll back the current transaction.*/

public void setRollbackOnly(); /* Mark the current transaction so that the only possible
outcome of the transaction is for the transaction to be rolled back.*/
public boolean getRollbackOnly();

public boolean isActive();
}
```

Remarque importante:

Ces transactions JPA peuvent en interne s'appuyer sur:

- ***des transactions simples et locales JDBC***
  ou bien
- ***des transactions complexes JTA (distribuées)***

Selon la configuration de ***META-INF/persistence.xml*** et la configuration du serveur d'application hôte (Jboss, WebSphere, Tomcat , ...).

D'autre part, en fonction du contexte d'intégration de JPA, les transactions seront explicitement ou implicitement gérées d'une des façons suivantes:

- **gestion déclarative** (et automatique/implicite) des **transactions** au sein des **EJB3**
- **gestion déclarative** (et automatique/implicite) des **transactions** au sein de **Spring**
- **gestion explicitement programmée des transactions** (sans framework d'intégration)

Exemple (avec gestion explicite):

```
try {
    entityManager.getTransaction().begin();
    Adresse nouvelleAdresse = new Adresse("rue elle","80000","Amiens");
    Client nouveauClient = new Client();
    nouveauClient.setNom("nom du nouveau client");
    nouveauClient.setAdressePrincipale(nouvelleAdresse);
    this.entityManager.persist(nouveauClient);
    num_cli = nouveauClient.getIdClient();
    System.out.println("id du nouveau client: " + num_cli);
    entityManager.getTransaction().commit();
    }
     catch (Exception e) {
       this.entityManager.getTransaction().rollback();
       e.printStackTrace();
     }
```

3. Différents états - objet potentiellement persistant

Etats objets	*Caractéristiques*
transient (proche état détaché)	Nouvel objet créé en mémoire, pris en charge par la JVM Java mais par encore contrôlé par une session Hibernate ou bien l'entityManager de JPA (le mapping objet/relationnel n'a jamais été activé). un tel objet n'a quelquefois par encore de clef primaire (elle sera souvent attribuée plus tard lors d'un appel à entityManager.persist() ou session.save())
persistant	objet actuellement sous le contrôle d'une session Hibernate ou de l'entityManager de JPA *(avant session.close() ou entityManager.close())*. Le mapping objet/relationnel est alors actif et une synchronisation (mémoire ==> base de données) est alors automatiquement déclenchée suite à une mise à jour (changement d'une valeur d'un attribut).
détaché	objet qui n'est plus sous le contrôle d'une session Hibernate ou de l'entityManager de JPA *(après session.close() ou entityManager.close())* . Les valeurs en mémoire sont conservées mais ne seront plus mises à jour (mapping objet/relationnel désactivé). Un objet détaché pourra éventuellement être ré-attaché à une session Hibernate (lors d'un update ou save_or_update() sur une session hibernate ou lors d'un appel à merge() sur l'entityManager de JPA) et ses éventuelles nouvelles valeurs pourront alors être synchronisées dans la base de données.

4. Cycle de vie d'un objet JPA/Hibernate

transient ==> **persistant** (==> ...)

(via **new**) (via ***session.save()*** ou ***entityManager.persist()***)

insert into ....

ou bien (après hibernation prolongée):

persistant ==> **détaché** (==> ***persistant***)

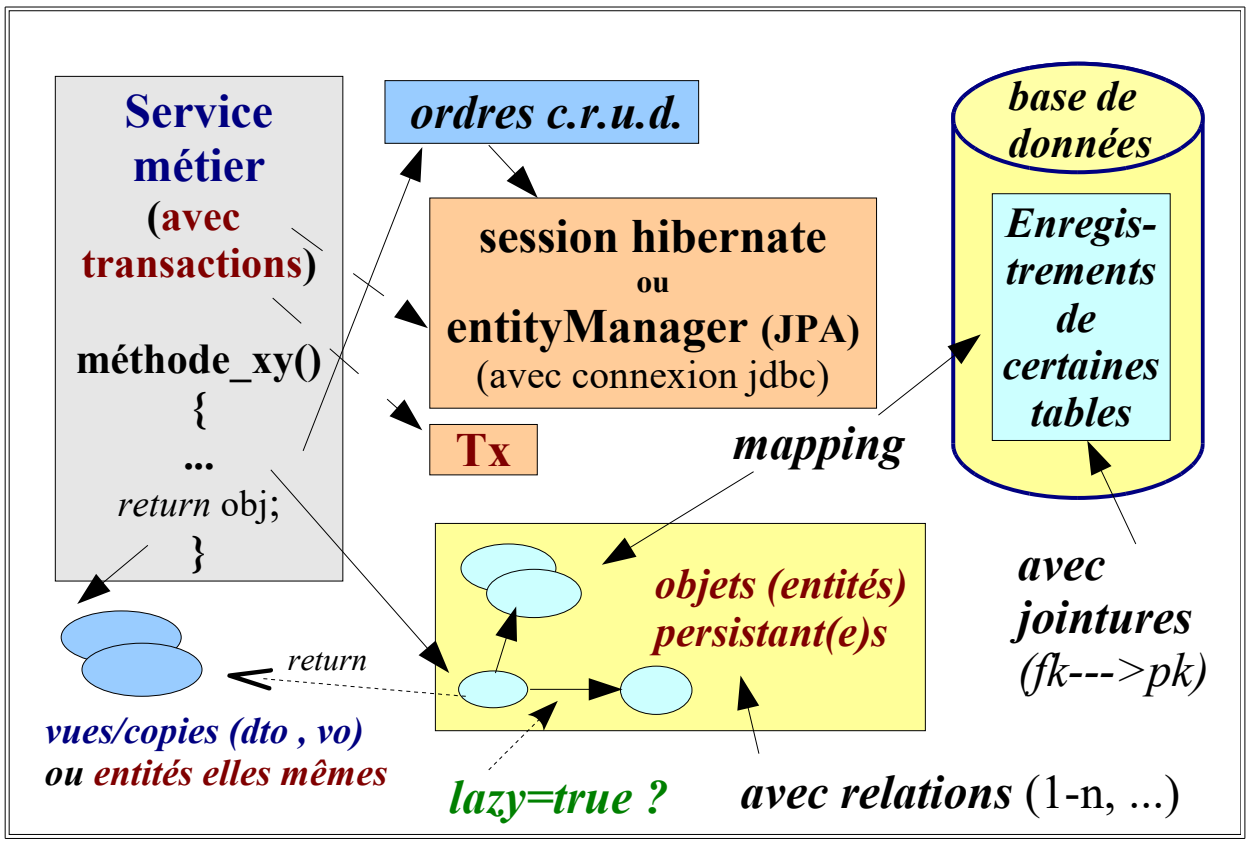
(via ***query***) (via session.close() ou entityManager.close()) (via ***session.update()*** , ... ou ***entityManager.merge()***)

select .... ***update .... set ...***

5. Synchronisation automatique dans l'état persistant

Synchronisation automatique dans l'état persistant (.update() ou .persist() non obligatoire) , mais .save() ou .merge() obligatoire pour un objet détaché.

6. objet persistant et architecture n-tiers



Remarques importantes:

- La technologie O.R.M. JPA ou hibernate sert essentiellement à remonter (et gérer) en mémoire un ensemble d'objets Java en tous points synchronisés avec les enregistrements d'une base de données relationnelle.
- Pour obtenir de bonnes performances (lazy=true) sans pour autant trop compliquer le code de l'application, il faut considérer les objets "entités persistantes" comme une structure d'ensemble orientée objet virtuellement liée à l'ensemble des données de la base (sachant que les mécanismes internes de JPA/Hibernate remontent les données en mémoire qu'en fonction des accès réellement effectués sur la structure objet [lazy=true]).
- La couche métier appelante (généralement développée avec Spring ou des EJB) doit souvent retourner des valeurs vers la couche présentation (IHM). Ces valeurs sont soit des références directes sur les entités persistantes elles mêmes ou bien des copies partielles sous formes de vues métiers (objets sérialisables / D.T.O. / V.O.) .
- Lorsque JPA/Hibernate est intégré dans Spring ou les EJB , les transactions sont automatiquement gérées par le conteneur et le code est alors significativement simplifié.

7. Principales méthodes JPA / EntityManager et Query

Principales méthodes de l'objet EntityManager:

méthodes	*traitements*
.**find**(class,pk)	Recherche en base et retourne un objet ayant les classe java et clef primaire indiquées
.**createQuery**(req_jpql)	Créer une requête JPQL et retourne cet objet
.**refresh**(obj)	Met à jour les valeurs d'un objet en mémoire en fonction de celles actuellement présentes dans la base de données (reload) .
.**persist**(obj_détaché) *ou bien assez inutilement* .***persist****(obj_déjà_persistant)*	Rend persistant un objet détaché : Sauvegarde les valeurs d'un nouvel objet dans la base de données (***insert into ...***) et retourne une exception de type `EntityExistsException` si l'entité existe déjà . La clef primaire est quelquefois automatiquement calculée lors de cette opération
.**merge**(obj_détaché) *ou bien assez inutilement* .***merge****(obj_déjà_dans_contexte_persistance)*	Sauvegarde ou bien met à jour (***update*** ...) les valeurs d'un objet dans la base de données. Cette méthode s'appelle .***merge***() car l'entité passé en argument peut quelquefois être utilisée pour remplacer les valeurs d'une ancienne version (avec la même clef primaire) déjà présente dans le contexte de persistance.
.**remove**(obj)	Supprime l'objet (delete ... from .... where .... dans la base de données)
.**flush**() déclenché indirectement via .***commit***()	Synchronise l'état de la base de données à partir des nouvelles valeurs des entités persistantes qui ont été modifiées en mémoire.

manager.**persist**() de **JPA** correspond à peu près au session.***save***() de ***Hibernate***

manager.**find**() de **JPA** correspond à peu près au session.***get***() de ***Hibernate***

manager.**remove**() de **JPA** correspond à peu près au session.***delete***() de ***Hibernate***

manager.**merge**() de **JPA** correspond à peu près au session.***save_or_update***() de ***Hibernate***

NB:

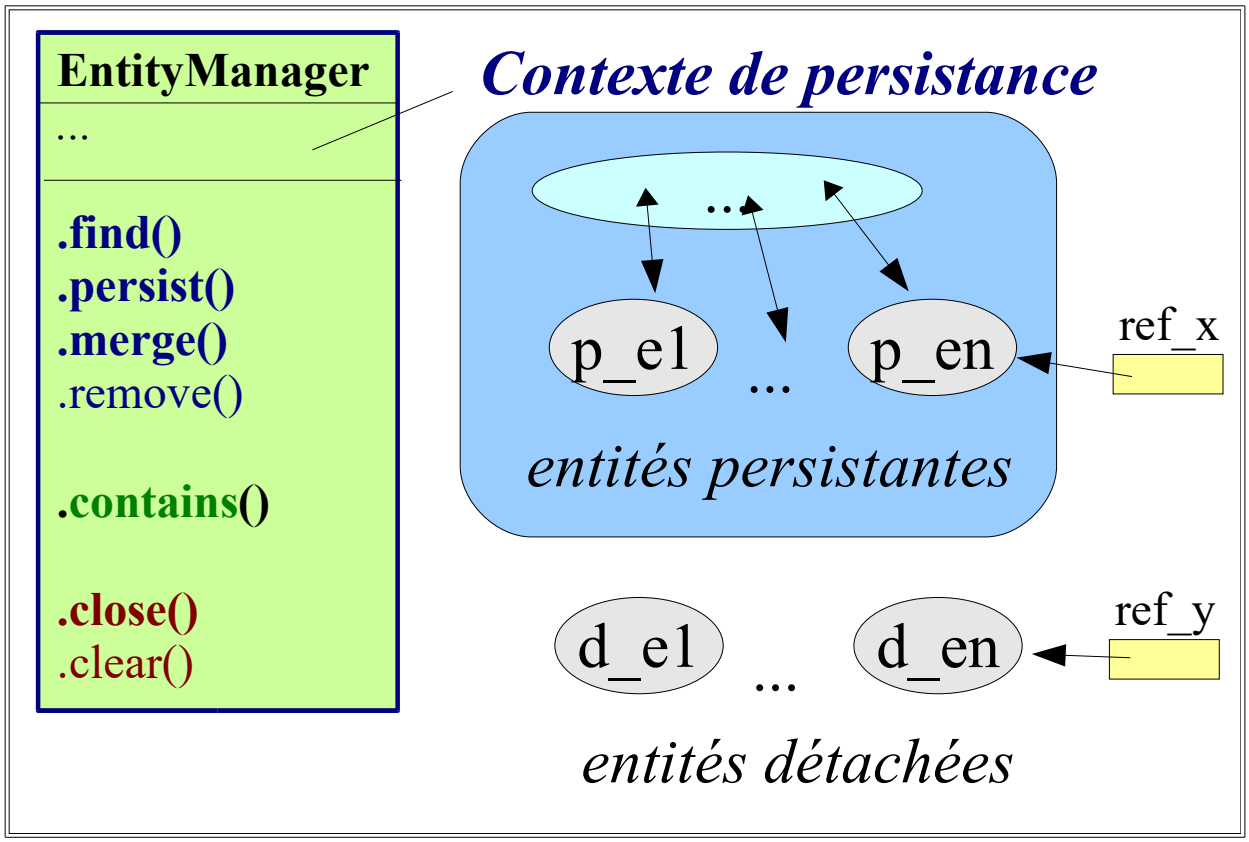
entityManager.getTransaction().**commit**() déclenche automatiquement *entityManager*.***flush()***

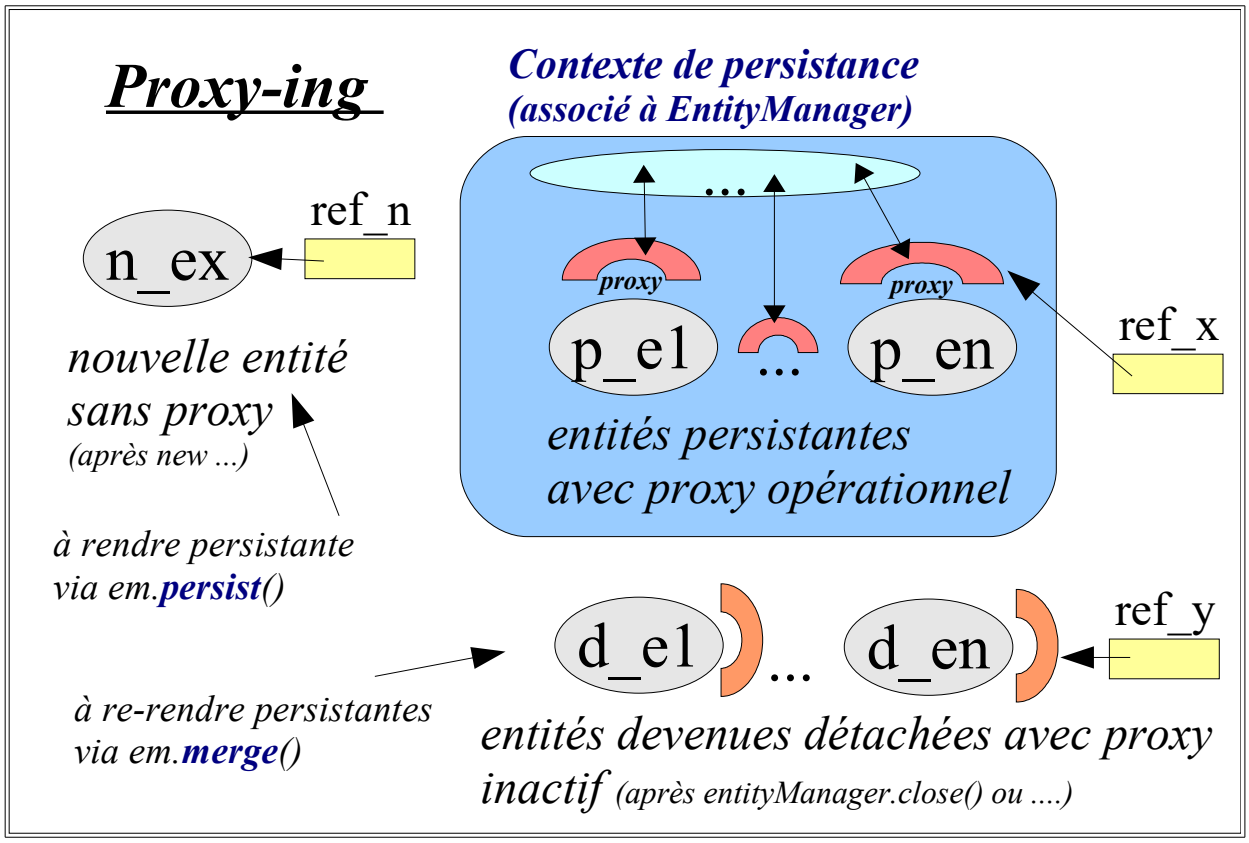
.**clear**() vide le contexte de persistance [tout passe à l'état détaché] sans effectuer de *.flush*()

Principales méthodes de l'objet Query:

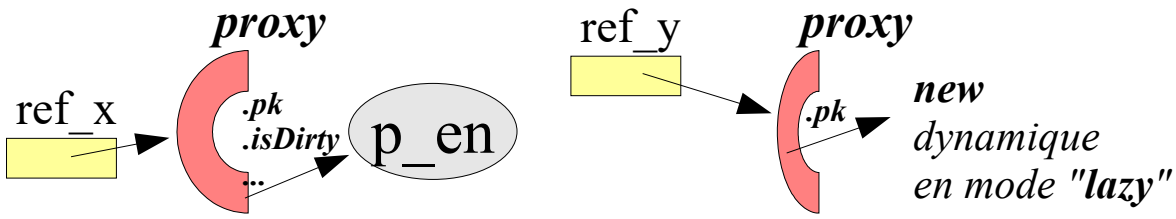
méthodes	*traitements*
.**setParameter**(paramName,paramValue)	met à jour un paramètre de la requête jpql
.**getSingleResult**()	récupère l'unique objet d'une requête simple.
.**getResultList**()	retourne sous forme d'objet "***java.util.List***" la liste des objets retournés en résultat de la requête JPQL.
.**executeUpdate**()	exécute une requête autre qu'un select
.***setMaxResults***()	fixe le nombre maxi d'éléments à récupérer

8. Contexte de persistance et proxy-ing





Proxy-ing et persistance



Le "**proxy**" *(généré par cglib ou autre)* sert entre autres à
- ***marquer les entités comme devant être mises à jour*** en base *(isDirty=true)* lors du prochain "***.flush()***"
- remonter les entités en mémoire de façon différée (***lazy***).

ref_persistant = entityManager.***merge***(ref_détaché);
ref_proxy_sans_entité = entityManager.***getReference***(class,pk);

VIII - Langage de requêtes JPQL

1.1. Présentation générale de HQL et JPQL

JPQL signifie ***JPA Query Language***
HQL ou **HbQL** signifie ***Hibernate Query Language***

Ces 2 langages correspondent à des ***minis langages dérivés de SQL*** et qui permettent d'**exprimer des requêtes** dans le cadre un **mapping objet/relationnel** géré par **un contexte de persistance JPA** ou par une **session Hibernate**.

1.2. Principaux éléments de syntaxe de JPQL et HQL

Le langage JPQL (ou HQL) n'est pas sensible à la casse pour sa partie SQL ("select" est équivalent à "SELECT") mais est sensible à la casse pour sa partie JAVA (la classe "Cat" n'est pas la même de "CAT" et la propriété c.weight n'est pas la même que c.WEIGHT).

Pour récupérer toutes les instances persistantes d'un certain type (c'est à dire tous les enregistrements de la table relationnelle associée) , une simple requête du genre "***from Cat***" *(ou bien "**from ppp.Cat**" en précisant optionnellement le package java)* est suffisante .

NB: La requête "***from Cat***" retourne non seulement des instances de la classe "***Cat***" mais également des instances des ***éventuelles sous classe***s (ex: "***DomesticCat***") . Le comportement de JPQL/HQL est bien en accord avec le "***polymorphisme***" du monde objet java.

De façon à pouvoir faire référence à une **instance** (du type précisé par from) dans une autre partie de la requête (ex: where) , on a généralement besoin d'un **alias** que l'on introduit par le mot clef optionnel "**as**" :

"from Cat **as** c" <==> "from Cat c"

"select c from Cat as c where c.weight > 8"

NB: la partie where est exprimée en utilisant la syntaxe ***nom_alias_instance.nom_propriété***
et non pas ~~*nom_alias_table.nom_colonne*~~ .

Autrement dit , la syntaxe de JPQL/HQL est centrée sur le monde objet (java) et pas sur le monde relationnel .

1.3. Quelques exemples de requêtes JPQL/HQL:

Rappel : **syntaxe** générale du **SELECT** du langage **SQL** :

SELECT fieldlist **FROM** tablenames [**WHERE** searchcondition]
[**GROUP BY** fieldlist [**HAVING** searchconditions]] [**ORDER BY** fieldlist]

Exemples JPQL/HQL:

```
SELECT cat.name FROM DomesticCat AS cat WHERE cat.name LIKE 'fri%'
```

```
SELECT avg(cat.weight), sum(cat.weight), max(cat.weight),
count(cat)  FROM Cat cat  ==> retourne une liste avec un seul
élément (ligne) de type Object[]
```

```
FROM Cat cat WHERE cat.mate.name is not null
```

```
SELECT c FROM Customer c JOIN c.orders o WHERE c.status = 1
(où c.orders est paramétré par @OneToMany ou @ManyToMany ...)
```

… **where** o.country **IN** ('UK', 'US', 'France')

… **where town** **LIKE** 'P%' où % signifie sous chaine quelconque de 0 à n caractères

… **where** p.age **BETWEEN** 15 and 19

```
SELECT DISTINCT auth FROM Author auth
    WHERE EXISTS
      (SELECT spouseAuth FROM Author spouseAuth WHERE spouseAuth = auth.spouse)
```

```
SELECT auth FROM Author auth WHERE auth.salary >= ALL(SELECT a.salary FROM Author a
WHERE a.magazine = auth.magazine)
```

```
SELECT mag FROM Magazine mag WHERE (SELECT COUNT(art) FROM mag.articles art) > 10
```

```
SELECT pub FROM Publisher pub JOIN pub.magazines mag ORDER BY o.revenue, o.name
```

1.4. Lancement d'une requête JPQL (JPA)

Exemples:

```
import javax.persistence.TypedQuery;
...
public List<Client> getAllClient()
        {
        TypedQuery<Client> query  =
              entityManager.createQuery("Select c from Client as c" , Client.class);
        return query.getResultList();
        }   ...
```

```
public   Categorie  getCategorieprincipale() {
                Categorie cat=null;
                Query q = entityManager.createQuery("select c from Categorie as c  "
                                                + "where  c.parentId  is  null");
                cat = (Categorie) q.getSingleResult();
                return cat;
        }

public List<Element> getElementByDesignation(String designation) {
                Query q = entityManager.createQuery("select e from Element as e "
                        +"where  e.designation=:design");
                q.setParameter("design", designation);          return q.getResultList();
        }
```

Remarque importante:

Les API "Hibernate" et "JPA 1" sont nées à l'époque de la transition entre les jdk <=1.4 et les jdk >= 1.5 (lorsqu'il n'existait pas encore de "generic" : List au lieu de List<T>) .

La méthode createQuery de l'interface EntityManager est surchargée.
Il existe une version qui renvoie un "TypedQuery" et qui accepte un second paramètre de type Class permettant de préciser le type de retour précis ce qui évite un warning .

D'autre part, les méthodes createQuery() et setParameter() ont été prévues pour s'enchaîner et l'on peut donc écrire :

```
return
 entityManager.createQuery("select c from Client as c where c.age >=:ageMini", Client.class)
                .setParameter("ageMini",18)
                .getResultList();
```

1.5. Lancement d'une requête native avec JPA

```
Query query = entityManager.createNativeQuery(
                    "SELECT NAME, SURNAME, AGE FROM PERSON");
List list = query.getResultList();
```

résultat ===>
une liste de tableaux (0bject[]) avec NAME à l'index 0 , SURNAME à l'index 1 et AGE à l'index 2

Via une version surchargée de createNativeQuery ayant un second paramètre on peut quelquefois préciser le type de retour souhaité (si compatible) .

1.6. NamedQuery placée dans une classe d'entité persistante

```
@Entity
@NamedQuery(name="Country.findAll", query="SELECT c FROM Country c")
public class Country {
  ...
}
```

ou bien

```
@Entity
@NamedQueries({
   @NamedQuery(name="Country.findAll",
            query="SELECT c FROM Country c"),
   @NamedQuery(name="Country.findByName",
            query="SELECT c FROM Country c WHERE c.name = :name"),
})
public class Country {
  ...
}
```

et appel via createNamedQuery("named_of_query" , ResultType.class) :

```
TypedQuery<Country> query =
    entityManager.createNamedQuery("Country.findAll", Country.class);

 List<Country> results = query.getResultList();
```

Intérêt de "NamedQuery" par rapport aux requêtes ordinaires ?

---> c'est une simple affaire de style .

- Dans le cas où le code JPA est dans un composant spécialisé de type DAO, les "NamedQuery" n'apportent quasiment aucun intérêt.
- Dans le cas où un service métier n'utilise aucun DAO en arrière plan et utilise l'entityManager en direct, les "NamedQuery" ont l'avantage de bien séparer détails de persistance avec logique métier.

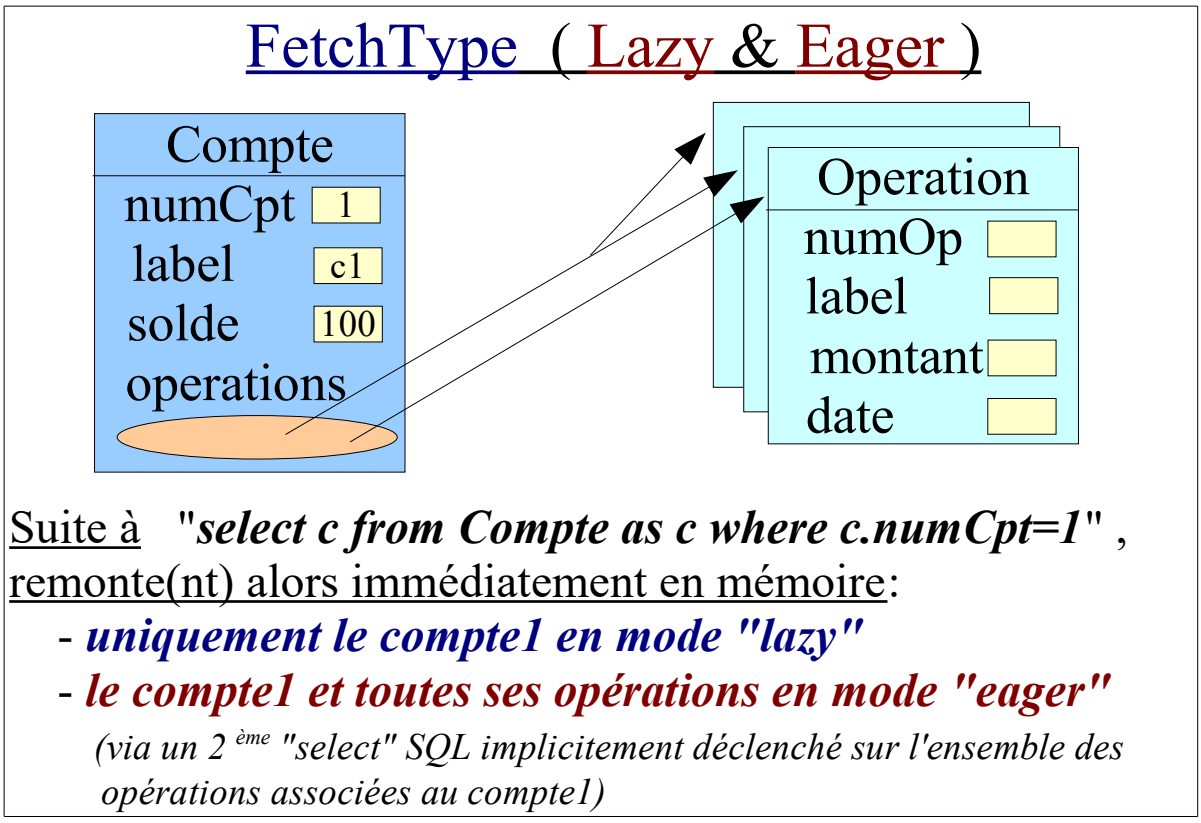
IX - O.R.M. JPA (généralités)

1. Vue d'ensemble sur les entités, valeurs et relations

1.1. Entités et objets valeurs (embedded)

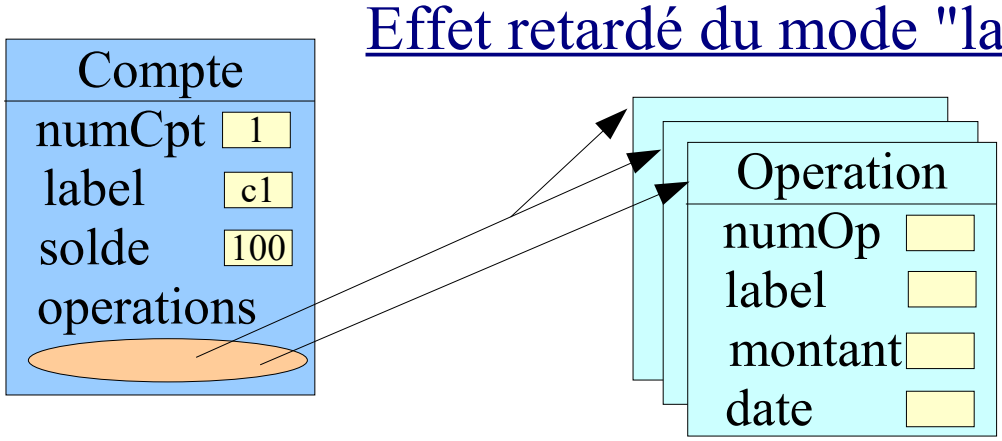
- Une **entité** (@Entity) doit absolument comporter une **clef primaire** (@Id)
- Un sous objet imbriqué (@Embedded) n'a pas de clef primaire et doit absolument être référencé par une entité pour exister.

1.2. FecthType (lazy & eager)



L'attribut optionnel **lazy="true"** d'hibernate (ou bien **fetchType=Lazy** de JPA) permet de demander une ***initialisation tardive*** des membres d'une collection (les valeurs des éléments de la collection ne seront recherchées en base que lorsqu'elles seront consultées via par exemple un itérateur java). Ceci permet d'obtenir dans certains cas de meilleurs performances . Il faut cependant veiller à récupérer les valeurs de la collection avant le commit et la fermeture de la session (sinon ==> c'est trop tard : ***LazyInitialisationException***).

Il est fortement conseillé d'affecter explicitement la valeur "lazy" ou "eager" au paramètre fetchType (très important sur le plan des performances).



- Si l'on accède qu'à "c1.getLabel() et c1.getSolde() les opérations ne sont alors jamais remontées en mémoire.
- Par contre, ***dès le parcours de la collection des opérations*** *(via un itérateur ou une boucle for)*, ***Les entités "Opération" sont alors automatiquement remontées en mémoire*** par JPA ***via une série de petits "select"*** SQL (jusqu'à "***1+N***" en tout").

Exemple JPA:

---> **fetch=FetchType.LAZY** or **fetch=FetchType.EAGER;**

```
@OneToMany( fetch=FetchType.LAZY , mappedBy="compte" )
private  List<Operation> dernieres_operations ;
```

NB : Si le mode "LAZY" est configuré dans une annotation JPA (ou dans un fichier hbm.xml), il est tout de même possible d'***expliciter ponctuellement une demande de chargement "tout d'un seul coup" via une requête JPQL*** du type "*select **distinct** c from Compte c **inner join fetch c.operations** inner join c.proprietaires cli where cli.numero = :numCli*"

1.3. Operations en cascade

Lorsqu'une opération (delete , updade , save, ...) est effectuée sur une entité persistante, elle est alors automatiquement répercutée sur les sous objets de type "valeurs/embbeded" .

Lorsque par contre on a affaire à une relation (1-1 ou 1-n) de type "entité-entité" , l'opération (save, update , delete) effectuée sur l'entité principale n'est pas automatiquement répercutée sur les entités liées. On peut cependant demander explicitement à JPA (ou hibernate) de répercuter une opération de mise à jour de l'entité principale vers une ou plusieurs entité(s) liée(s) via l'attribut ***cascade*** d'une relation .

Syntaxe JPA (dans les annotations):

```
@OneToMany(cascade=CascadeType.ALL , ....)
```

ou bien avec ensemble de cascades combinées:

```
cascade = {CascadeType.PERSIST, CascadeType.MERGE}, ...
```

NB: depuis JPA2 , la cascade "all" + "***delete-orphan***" permet en plus de supprimer automatiquement en base des entités "enfants" qui seraient détachées de leurs parents (suite un une suppression de référence(s) dans une collection par exemple) .

NB: ceci fonctionne même si la mention "cascade" n'est pas précisée dans le schéma relationnel de la base de données.

2. Identité d'une entité (clef primaire)

2.1. Générateurs de clefs primaires

Une **clef primaine** (*ex*: numéro de facture) est assez souvent **générée automatiquement** via un **algorithme** basé sur la notion de **compteur**.

Exemple (JPA):

```
@Id
@GeneratedValue(strategy=GenerationType.AUTO) //pour auto incrémentation coté base
@Column(name="numero")
public long getNum() {
        return num;
}
```

2.2. Eventuelle clef primaire composée

Avec JPA:

Si une clef primaire est composée, il faut alors utiliser les annotations

@EmbeddedId (pour annoter la propriété lié au sous objet "clef primaire")

et

@Embeddable (pour annoter une c*lasse de clef primaire devant être sérialisable , publique , vu comme un "JavaBean" et devant coder equals() et hashcode()*).

Exemple de classe de clef primaire composée:

```
@Embeddable
public class CustomerPK implements java.io.Serializable
{
  private long id;
  private String name;

  public CustomerPK()  {  }

  public CustomerPK(long id, String name)  {
```

```
    this.id = id;      this.name = name;
  }

  public long getId()   {  return id;  }

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

  public String getName()   {      return name;   }

  public void setName(String name)   {      this.name = name;   }

  public int hashCode()   {
    return (int) id + name.hashCode();
  }

  public boolean equals(Object obj)
  {
    if (obj == this) return true;
    if (!(obj instanceof CustomerPK)) return false;
    if (obj == null) return false;
    CustomerPK pk = (CustomerPK) obj;
    return pk.id == id && pk.name.equals(name);
  }
}
```

```
@Entity
public class Customer implements java.io.Serializable
{
  CustomerPK pk;
  ...
  public Customer(){ }

  @EmbeddedId
  public CustomerPK getPk() {
    return pk;
  }

  public void setPk(CustomerPK pk){
    this.pk = pk;
  }
  ...
}
```

3. Propriétés d'une colonne

```
@Column(updatable = false, name = "flight_name", nullable = false, length = 50)
```

L'annotation @Column se transpose en sous balise <column> des fichiers de mapping hibernate (.hbm.xml).

4. Relations (1-1, n-1, 1-n et n-n)

Annotations JPA	*Significations*
@OneToOne	1-1 (fk [unique] ---> pk) *[<many-to-one unique='true'> de hibernate]*
@OneToMany	1-n (collection avec clef [pk <----fk=clef])
@ManyToOne	n-1 (fk ----> pk)
@ManyToMany	n-n (fk1 ----> (k1,k2) ---> pk2)

NB: les annotations **@OneToOne , @OneToMany** et **@ManyToMany** peuvent éventuellement comporter un attribut "***mappedBy***" dont la valeur correspond à la ***propriété qui sert à établir une correspondance inverse (et secondaire)*** au sein d'une **relation bidirectionnelle** .

NB: **les mises à jour des relations effectuées uniquement du coté secondaire d'une relation bi-directionnelle ne seront pas automatiquement sauvegardées** (tant que le coté principal de la relation n'aura pas été explicitement réajusté).

L'annotation **@JoinColumn** permet d'indiquer *(via name="...")* le ***nom de la colonne correspondant à la clef étrangère*** dans la base de données.

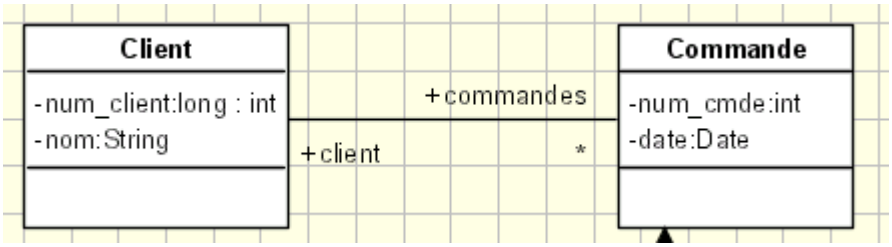
exemple:

```
@OneToOne(cascade = {CascadeType.ALL})
@JoinColumn(name = "ADDRESS_ID")
public Address getAddress()
{
   return address;
}

public void setAddress(Address address)
{
   this.address = address;
}
```

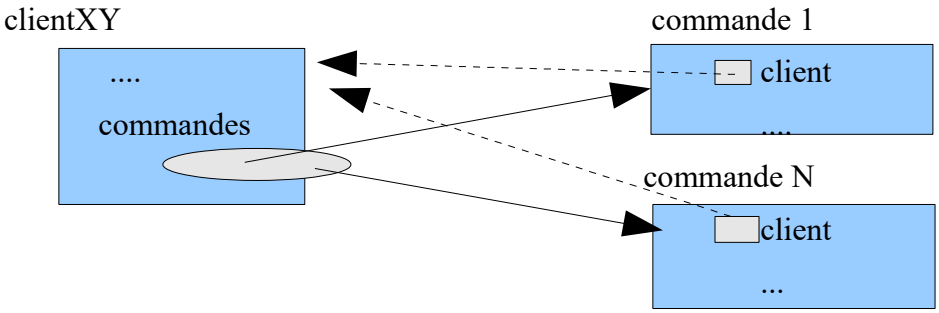
X - O.R.M. JPA – détails (1-n , 1-1 , n-n, ...)

1.1. Relations 1-n et n-1 (entité-entités)

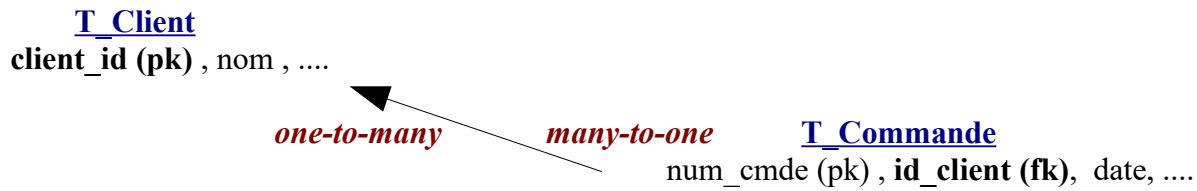
UML:



Java:



Base de données:



Annotations JPA:

```
@Entity
@Table(name = "T_Commande")
public class Commande
{
 @ManyToOne
  @JoinColumn(name = "id_client") //fk (nom de colonne dans la table)
private Client client;
...
   public Client getClient()   {      return client;   }
   public void setClient(Client  client)   {      this.client = client;   }
}
```

```
@Entity
@Table(name = "T_Client")
public class Client
{
 //NB : "client" (valeur de mappedBy) = nom (java) de la relation inverse
  @OneToMany(fetch=FetchType.LAZY,mappedBy="client")
 private List<Commande> commandes;
```

```
  ...
  public List<Commande> getCommandes()  {   return commandes;  }
  public void setCommandes(List<Commande> cmdes) {  this.commandes = cmdes;   }

  public void addCommande(Commande c) {
    if (commandes == null) commandes = new ArrayList<Commande>();
    commandes.add(c);   }
}
```

Rappels importants (pour JPA):

- La valeur de l'attribut "***mappedBy***" de l'annotation **@OneToMany** correspond à la ***propriété (dans l'autre classe java) qui sert à établir une correspondance inverse*** au sein d'une **relation bidirectionnelle** .
- L'annotation **@JoinColumn** permet d'indiquer (via name="...") le nom de la colonne correspondant à la clef étrangère dans la base de données.

1.2. Relations 1-n (avec JoinTable) évolutives vers du n-n

Bien qu'une relation 1-n (symétriquement n-1) soit souvent basée sur une jointure simple avec une clef étrangère en base, il est tout de même possible de mettre en place une relation 1-n avec une table de jointure en base . Déjà structurée en base comme une relation n-n , une telle relation 1-n a le mérite d'être facilement évolutive vers du n-n.

Exemple :

```
@Entity
@Table(name="Compte")
public class Compte {
      ...
      @ManyToOne //evolutif vers @ManyToMany
      @JoinTable(name="ClientCompte",
                  joinColumns={@JoinColumn(name="num_compte")},
                  inverseJoinColumns={@JoinColumn(name="num_client")})
      private Client client;
}
```

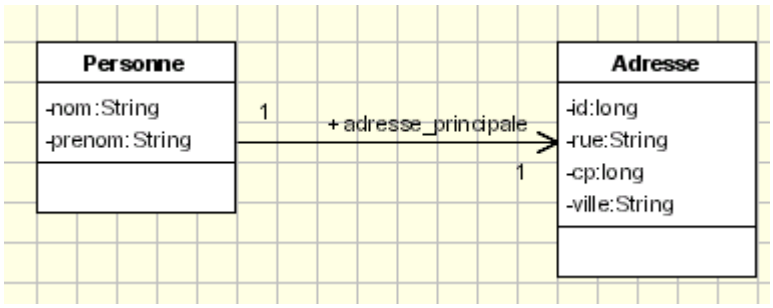
```
@Entity
@Table(name="Client")
public class Client extends Personne {
      ...
      @OneToMany(mappedBy="client")
      private List<Compte> comptes;
      …
}
```

avec

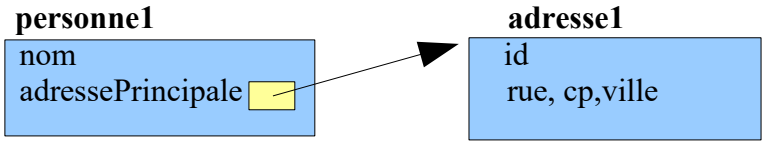
```
create table ClientCompte(   num_client integer,   num_compte integer,
                                primary key(num_client,num_compte));
ALTER TABLE ClientCompte ADD CONSTRAINT client_avec_compte_valide
      FOREIGN KEY  (num_compte) REFERENCES Compte(numero);
ALTER TABLE ClientCompte ADD CONSTRAINT compte_avec_client_valide
      FOREIGN KEY (num_client) REFERENCES Client(num_client);
```

1.3. Relation 1-1

UML:

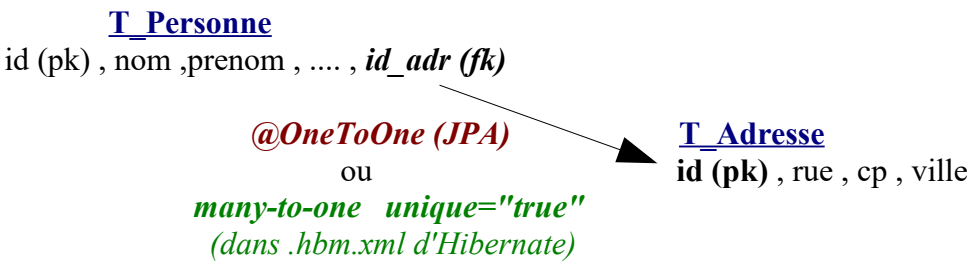


Java:



===> exemple: ***personne1.getAdressePrincipale().getVille()***;

Base de données:



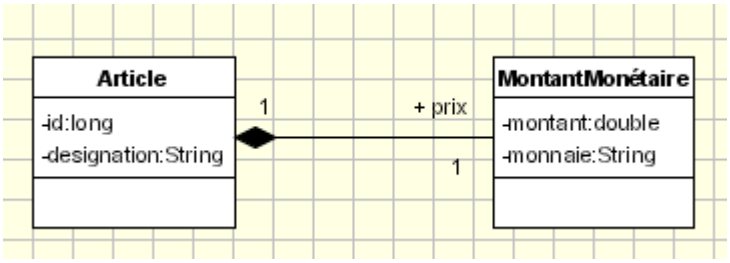
JPA:

```
@Entity
public class Personne
{
  @OneToOne()
   @JoinColumn(name = "id_adr") //fk
  private  Adresse   adressePrincipale;
   ...
   public  Adresse  getAdressePrincipale()  {
      return adressePrincipale;
   }
   public void setAdressePrincipale(Adresse adressePrincipale) {
      this. adressePrincipale = adressePrincipale;
   }
}
```

1.4. Sous objets incorporable au sein d'une entité

Appelés objets "*Valeur composite*" ou "*Component*" dans la terminologie *Hibernate* , et appelés objets "***Embeddable***" dans la terminologie ***JPA***, d'**éventuels sous objets composés(non partageables et sans clef primaire) peuvent être utilisés pour structurer une entité**.

UML:



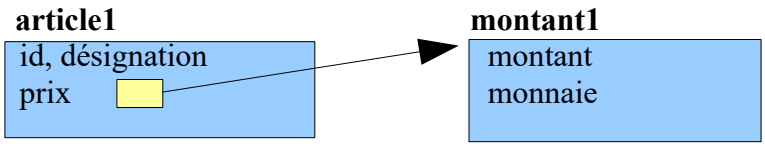
En base de données:

T_Article
id (pk) , designation ,***montant , monnaie*** , ...

==> mapping relationnel associé (par défaut):

- pas de table relationnelle séparée pour les sous objets "valeurs"
- liste des propriétés du composant directement incluse dans la table de l'objet contenant .

En Java:



==> xxx.***getPrix().getMontant()***; et xxx.***getPrix().getMonnaie()***;

NB: La classe java du composant imbriqué n'a pas d' ID (pas de clef primaire) . Elle respecte néanmoins les conventions "**JavaBean**" pour les éléments internes (ici montant et monnaie) .

JPA:

Une classe d'objet incorporable doit normalement être annotée via "**@Embeddable**" .

Une propriété d'une entité qui référence un objet imbriqué doit être annotée via **@Embedded** .

Les spécifications JPA se limitent à un seul niveau d'imbrication.

exemple:

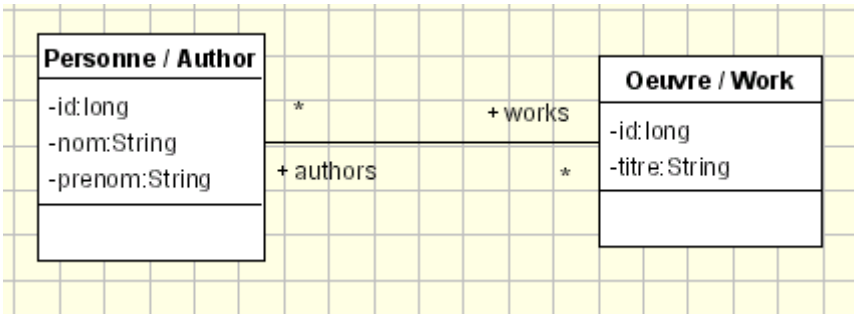
```
@Embeddable
public class MontantMonetaire
{
  private double montant;  // + get/set
  private String monnaie;  // + get/set
  public MontantMonetaire()   { }

  public MontantMonetaire(double montant, String monnaie)   {
    this.montant = montant;      this.monnaie = monnaie;
  }
}
```

```
@Entity
@Table(name = "T_Article")
public class Article
  private long id;  // avec get/set et @Id

  @Embedded
 /*  @AttributeOverrides({
    @AttributeOverride(name = "montant", column = @Column(name = "montant")),
    @AttributeOverride(name = "monnaie", column = @Column(name = "monnaie"))
  }) */
  private MontantMonetaire   prix;
  ...
  public MontantMonetaire getPrix()   {   return prix;   }
  public void setPrix( MontantMonetaire prix)   {    this.prix = prix;   }
}
```

NB : @AttributeOverride **n'est réellement utile que si les noms des colonnes sont différentes des noms des propriétés java de l'objet incorporé .**

1.5. Relation n-n ordinaire

UML:



Mapping relationnel ==> ***Table intermédiaire***

T_Author
auth_id (pk) , nom ,prenom , ...

Author_Work
author_id (fk) , work_id(fk)

many-to-many

T_Work
w_id (pk) , titre , ...

Java:
--> des collections dans les 2 sens
avec un sens principal et un sens secondaire (avec mappedBy).

JPA:

du coté "principal" de la relation (coté où les mises à jour seront "persistées"):

```
@Entity
public class Work
{
@ManyToMany( fetch = FetchType.LAZY)
  @JoinTable(name = "Author_Work",
                joinColumns = {@JoinColumn(name = "work_id")},
                inverseJoinColumns = {@JoinColumn(name = "author_id")})
private List<Author> authors;
...
  public List<Author> getAuthors()  {
    return authors;
  }
  public void setAuthors(List<Author> authors)  {     this. authors = authors;  }
..
}
```

NB: l'attribut ***joinColumns*** (de @JoinTable) correspond à la ***clef étrangère pointant vers l'entité courante*** et l'attribut **inverseJoinColumns** correspond à la **clef étrangère pointant vers les éléments de l'autre coté de la relation (ceux qui seront rangés dans la collection paramétrée)**.

et du coté inverse/secondaire:

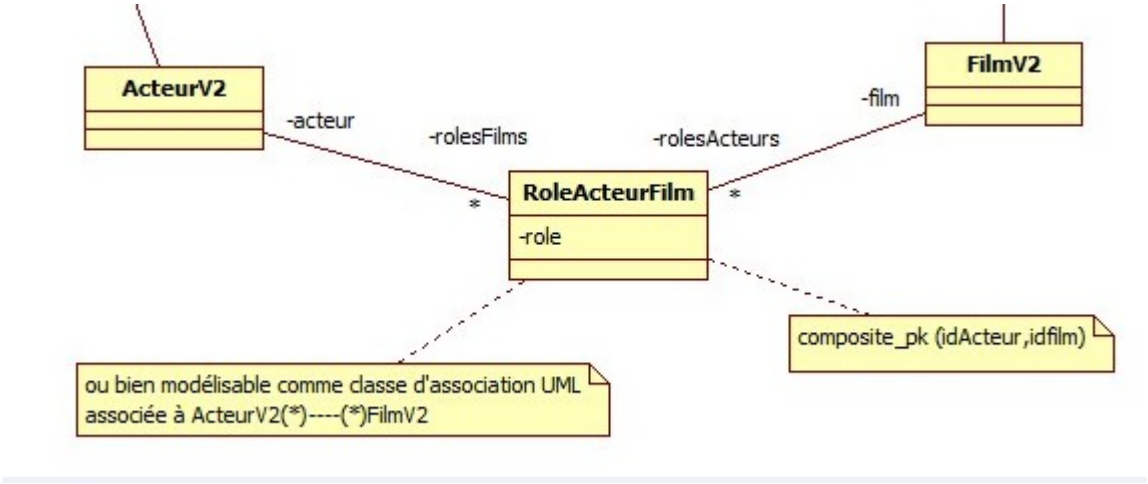
```
@Entity
public class Author
{
 @ManyToMany(mappedBy="authors")
  private List<Work> works;
...
  public List<Work> getWorks()  {
    return works;
  }
  public void setWorks(List<Work> works)  {     this.works = works;  }
}
```

1.6. Relation n-n avec détails d'association

Remarque importante sur les relations (n-n):

Lorsque la table de jointure comporte des informations supplémentaires autres que les clefs étrangères, on est alors obligé de décomposer la relation n-n en deux relations (1-n + n-1) et les éléments de la table de jointure sont alors vus comme des entités ayant une clef primaire (souvent composée par les 2 clefs étrangères).

Exemple:




```
CREATE TABLE ActeurFilmV2
     ( idActeur integer,
       idFilm integer ,
       role VARCHAR(64),
       primary key(idActeur,idFilm));

ALTER TABLE ActeurFilmV2 ADD CONSTRAINT avec_acteur_valideV2
           FOREIGN KEY  (idActeur) REFERENCES Acteur(idActeur);
ALTER TABLE ActeurFilmV2 ADD CONSTRAINT avec_film_valideV2
            FOREIGN KEY  (idFilm) REFERENCES Film(idFilm);
```

```
@Entity
@Table(name="Acteur")
public class ActeurV2  {

     @Id
     @GeneratedValue(strategy=GenerationType.AUTO)
     private long idActeur ;

     private String nom ;

     @OneToMany(mappedBy="acteur")
     private List<RoleActeurFilm> rolesFilms;

     ...
}
```

```
@Entity
@Table(name="ActeurFilmV2")
public class RoleActeurFilm {
     @EmbeddedId
     private RoleActeurFilmCompositePk pk;
     private String role;

     @ManyToOne
     @JoinColumn(name="idActeur", insertable=false, updatable=false)
     private ActeurV2 acteur;

     @ManyToOne
```

```
	@JoinColumn(name="idFilm" , insertable=false, updatable=false)
	private FilmV2 film;

	public RoleActeurFilm() {
		super();
	}

	public RoleActeurFilm(String role, ActeurV2 acteur, FilmV2 film) {
		super();
		this.role = role; this.acteur = acteur; this.film = film;
		pk=new RoleActeurFilmCompositePk(acteur.getIdActeur(),
						film.getIdFilm());
	}
	//+get/set , ...
}
```

NB: insertable=false, updatable=false est indispensable pour que JPA puisse gérer un double mapping où les colonnes "idFilm" et "idActeur" sont déjà précisées au sein de la clef primaire (composite pk).

```
@Embeddable
public class RoleActeurFilmCompositePk implements Serializable {
	private static final long serialVersionUID = 1L;

	private long idActeur;
	private long idFilm;

	public RoleActeurFilmCompositePk() {
		super();
	}

	public RoleActeurFilmCompositePk(long idActeur, long idFilm) {
		super();
		this.idActeur = idActeur;
		this.idFilm = idFilm;
	}

	//+get/set , equals et hashcode (generated by eclipse)
}
```

```
@Entity
@Table(name="Film")
public class FilmV2 {

	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)
	private long idFilm;
	private String producteur;
	private String titre;

	@Column(name="dateSortie")
	@Temporal(DATE) //partie significative = DATE et pas DateTime
	public Date date;


	@OneToMany(mappedBy="film")
	private List<RoleActeurFilm> rolesActeurs ;
```

```
        . . .

}
```

XI - Gestion des transactions (niveau EJB)

1. Transactions distribuées et commit à 2 phases

1.1. Qualités (A.C.I.D.) d'une transaction distribuée basique

Transactions distribuées

•**A.C.I.D.** ==> **A**tomicity , **C**onsistency , **I**solation , **D**urability

• L' **Atomicité** désigne le comportement "**tout ou rien**" (Le tout vu en tant qu'élément unique et atomique doit soit réussir , soit échouer). Il n'y a pas de demi-mesure.

•La **Consistance** d'une transaction désigne le fait que **les différentes opérations doivent laisser le système dans un état stable et cohérent**.

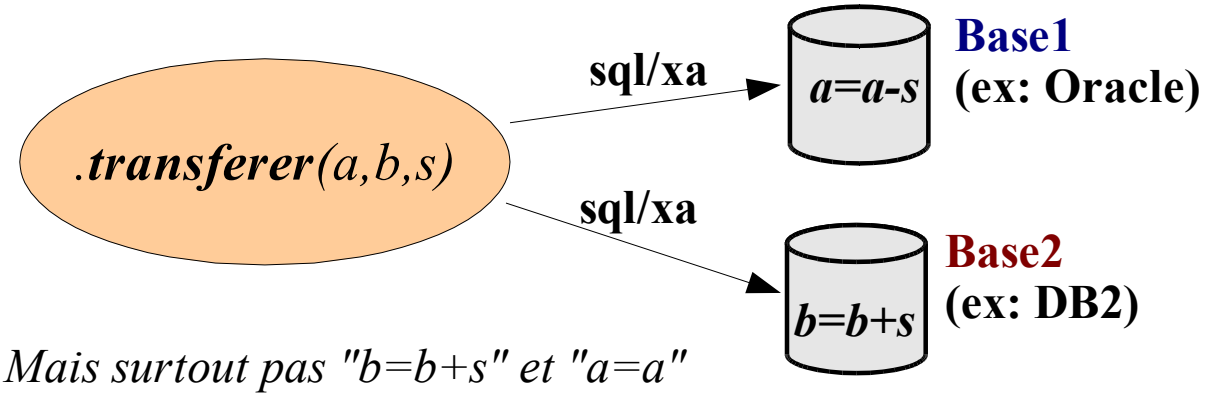
• Le concept d' **Isolation** signifie içi que **2 transactions concurrentes n'interfèrent pas entre elles** (Points critiques: résultats intermédiaires et opérations annulées).

•La **Durabilité** indique que **les résultats d'une transaction doivent absolument être mémorisés de façon durable** (sur un support physique) de façon à survivre suite à une éventuelle défaillance(un fichier de Log peut également être très utile).

1.2. Protocole XA pour le commit à 2 phases

Protocole XA et commit à 2 phases (1)

Une ***transaction distribuée*** peut faire intervenir de **multiples ressources** telles que celles-ci par exemple:
- Base 1 (ex: Oracle via JDBC) , Base 2 (ex: DB2 via JDBC)
- Moniteur transactionnel (ex:Tuxedo ou CICS via connecteurs).
- Système de message asynchrone (ex: MQSeries via JMS), ...



Mais surtout pas "b=b+s" et "a=a"

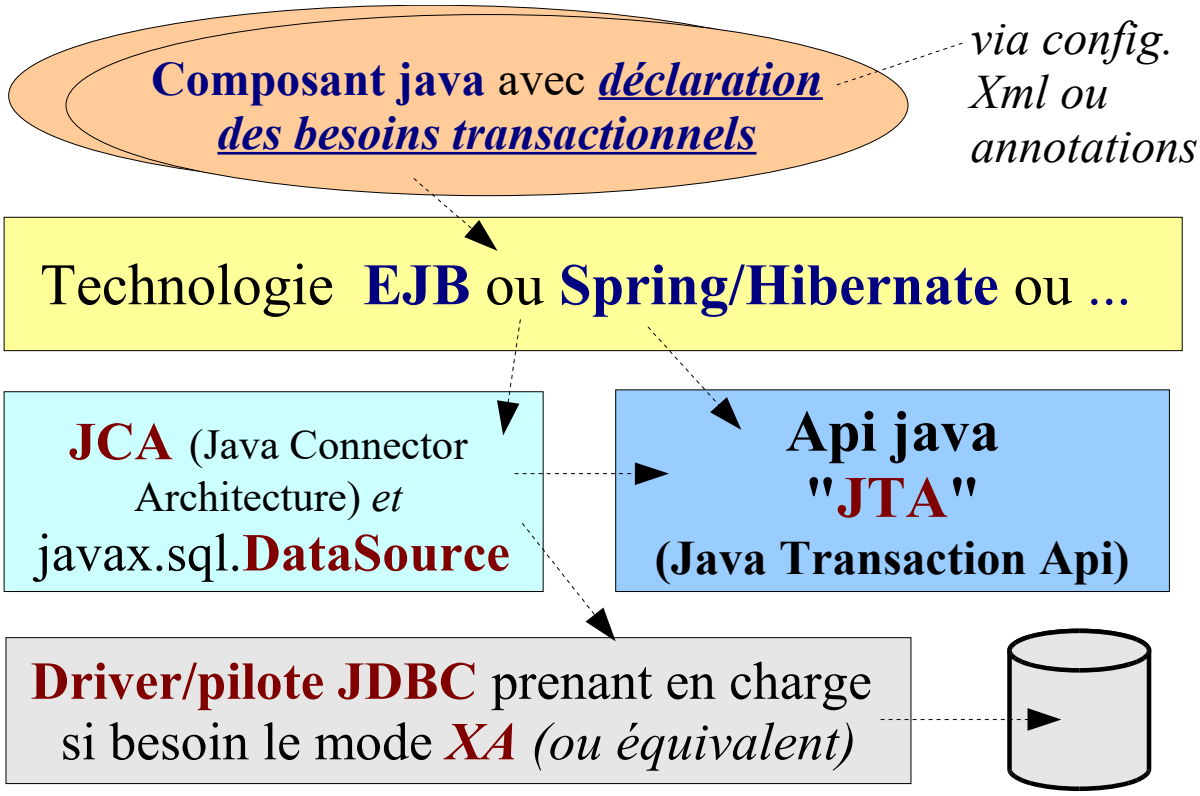
Protocole XA et commit à 2 phases (2)

De façon à ce que toutes les opérations à tous les niveaux (chacune des bases de données, ...) soient globalement annulées ou validées, on a recours à la technique suivante:

- 1 - Chaque ressource mise en jeu dans la transaction effectue des opérations dans une zone mémoire à part (ex: opérations SQL que l'on pourra éventuellement annuler) puis envoie un signal pour indiquer qu'à son niveau tout va bien.
- 2 - Un élément "pilote de la transaction" centralise ces acquittements.
- 3 - Si chaque protagoniste de la transaction distribuée a réussi sa tâche, le pilote envoie à chacun d'eux l'ordre d'entériner la mise à jour (commit final). Si un seul protagoniste de la transaction distribuée a échoué dans sa tâche, le pilote envoie à tout le monde l'ordre d'annuler la mise à jour (rollback final).

- Cette technique standard du **commit à deux phases** est formalisée au niveau d'un **protocole** normalisé dénommé **XA** .

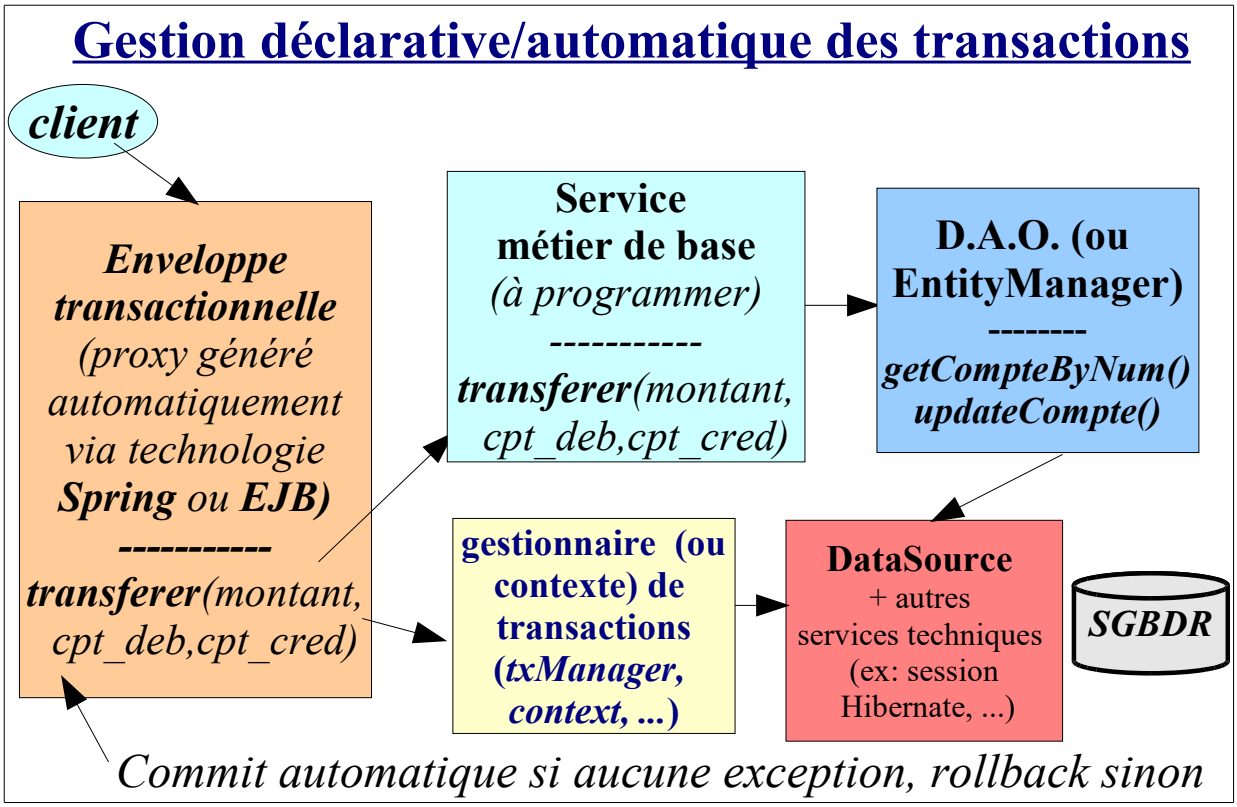
2. Infrastructure transactionnelle de JEE



Tout serveur JEE offre une infrastructure sérieuse pour bien gérer/piloter les transactions .

- L'Api JTA permet une bonne délimitation des transactions distribuées . Son utilisation directe nécessite beaucoup de ligne de code pour paramétrer le contexte transactionnel et pour explicitement déclencher les "commit" ou "rollback" .
- Les technologies de bas niveau "JCA , DataSource et JDBC/XA" servent à bien relayer les ordres de "commit" / "rollback" jusqu'à la source de données (SGBDR ou ...).
- Des technologies de haut niveau telles que EJB ou Spring/Hibernate permettent entre autres d'automatiser les transactions (commit implicite si aucune exception de remonte, rollback systématique sinon).

3. Gestion déclarative des transactions



Le code généré dans l'enveloppe transactionnelle est a peu près de cette teneur:

```
public void transferer(double montant, long num_cpt_deb, long num_cpt_cred){
// initialisation (si nécessaire) de  de l'entityManager de JPA
// selon existence dans le thread courant
tx = ...beginTransaction(); // sauf si transaction (englobante) déjà en cours
try{
      serviceDeBase.transferer(montant,num_cpt_deb,num_cpt_cred);
      tx.commit(); // ou ... si transaction (englobante) déjà en cours
}
catch(RuntimeException ex){      tx.rollback();   /* ou setRollbackOnly(); */      ... }
catch(Exception e){    e.printStackTrace(); }
finally{    // fermer si nécessaire  EntityManager JPA
              //  (si ouvert en début de cette méthode)
 }
}
```

4. Propagation du contexte transactionnel

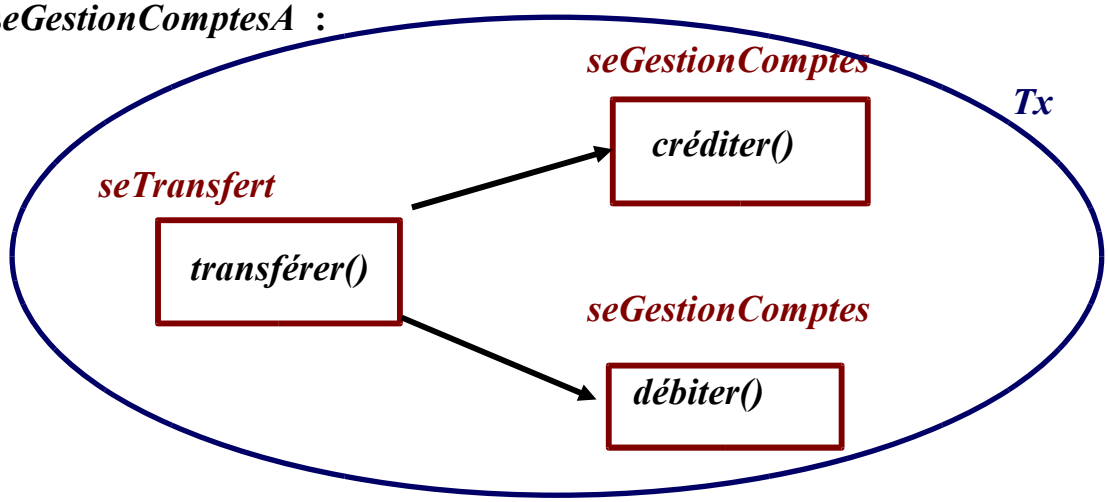
Contexte transactionnel & enrôlement des objets métiers

Exemple (début - initialisation):

Activation d'une méthode ***transférer()*** sur un EJB Session *seTransfert* . Cette méthode étant associée à un attribut déclaratif valant "**Required**" et parce qu'aucune transaction existe pour l'instant une nouvelle transaction *Tx* est alors automatiquement créée et le contexte associé englobe *seTransfert*.transferer()

Exemple transaction distribuée (suite - enrôlement):

Le code interne de la méthode *transférer()* active les méthodes *créditer(cpt)* et *débiter(cpt)* sur un ou plusieurs autre(s) EJB .
Si les méthodes *débiter()* et *créditer()* sont marquées via l'attribut déclaratif "**Required**" ou bien "**Supports**", alors **le contexte transactionnel s'agrandit automatiquement pour enrôler et englober les appels au niveau de l'EJB *seGestionComptesA* :**



5. Effets du contexte transactionnel sur les EJB

Transactions distribuées & EJB : Comportements

◆ **Une transaction automatiquement gérée par le serveur (conteneur d'EJB) est implicitement validée lorsque toutes ses parties se sont bien passées (sans exception).**

◆ Dès qu'une des opérations de la transaction génère une **exception système** héritant de ***RuntimeException*** (telle que **EJBException**) , l'ensemble de la transaction est alors automatiquement annulée.

◆ Si l'on souhaite **annuler explicitement une transaction** (suite à un test quelconque) il faut dans ce cas appeler la méthode **context.setRollbackOnly() .**

OK en version EJB2 , à vérifier avec EJB3:

- **En cas d'annulation de la transaction, les attributs internes (en mémoire) d'un Bean de type "Entité" sont automatiquement restaurés** (à leurs anciennes valeurs qu'il y avait dans la base de données).
- Par contre, **les attributs interne d'un Bean de type "Session à état" ne sont pas automatiquement restaurés** en cas d'annulation. Si l'on veut pouvoir gérer soit même cette tâche, il suffit de programmer les fonctions suivante de l'interface **SessionSynchronization**:

```
afterBegin()
        { //mémorisation valeurs }
afterCompletion(boolean committed)
        { if(committed==false) // Restitution }.
```

5.1. Annulation implicite d'une transaction en cas d'exception

Si une quelconque des méthodes (ou sous méthodes) d'un ejb remonte une exception de type "unchecked" (héritant de RuntimeException et à try/catch facultatif) , alors la transaction sera alors automatiquement annulée.
NB: toutes des exceptions de JPA et des EJB2 héritent de RuntimeException.

Cette façon d'annuler une transaction est la plus conseillée car elle permet en outre de remonter un message d'erreur significatif .

5.2. Annulation explicite d'une transaction :

exemple:

```
 @Resource private SessionContext context;
...
public void transfertVersCompteEpargne(double montant) throws
   SoldeInsuffisantException  {
   try {
      if ((soldeCompteCourant - montant) < 0) {
         context.setRollbackOnly();
         throw new SoldeInsuffisantException();
      }
    soldeCompteCourant -= montant;  soldeCompteEpargne += montant;
    majCompteCourant(soldeCompteCourant); majCompteEpargne(soldeCompteEpargne);
   } catch (SQLException ex) {
       throw new EJBException("Transaction failed due to SQLException: "
                              + ex.getMessage());
   }
}
```

6. Gestion déclarative des transactions / EJB3

6.1. Attributs transactionnels sur EJB (approche déclarative)

Attribut de transaction tx	Transaction en cours	Transaction au niveau du Bean
Required (par défaut)	none	T2
	T1	T1
RequiresNew	none	T2
	T1	T2
Mandatory	none	error
	T1	T1
NotSupported	none	none
	T1	none
Supports	none	none
	T1	T1
Never	none	none
	T1	error

6.2. Annotations sur EJB3 concernant les transactions

@TransactionManagement (**BEAN** or **CONTAINER**) , **default=CONTAINER**
à placer facultativement devant la classe d'un EJB session.

@TransactionAttribute (`MANDATORY` ou `REQUIRED` ou
`REQUIRES_NEW` ou `SUPPORTS` ou
`NOT_SUPPORTED` ou `NEVER`)
à placer facultativement devant les méthodes qui nécessitent un comportement transactionnel.

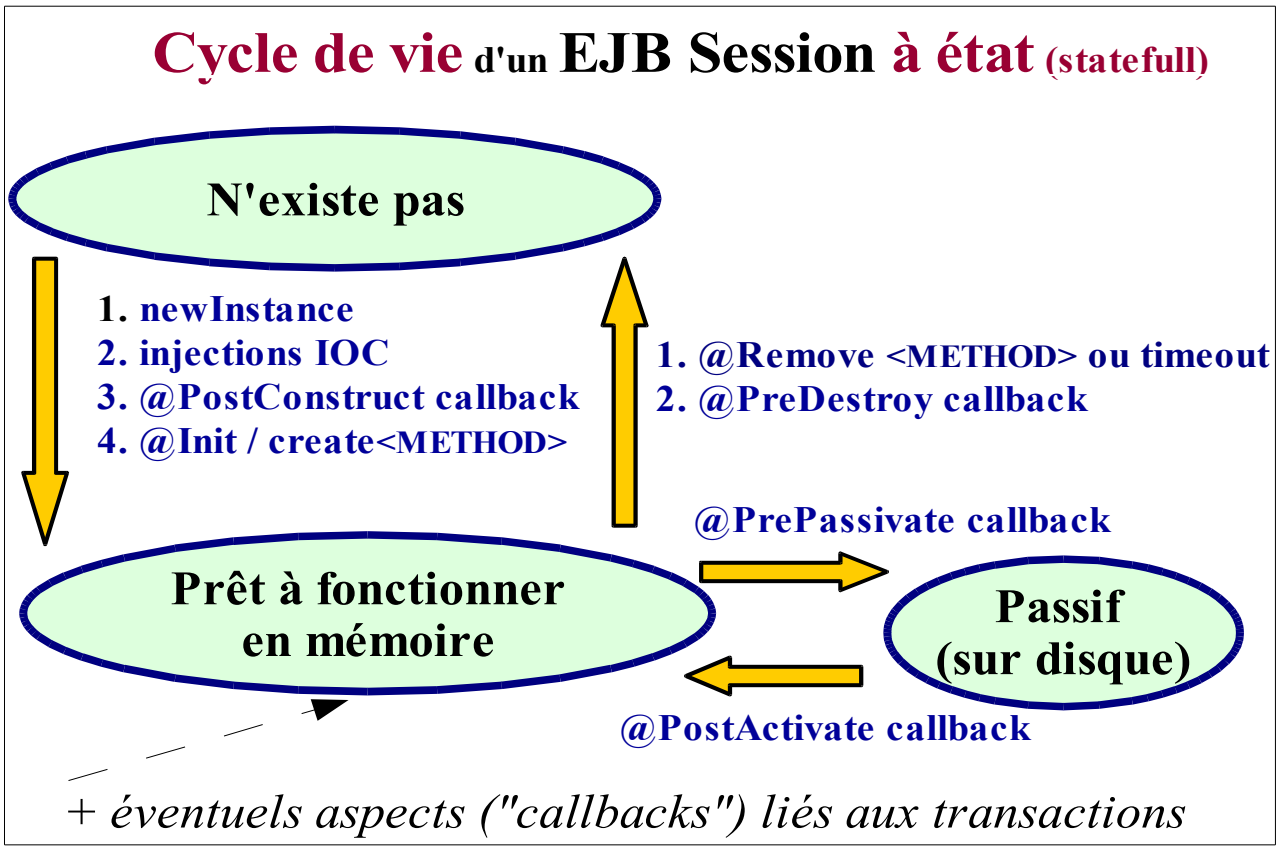
NB: Ces 2 annotations sont implicitement placées d'office avec les valeurs par défaut (CONTAINER , REQUIRED) qui conviennent très bien dans 95% des cas.
---> Les EJB3 fonctionnent donc en mode transactionnel même si rien n'est indiqué .

XII - EJB session à état (Stateful)

1. EJB Session à état conversationnel (stateful)

1.1. cycle de vie d'un EJB3 session à état

Si est à état , un EJB session comporte certains attributs internes dont les valeurs initialisées par certaines méthodes sont utilisées plus tard par d'autres méthodes. Si un tel composant n'est pas utilisé pendant un long moment , le container EJB peut alors décider de rendre celui-ci **passif** (ses données internes sont alors **automatiquement sérialisées et stockées sur disque** de façon à pouvoir être ultérieurement remontées en mémoire lors de l'activation).



NB: le timeout a aussi un effet sur les objets à l'état passif . Ceux ci sont alors supprimés en cas de très longue inactivité (@PreDestroy callback déclenchée si elle existe).

NB: les **injections IOC** et les **méthodes "callbacks" sont facultatives** .

Données classiquements stockées au sein d'un EJB session à état :

- id / username
- préférences d'un utilisateur (profil , ...)
- éléments sélectionnés (panier, ...)
- ...

NB: Lorqu'un EJB session à état est utilisé depuis la partie WEB d'une application J2EE , **une référence sur l'EJB session lié à l'utilisateur courant doit normalement être stockée au sein d'une session HTTP** .

Remarque importante:
Lorsqu' associé à un **EJB3 session à état** (***stateful***) ,
chaque appel à **lookup**() conduit à la création d'une nouvelle instance.
Il en va de même lors de l'établissement/initialisation d'une injection IOC .

exemple:

```
@Stateful
public class ShoppingCartBean implements ShoppingCart {
...
private String customer;
public void startToShop(String customer) { this.customer = customer; ... }
public void addToCart(Item item) {...}
...
@PreDestroy
void endShoppingCart() {...};
}
```

1.2. Initialisation et terminaison explicites (@Init et @Remove)

L'annotation **@Remove** peut être utilisée devant une méthode d'un EJB3 session à état .
Cette annotation demande au conteneur d'EJB3 de **supprimer automatiquement l'EJB juste après l'exécution (normale ou pas) de la méthode**.

exemple:

```
@Stateful
public class ShoppingCartBean implements ShoppingCart {

@Init
public void initialisation(){
//méthode appelée automatiquement (pour éventuellement initialiser une nouvelle instance)
//cette méthode est appelée après l'éventuelle callback de type @PostConstruct
}
....
@Remove
public void finishShopping() {...}
}
```

XIII - EJB "M.D.B." et invocation asynchrone

1. Présentation de JMS

JMS *(Java Message Service)*

JMS est une **API** permettant de faire **dialoguer des applications** de façon **asynchrone** .
Architecture associée: **MOM** (**M**essage **O**riented **M**iddleWare).

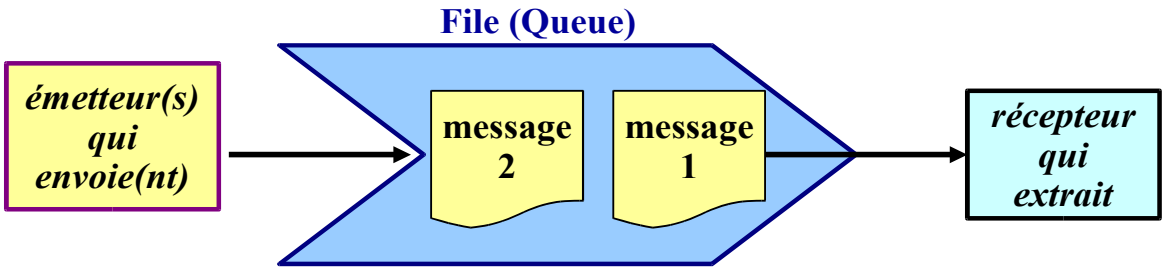
NB: ***JMS n'est qu'une API qui sert à accéder*** à un ***véritable fournisseur de Files de messages*** (ex: **MQSeries/Websphere_MQ** d'IBM , **ActiveMQ** d'apache, ...)

Dans la terminologie JMS, les Clients JMS sont des programmes Java qui envoient et reçoivent des messages dans/depuis une file (**message queue**).

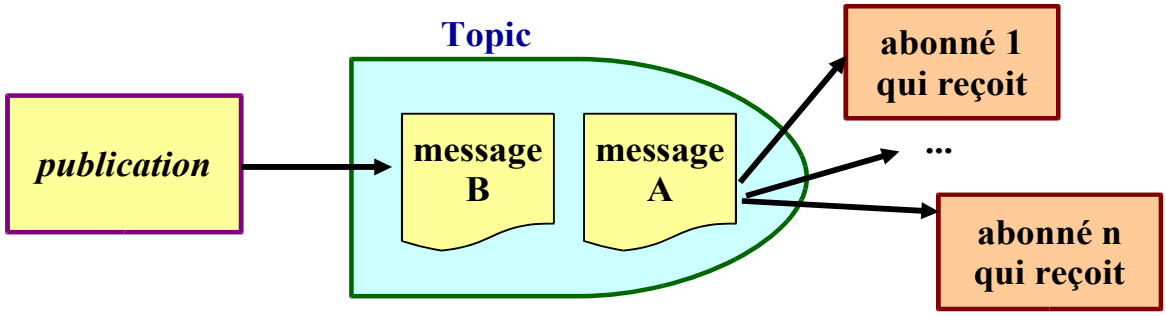
Une file de message sera gérée par un "Provider JMS" .
Les clients utiliseront **JNDI** pour accéder à une file.
L'objet **ConnectionFactory** sera utilisé pour établir une connexion avec une file.
L'objet **Destination** (*File* ou *Topic*) sert à préciser la destination d'un message que l'on envoi ou bien la source d'un message que l'on souhaite récupérer.

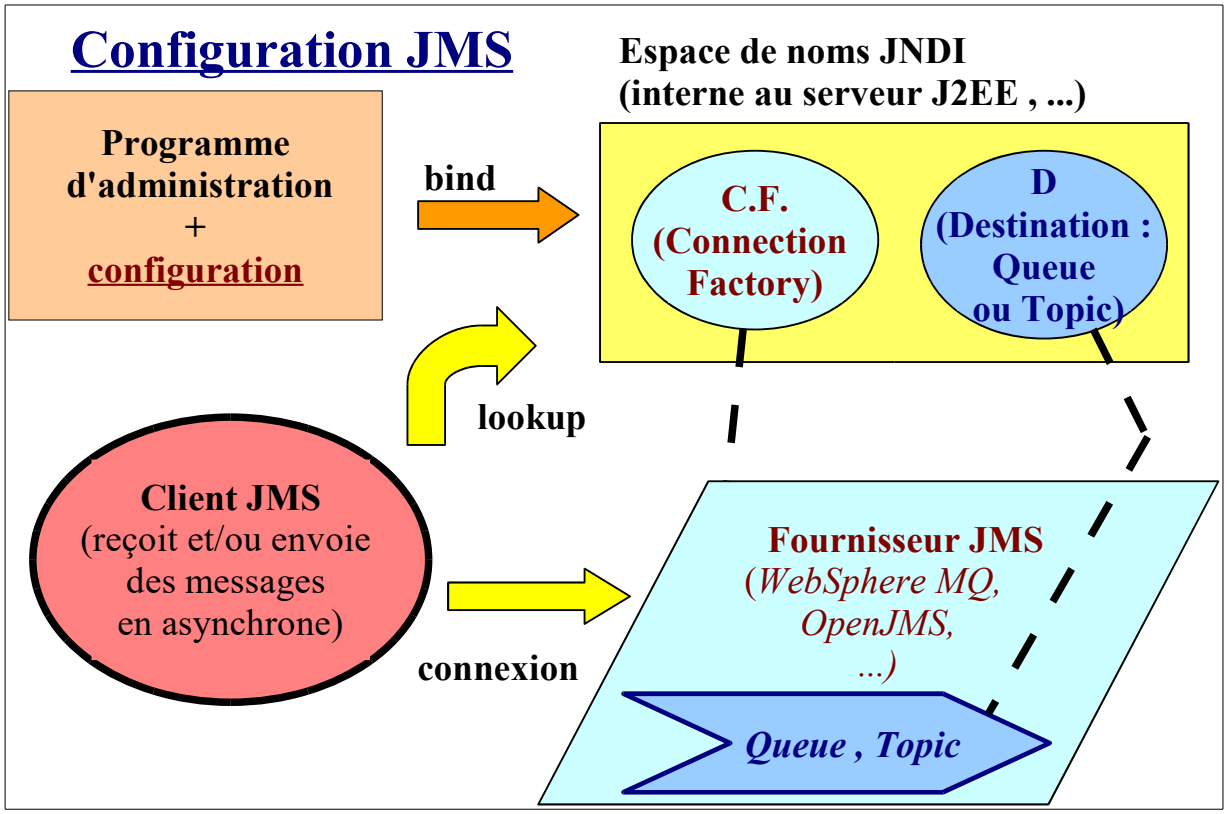
JMS permet de mettre en oeuvre les 2 modèles suivants:
- **PTP** (Point To Point)
- **Pub/Sub** (Published & Subscribe) **.../...**

JMS Queue : *Point To Point*

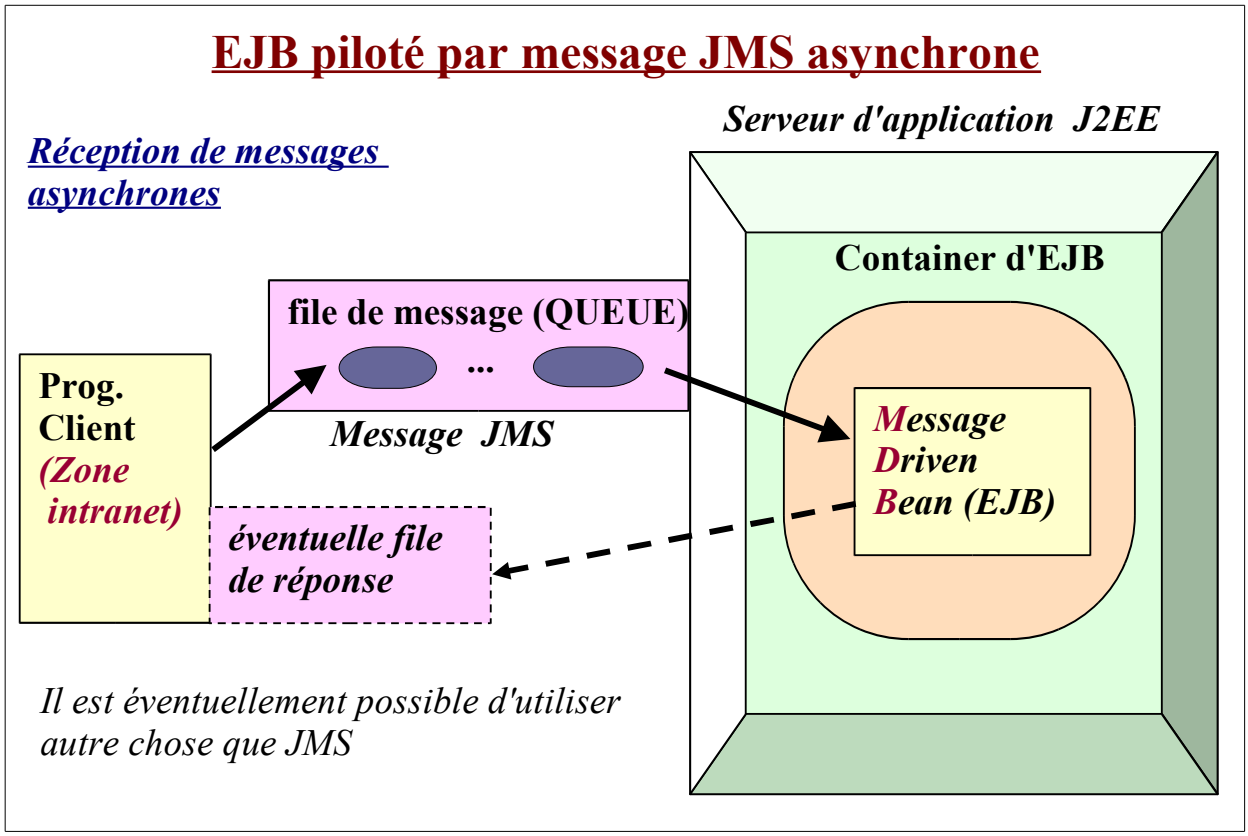


JMS Topic : *Publish / Subscribe*

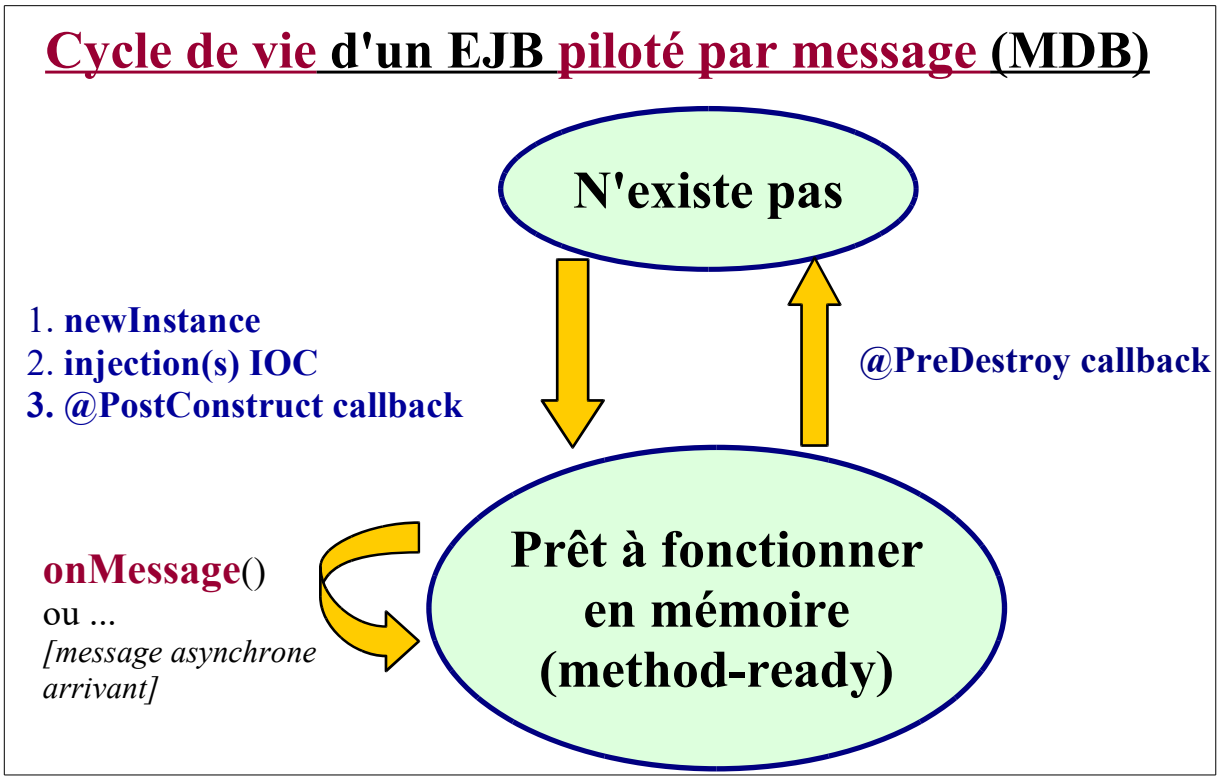




2. EJB3 de type MDB



2.1. cycle de vie d'un EJB3 MDB



Le conteneur d'EJB maintient généralement un **pool** interne d'EJB3 "mdb" pour atteindre de bonnes performances.

2.2. Annotations et interfaces pour EJB3 de type MDB

```
@MessageDriven(activationConfig =
{
@ActivationConfigProperty(propertyName="destinationType",
                          propertyValue="javax.jms.Queue"),
@ActivationConfigProperty(propertyName="destination",
                          propertyValue="queue/xxx")
})
public class MessageDrivenXxx implements  javax.jms.MessageListener
                                                    {
...
}
```

2.3. Exemple d' EJB3 de type MDB

myejb.MessageDrivenCalculator

```
package myejb;
import javax.annotation.PostConstruct;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.EJB;
```

```
import javax.ejb.MessageDriven;
import javax.jms.JMSException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.MessageListener;

@MessageDriven(activationConfig =
{
@ActivationConfigProperty(propertyName="destinationType",
                          propertyValue="javax.jms.Queue"),
@ActivationConfigProperty(propertyName="destination", propertyValue="queue/calculator")
})
public class MessageDrivenCalculator implements MessageListener
{
        @EJB(name="CalculatorBean")
        private Calculator calculateur = null;

        @PostConstruct
        void verifInjection() {
                if(calculateur == null)  System.err.println("Mauvaise injection IOC");
        }

        public void onMessage(Message recvMsg)
          {
            System.out.println("Received message:" + recvMsg.toString());
            if(recvMsg instanceof MapMessage)
            {
                int res=0;
                try {
                    if(calculateur!=null)  {
                        MapMessage mm=(MapMessage) recvMsg;
                          if(mm.getString("op").equals("+"))
                              res= calculateur.add(mm.getInt("a"),mm.getInt("b"));
                    } else System.err.println("calculateur is null (IOC)");
                    } catch (JMSException e) {   e.printStackTrace();   }
                System.out.println("res="+res);
            }
          }
}
```

2.4. client (externe) de test (envoyant un message dans une file)

(avec ici une configuration pour Jboss 7)

```
public static void main(String[] args) {
   try {
        Properties jndiProps = new Properties();
        jndiProps.put(Context.INITIAL_CONTEXT_FACTORY,
                        "org.jboss.naming.remote.client.InitialContextFactory");
        jndiProps.put(Context.PROVIDER_URL, "remote://localhost:4447");
                                    //  "http-remoting://localhost:8080" pour Jboss Wildfly 9.2
```

```
        jndiProps.put(Context.SECURITY_PRINCIPAL, "admin");
        jndiProps.put(Context.SECURITY_CREDENTIALS, "pwd");
        //avec admin = utilisateur ajouté via la commande JBOSS7_HOME/bin/add-user
        //mot de passe=pwd et rôles associés admin,guest
        //et avec "guest" = rôle configuré sur la partie "messaging" de standalone(-full).xml

        // jndiProps.put("jboss.naming.client.ejb.context", true); //si besoin @Remote en plus
        Context ic = new InitialContext(jndiProps);

        QueueConnectionFactory factory = (QueueConnectionFactory)
                                                ic.lookup("jms/RemoteConnectionFactory");
        //avec <entry name="java:jboss/exported/jms/RemoteConnectionFactory"/>
         //dans standalone(-full).xml

        Queue queue = (Queue) ic.lookup("jms/queue/test");
        // avec queue/test doit etre exporté dans standalone(-full).xml
        //<entry name="java:jboss/exported/jms/queue/test"/>

        QueueConnection cnn = factory.createQueueConnection(
                        jndiProps.getProperty(Context.SECURITY_PRINCIPAL),
                        jndiProps.getProperty(Context.SECURITY_CREDENTIALS));
        QueueSession session = cnn.createQueueSession(false,
                                                QueueSession.AUTO_ACKNOWLEDGE);

        TextMessage msg = session.createTextMessage();
        msg.setText("<msg>message in the bottle</msg>");
        QueueSender sender = session.createSender(queue);
         sender.send(msg);
        System.out.println("Message sent successfully to remote queue.");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

NB1: Les connexions JNDI et JMS sont sécurisées avec Jboss 7 et doivent être associées à une authentification d'un utilisateur configuré du serveur Jboss7 ayant le rôle "guest" (ou …) lui même configuré dans standalone(-full).xml
La commande interactive **add-user** de JBOSS7_HOME/bin permet d'ajouter un utilisateur avec un mot de passe crypté dans standalone/configuration/**application-users.properties** et avec une liste de rôles en clair dans **application-roles.properties** .

NB2 : Les noms JNDI qui sont vus depuis les applications externes sont ceux qui sont préfixés par "java:jboss/exported" dans standalone(-full).xml

Autre exemple partiel :

```
Queue queue = (Queue) ctx.lookup("jms/queue/myQueue");

    cnn = factory.createQueueConnection();
    session = cnn.createQueueSession(false, QueueSession.AUTO_ACKNOWLEDGE);

    MapMessage msg = session.createMapMessage();
```

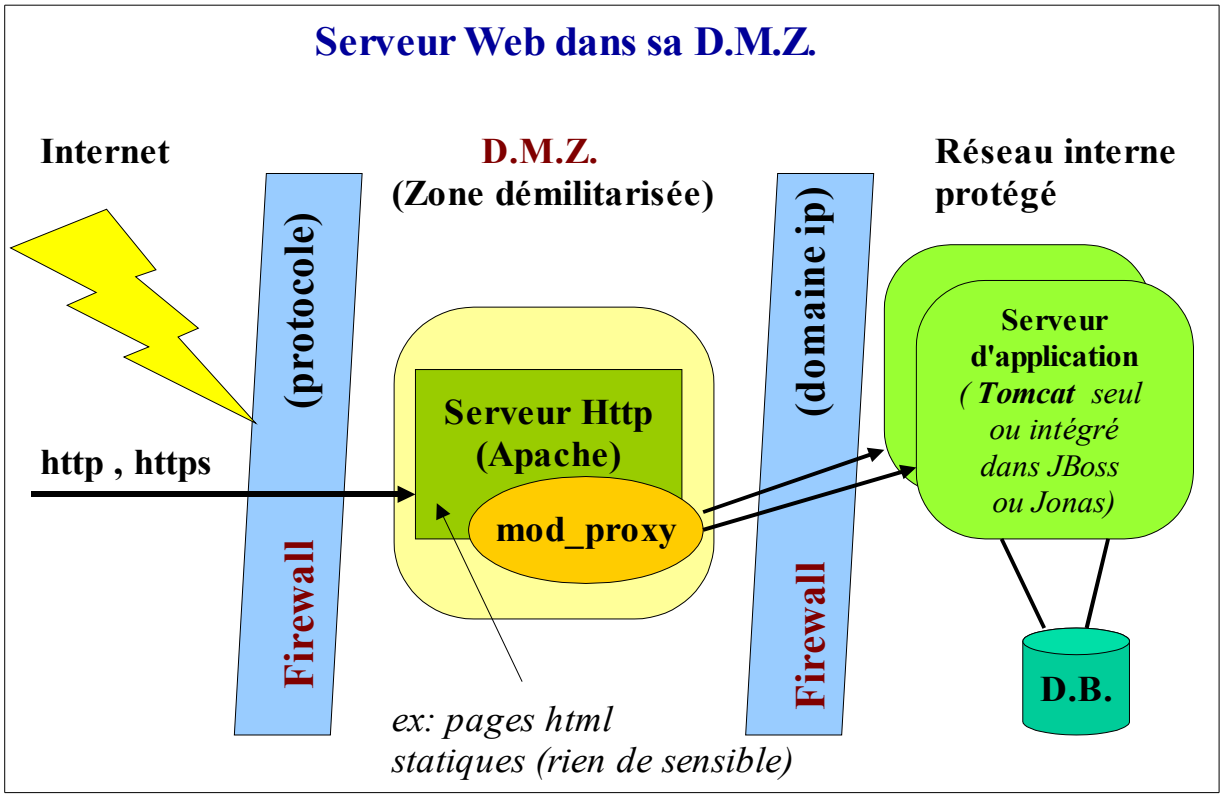
```
msg.setInt("a",5);
msg.setString("op","+");
msg.setInt("b",6);

sender = session.createSender(queue);
sender.send(msg);
```

Dans configuration/**standalone-full.xml** :

```
<permission type="send" roles="guest"/>          < !-- déjà configuré -->
 <permission type="consume" roles="guest"/>
  ….
 <jms-destinations>
     ...
      <jms-queue name="myQueue">
         <entry name="java:jboss/exported/jms/queue/myQueue"/>
      </jms-queue>

      <jms-queue name="test">
         <entry name="java:jboss/exported/jms/queue/test"/>   < !-- à ajouter -->
      </jms-queue>

   </jms-destinations>
```

XIV - Sécurité JEE (au niveau des EJB)

1. D.M.Z. et Firewalls



2. Sécurité J2EE/JEE5

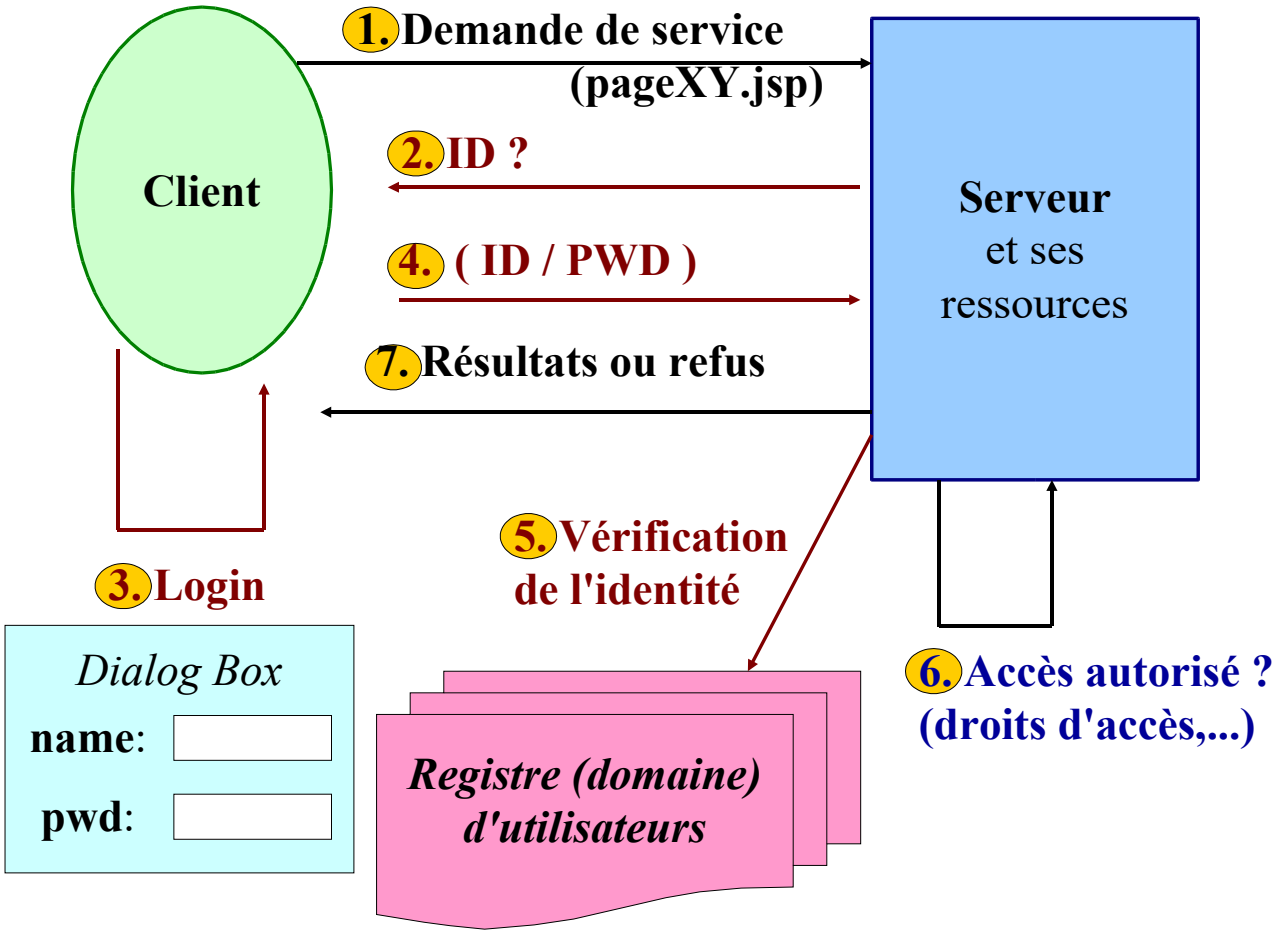
Sécurité J2EE/JEE5

Gérer la sécurité "J2EE/JEE5" consiste essentiellement à :

- **Préciser le ou les rôle(s) logique(s) requis pour pouvoir déclencher une certaine méthode sur un EJB ou pour activer certaines URL d'une application Web.**
  ➔ *Travail généralement effectué par le développeur*

- **Authentifier l'utilisateur** (UserName , PassWord).
  ➔ *Via technologie **HTTPS** ou **JAAS** ou ...*

- **Associer un ou plusieurs utilisateur(s) ou groupe(s) [d'un annuaire ldap ou ...] à chaque rôle logique**.
  ➔ *Paramétrage et configuration liés au déploiement d'une application J2EE et à l'administration du serveur*


Client
1. Demande de service
(pageXY.jsp)
2. ID ?
4. (ID / PWD)
7. Résultats ou refus
Serveur
et ses
ressources
3. Login
Dialog Box
name:
pwd:
5. Vérification
de l'identité
Registre (domaine)
d'utilisateurs
6. Accès autorisé ?
(droits d'accès,...)

3. Rôles associés aux EJB3 (via annotations)

Les annotations **@DeclareRoles** , **@RoleAlowed** et **@PermitAll** (de *javax.annotation.security*) sont applicables sur les EJB3.

Elles permettent de paramétrer les besoins en terme contrôle d'accès sur les méthodes d'un EJB.

@DeclareRoles	**permet de déclarer une liste de rôles existants (et à prendre en compte)**
@RoleAlowed	**permet d'indiquer une liste de rôles dont au moins un est requis.**
@PermitAll	**permet d'indiquer un accès libre (sans aucun rôle requis)**

NB: En théorie, une annotation attachée à une méthode précise est prioritaire sur une annotation (de même nom) globalement attachée à l'ensemble de la classe.

exemple:

```
package …..............;

import javax.annotation.security.DeclareRoles;
import javax.annotation.security.PermitAll;
import javax.annotation.security.RolesAllowed;
import javax.ejb.Remote;
import javax.ejb.Stateless;

@Stateless
// @Remote sur l'interface CalculatorRemote
@DeclareRoles({"student","teacher"})
public class CalculatorBean implements CalculatorRemote
{
  @PermitAll
  public int add(int x, int y) { // accessible à tout le monde
        System.out.println("add : caller identity: "+    ctx.getCallerPrincipal().toString());
          if(ctx.isCallerInRole("student")) System.out.println("role=student");
          if(ctx.isCallerInRole("teacher")) System.out.println("role=teacher");
    return x + y;
  }

  @RolesAllowed({"student","teacher"})
  public int substract(int x, int y)    {
    return x - y;
  }

  @RolesAllowed({"teacher"})
  public int divide(int x, int y)   {
    return x / y;
  }
}
```

4. Domaine de sécurité (Jboss)

Pour effectuer des tests en mode développement, il faut en outre:

- configurer des domaines de sécurité (Realm) au niveau du serveur JEE6 ou 7 .
- associer quelques utilisateurs (ou groupe d'utilisateurs) fictifs aux rôles de l'application.
- coder ou paramétrer une authentification du coté client.

Dans le cas du serveur open source JBoss >=7 , la configuration d'un domaine de sécurité s'effectue essentiellement en étudiant (et en ajustant si besoin) le sous-système "**urn:jboss:domain:security:1.1**" du fichier configuration/**standalone.xml** .
Voici la configuration par défaut:

```
<subsystem xmlns="urn:jboss:domain:security:1.1">
        <security-domains>
          <security-domain name="other" cache-type="default">
            <authentication>
              <login-module code="Remoting" flag="optional">
                <module-option name="password-stacking" value="useFirstPass"/>
              </login-module>
              <login-module code="RealmUsersRoles" flag="required">
                <module-option name="usersProperties"
                        value="${jboss.server.config.dir}/application-users.properties"/>
                <module-option name="rolesProperties"
                        value="${jboss.server.config.dir}/application-roles.properties"/>
                <module-option name="realm" value="ApplicationRealm"/>
                <module-option name="password-stacking" value="useFirstPass"/>
              </login-module>
            </authentication>
          </security-domain>
          <security-domain name="jboss-web-policy" cache-type="default">
            <authorization>
              <policy-module code="Delegating" flag="required"/>
            </authorization>
          </security-domain>
          <security-domain name="jboss-ejb-policy" cache-type="default">
            <authorization>
              <policy-module code="Delegating" flag="required"/>
            </authorization>
          </security-domain>
        </security-domains>
      </subsystem>
```

Le référencement de ce domaine de sécurité Jboss s'effectue au niveau de l'application JEE6 en ajoutant un bloc xml ***<s:security-domain>*** dans le fichier **META-INF/jboss-ejb3.xml** du module d'EJB de l'application:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss:jboss  xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
      xmlns:s="urn:security:1.1"
      version="3.1" impl-version="2.0">

   <assembly-descriptor>
      <s:security>
         <!-- Even wildcard * is supported -->
         <ejb-name>GestionDevisesImpl</ejb-name>
        <!-- <ejb-name>*</ejb-name> * demande à tout sécuriser (même les EJB de type DAO)-->
         <!-- Name of the security domain which is configured in the EJB3 subsystem -->
         <s:security-domain>other</s:security-domain>
      </s:security>
   </assembly-descriptor>
</jboss:jboss>
```

Conformément au paramétrage du domaine de sécurité "***other***" déclaré dans ***standalone.xml*** les **utilisateurs (avec leurs mots de passe) et les rôles associés** doivent être renseignés via la commande bin/**add-user** (.bat ou .sh) et seront stockés dans les fichiers suivants (recherchés dans JBOSS7_HOME/**standalone/configuration**) :

application-users.properties

```
#utilisateur=mot_de_passe_crypté_via_commande_add_user
#format exact : username=HEX( MD5( username ':' realm ':' password))
s1=s1pwd_crypte
s2=s2pwd_crypte
t1=t1pwd_crypte
admin=2a0923285184943425d1f53ddd58ec7a
```

application-roles.properties

```
#utilisateur=liste_de_roles
s1=student,guest
s2=student,guest
t1=teacher,admin,guest
admin=admin,guest
```

NB: En mode production on peut évidemment ***faire mieux*** en:

- **configurant un domaine de sécurité basé sur un annuaire LDAP (et non pas sur des fichiers ".properties" de l'application). *[==> approfondir l'administration d'un serveur JEE].***

5. Tests

5.1. Via une authentification coté "Web"

Si la sécurité JEE est également bien paramétrée coté WEB (security-constraint , login-config , ... dans WEB-INF/web.xml) et (selon la version du serveur Jboss) si **WEB-INF/jboss-web.xml** comporte aussi

```
<security-domain>other   ou   ...</security-domain>
```

alors les informations d'authentification récupérées par le conteneur web sont automatiquement repassées au conteneur d'EJB .

5.2. Depuis un client externe (via Jndi et/ou Jaas)

```
…
Properties props = new java.util.Properties();
props.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory");
// props.put(Context.PROVIDER_URL, "http-remoting://localhost:8080"); // for wildfly 9
props.put(Context.PROVIDER_URL, "remote://localhost:4447"); //for Jboss7.1
props.put(Context.SECURITY_PRINCIPAL, "s1");//"admin" , "guest" , "..."
props.put(Context.SECURITY_CREDENTIALS, "s1pwd"); //"pwd", "guest007"
props.put("jboss.naming.client.ejb.context", true);
Context jndiContext = new InitialContext(props);
….
ejbCalculator = (CalculatorRemote) jndiContext.lookup("my-jee-app/my-jee-app-ejb-
impl/CalculatorBean!tp.myapp.ejb.itf.CalculatorRemote");
….
res = ejbCalculator.substract(8,6) ; // autorisé pour rôle = "student"
...
res = ejbCalculator.divide(9,3) ; // refusé pour rôle = "student"
….
```

XV - Aspects divers (timer, aop, ...)

1. Timer sur EJB3.0 (déclenchement différé)

```
package myejb;

public interface ExampleTimer {
        public void scheduleTimer(long milliseconds);
}
```

```
package myejb;

import java.util.Date;
import javax.annotation.Resource;
import javax.ejb.Remote;
import javax.ejb.SessionContext;
import javax.ejb.Stateless;
import javax.ejb.Timeout;
import javax.ejb.Timer;

@Stateless
@Remote(ExampleTimer.class)
public class ExampleTimerBean implements ExampleTimer
{
 @Resource
  private  SessionContext ctx;

  public void scheduleTimer(long milliseconds)  {
    ctx.getTimerService().createTimer(new Date(new Date().getTime() + milliseconds),
                                      "Hello World");
  }

  @Timeout
  public void timeoutHandler(Timer timer)  {
    System.out.println("--------------------");
    System.out.println("* Received Timer event: " + timer.getInfo());
    System.out.println("--------------------");
    timer.cancel();
  }
}
```

```
public static void test_timerEjb3() throws Exception  {
          InitialContext ctx = new InitialContext(this.props);
          ExampleTimer timerEjb = (ExampleTimer)
                                      ctx.lookup("myJeeApp/ExampleTimerBean/remote");
          timerEjb.scheduleTimer(3000);
  }
```

Autre exemple :

```
@PostConstruct
        public void init(){
                long duration=10;//la premiere fois apres 10 ms
                long interval=1000*60*5;
                context.getTimerService().createTimer(duration,interval,
                        "sur XyImpl toutes_les_5minutes_tant_que_pas_cancel");
        }

@Timeout
        public void timeOutHandler(Timer timer){
          System.out.println("timeoutHandler (XyImpl) : " + timer.getInfo());
          // timer.cancel();//pour arrêter (si periodique avec interval)
        }
```

2. Nouveaux timers simplifiés depuis EJB 3.1

```
@Schedule(second="0,30", minute="*", hour="*")
public void automaticPeriodicExecution(Timer automaticTimer) {
        System.out.println("Executing (toutes les 30s) ...");
                System.out.println("Execution Time : " + new Date());
                System.out.println("_____________________________________________");
        }
```

3. EJB "Singleton" depuis JEE6 / EJB 3.1

```
//@Stateless
@Singleton //comme @Stateless mais toujours un seul ,
                // ce qui est pratique pour placer des données partagées dedans
public class XyImpl implements XyLocal , XyRemote {
…
}
```

4. Intercepteurs pour EJB3 / extension AOP

On peut associer un ou plusieurs **intercepteurs** à un EJB de façon à ce que **certains traitements techniques** (**aspects** secondaires de type "*log*" , "*mesures*" , ...) soient **automatiquement ajoutés et déclenchés lors d'un appel à une méthode de l'EJB** .

Exemple: EJB (avec intercepteur(s) associé(s)):

```
@Stateless
@Interceptors({xxx.yyy.Metrics.class , xxx.yyy.EventuelAutreIntercepteur.class})
public class AccountManagementBean implements AccountManagement {
        public void createAccount(int accountNumber, AccountDetails details) { ... }
        public void deleteAccount(int accountNumber) { ... }
        public void activateAccount(int accountNumber) { ... }
        public void deactivateAccount(int accountNumber) { ... }            ...
}
```

Classe de l'intercepteur affichant le temps d'exécution des méthodes:

```
public class Metrics {

@AroundInvoke
public Object  profile(InvocationContext inv) throws Exception {
        long time = System.currentTimeMillis();
        try {
                return inv.proceed();
            } finally {
                 long endTime = time - System.currentTimeMillis();
                System.out.println(inv.getMethod() + " took " + endTime + "milliseconds.");
                }
       }
}
```

NB:
La signature d'une méthode annotée par "***@AroundInvoke***" doit être la suivante:

```
public Object <METHOD>(InvocationContext) throws Exception
```

L'interface prédéfinie ***javax.interceptor.InvocationContext*** permet d'obtenir des informations sur la méthode métier à enrichir :

```
public interface InvocationContext {
public Object getTarget();
public Method getMethod();
public Object[] getParameters();
public void setParameters(Object[] params);
public java.util.Map<String, Object> getContextData();
public Object proceed() throws Exception;
}
```

ANNEXES

XVI - Essentiel CDI

1. CDI: Context and Dependencies Injections (>= JEE6)

CDI *(alias JSR 299 alias **WebBean**)* constitue l'**un des principaux apports de JEE6** vis à vis de JEE5.

Les spécifications "**CDI**" visent à mettre en œuvre des injections de dépendances automatiques entre des composants (de conteneurs éventuellement différents).
Un "EJB 3.1 session" pourra par exemple être injecté dans un "managedBean" de JSF .

Attention: CDI (alias JSR-299) utilise en interne DI / JSR-330 (@Inject et @Named) mais va plus loin en apportant bien plus de fonctionnalités (dans un environnement JEE6) .

Les fonctionnalités additionnelles de CDI par rapport à DI sont essentiellement :

- la prise en charge des contextes (associés aux scopes "request", "session" , "application" et "conversation")
- l'association automatique "EJB session stateful" avec session Http .
- des injections transparentes entre EJB et JavaBean web (jsf,...) .
- prise en charge de certains événements
- prise en charge des intercepteurs (décorateurs/aop) .
- une approche fortement typée et avec un très faible couplage

En un mot les spécifications CDI / WebBeans tentent d'unifier harmonieusement le monde des EJB 3.x (gérant les transactions et la persistance) avec le monde du Web (Servlet/JSP/JSF) .

On reconnaît l'utilisation de CDI à la présence d'un fichier "**beans.xml**" (éventuellement vide mais obligatoire) dans les répertoires **WEB-INF** (web) ou **META-INF** (ejb) .

Pour l'instant, les deux principales implémentations des spécifications CDI sont :

- **Weld** (de Jboss)
- **OpenWebBeans** (de Apache)

Beaucoup de principes de CDI/JSR-299 proviennent du framework Seam (V1,V2) de Jboss .

Maintenant que CDI/JSR-299 est normalisé au niveau de JEE6 , **Seam** V3 s'intègre parfaitement dans JEE6 en utilisant à la lettre CDI/JSR-299 (et n'est plus à considérer comme un framework propriétaire mais comme une **extension portable pour JEE6**) .

1.1. Le sous ensemble JSR-330 (DI)

JSR-330 (**D**ependencies **I**njections) [API packagée dant ***java-inject.jar***] est essentiellement constituée de deux annotations fondamentales normalisées :

- **@Named** (pour identifier/nommer ce qui pourra être injecté)
- **@Inject** (pour effectuer une injection de dépendance)

et de quelques ***méta-annotations*** qualificatives (**@Qualifier , @Scope , …**)

Ces annotations ont le mérite de constituer les bases d'un **standard** assez universel dans le monde java récent (***@Named et @Inject sont utilisables au niveau de Spring 3 , de Seam 3 et de JEE6***).

….

1.2. Anatomies des liaisons/injections (JSR 299)

Les injections automatiques seront paramétrées par une série de critères qui devront être mis en concordance entre :

- un élément/composant potentiellement injectable
- une référence sur une dépendance à injecter.

Critères influençant les liaisons par injections :

Api type : type java (souvent une Interface , éventuellement une classe)

Qualifier (qualificatif): Annotation spécifique (définie par l'utilisateur/développeur) (ex : @Variante1 , @Variante2) et elle même annotée par la **méta-annotations @Qualifie**r .

Les **"qualificatifs" par défaut** sont **@Any** (du coté composant injectable) et **@Default @Any**(du coté référence à initialiser) .

Alternative de déploiement (une ou plusieurs ***@Alternative*** dans le code) et alternative à utiliser (à la place de @Default) déclarée au sein de ***<alternatives>*** dans *beans.xml*.

Portée/Scope (@RequestScope, @SessionScope , ….) sachant qu'un composant d'une portée globale/longue_durée_de_vie peut être injecté dans un composant d'un scope plus étroit/plus éphémère et non l'inverse .

Correspondance de nom (entre @Named() de web bean et EL in JSF2 , JSP2)

@Named(***"webBeanXY"***) <---> ***#{webBeanXY***.ppp**}**

Rappel : le nom donné par défaut par @Named à un composant WebBean est le nom de la classe java (en remplacement le premier caractère par une minuscule).

1.3. Qualifier

On peut définir de nouveaux qualificatifs (personnalisés) en créant de nouvelles annotations elles mêmes basées sur la méta-annotation @Qualifier.

Exemple :

```
package tp.myapp.web.cdi.qualifier;

import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.PARAMETER;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import javax.inject.Qualifier;

@Retention(RetentionPolicy.RUNTIME)
@Target({TYPE,METHOD,FIELD,PARAMETER})
@Qualifier
public @interface English { }
```

et

```
@Retention(RetentionPolicy.RUNTIME)
@Target({TYPE,METHOD,FIELD,PARAMETER})
@Qualifier
public @interface French {}
```

utilisation possible :

```
public interface MessageGenerator {
	public String messageFromInput(String input);
}
```

```
@Named //javax.inject
@ApplicationScoped //javax.enterprise.context
@English
//@Default //javax.enterprise.inject
public class MsgGenV1 implements MessageGenerator {

	public String messageFromInput(String input) {
		return "hello " + input;
	}
}
```

```
@Named //javax.inject
@ApplicationScoped //javax.enterprise.context
@French
//@Any //(by default)
public class MsgGenV2 implements MessageGenerator {

	public String messageFromInput(String input) {
		return "bonjour " + input;
	}
}
```

```
@Named
@SessionScoped
public class MySessionBean implements Serializable{

	private static final long serialVersionUID = 1L;

	private String name; //+get/set
	private String message; //+get/set

	@Inject
	//@English
	@French
	//@Default by default
	private MessageGenerator msgGen;

	public String doAction(){
		message = msgGen.messageFromInput(name);
		System.out.println("message: "+message);
		return null;
	}
}
```

- Si le qualificatif @French est mentionné , la version française (MsgGenV2) est injectée/utilisée.
- Si le qualificatif @English est mentionné , la version anglaise (MsgGenV1) est injectée/utilisée.
- Si aucun qualificatif n'est mentionné , une exception est levée dès le démarrage de (en cas d'ambiguïté) .
- Si la version anglaise est doublement qualifiée (@English et **@Default**) et que la version française reste simplement qualifiée , il n'y plus d'ambiguïté , c'est la version par @Default qui sera injectée/utilisée .
- @Default et @Any sont prédéfinies dans le package `javax.enterprise.inject`

1.4. @Produces (pour méthode de production d'instances à injecter)

```
//@ Named  n'est pas indispensable
//@ ApplicationScoped  n'est pas indispensable
public class MsgGenAutomaticFactory {
	private Random r = new Random();

	@Inject @French
	private MessageGenerator msgGenFr;

	@Inject @English
	private MessageGenerator msgGenEn;

	@Produces @Default
	public MessageGenerator getMsgGen(){
```

```
            MessageGenerator msgGen=null;
            int n = r.nextInt() % 2;
            msgGen = (n==0) ? msgGenEn : msgGenFr;
            System.out.println("msgGen build by producer: "
                              + msgGen.getClass().getSimpleName());
            return msgGen;
      }
}
```

Utilisation possible:

```
@SessionScoped // or @ RequestScoped
public class MyManagedBean implements Serializable{
      ...
      @Inject
      //@Default by default
      private MessageGenerator msgGen;

      public String doAction(){
            message = msgGen.messageFromInput(name);
            return null;
      }
}
```

NB: selon que l'injection soit effectué dans un bean de scope @RequestScoped ou @SessionScoped , la méthode de production sera invoquée une ou plusieurs fois.

La fabrication/production de l'instance peut prendre tout un tas de formes :

- statique (en masquant les indirections ou l'utilisation d'une fabrique ordinaire)
- pré-fabriquée (dans pool ou ...)
- selon alternative (au runtime)
- selon requête dynamique (entityManager.createQuery("....").getSingleResult())
- ….

1.5. Scopes (prédéfinis et extensions)

Les annotations de types "Scope" prédéfinies (dans *javax.enterprise.context*) sont :

- **@RequestScoped**
- **@SessionScoped**
- **@ApplicationScoped**
- **@ConversationScoped** ***(spécifique JSF request <= conversation <= session)***
- **@Dependent** ***(par défaut , selon contexte du bean contenant la référence)***

Méta annotation "Scope" pour éventuellement définir de nouveaux scopes :

```
@Retention(RUNTIME)
@Target({TYPE, METHOD})
```

```
@Scope //de javax.inject
public @interface MyNewScope {}
```

Encore faut-il associer une certaine sémantique à ce nouveau scope .

==> Pour de futures interrogations/réflexions .

1.6. Alternatives (de remplacement , explicitées dans beans.xml)

```
public interface NewsGenerator {
	public String getLastNews();
}
```

```
@Named
@ApplicationScoped
@Default
public class NewsGen implements NewsGenerator {
	public String getLastNews() {
		return "fresh news";
	}
}
```

```
@Named
@ApplicationScoped
@Alternative
public class AlternativeNewsGen implements NewsGenerator {
	public String getLastNews() {
		return "alternative news";
	}
}
```

et selon WEB-INF/beans.xml

```
<!--  config file for CDI (JEE6 / JSR299) in WEB-INF or META-INF (ejb) -->
<beans xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">

	<alternatives>
		<class>....</class> <class>....</class>
	</alternatives>
</beans>
```

avec

```
 <alternatives>
		<!-- <class>tp.myapp.web.cdi.news.AlternativeNewsGen</class> -->
	</alternatives>
```

ou bien

```
 <alternatives>
		<class>tp.myapp.web.cdi.news.AlternativeNewsGen</class>
	</alternatives>
```

la version qui sera injectée/utilisée sera :

- NewsGen *(par défaut)*
- ou bien AlternativeNewsGen (en remplacement).

XVII - Essentiel RMI (Remote Method Invocation)

1. Principes "RPC" (Remote Procedure Call)

Objets distribués en mode synchrone
--> Appels de méthodes à distance

Problématiques:

- comment traverser le réseau ?

- les applications "client" et "serveur" ont des espaces mémoires différents.

- comment rendre les appels transparents ?

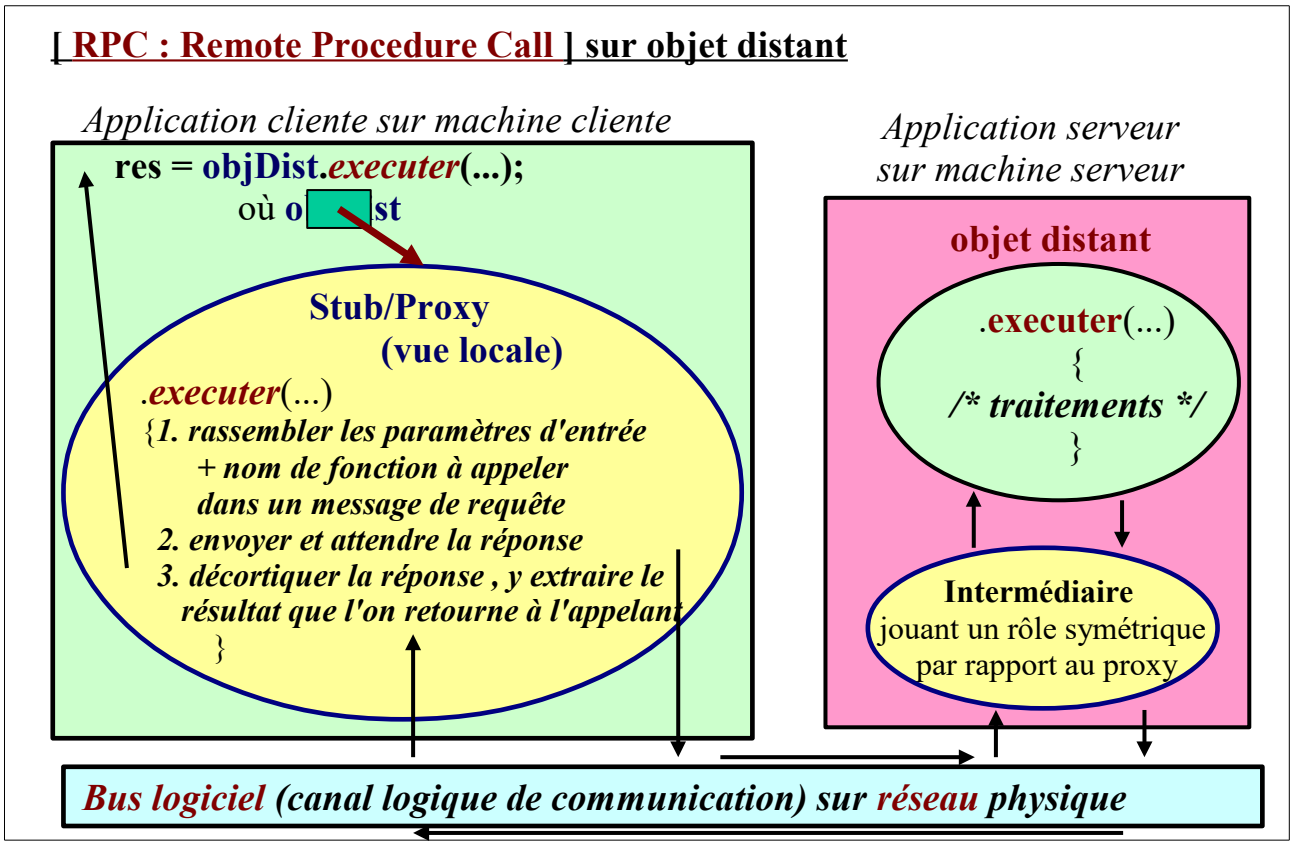
- comment établir une connexion simple indépendante de la localisation du serveur (adresse ip, n° port , ...) ?

Le réseau ,

.... passer par dessus .... ?
... passer par dessous .... ?

Impossible , il faut le traverser !

2. Principe général des RPC



Un "**stub**" ou "**proxy**" est un objet local (dans l'espace mémoire du programme client) qui:

- **d'un point de vue données , comporte les informations techniques (*adresse IP , n° port , identifiant objet distant , ...*) nécessaires pour toute communication avec l'objet distant .**
- **d'un point de vue traitements, reprend tous les prototypes de fonctions de l'objet distant . Au sein du Stub, ces méthodes de mêmes noms délèguent les appels au coté serveur .**

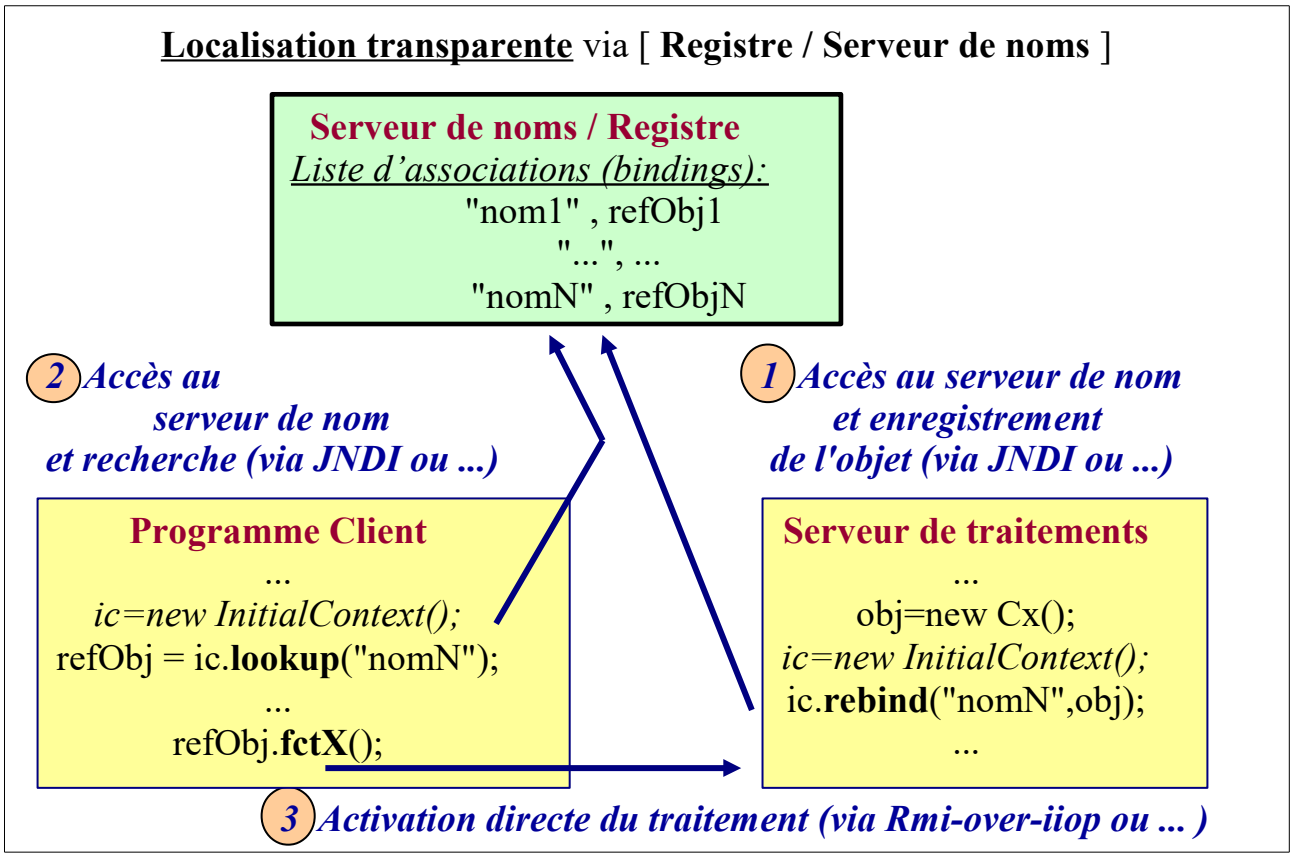
Quelque soit la technologie utilisée (CORBA, RMI, COM+ , Service WEB , ...) , le code (très technique) du **stub** est toujours **généré automatiquement** (via un générateur de code ou via un mécanisme complètement dynamique) .

D'une façon générale, le Stub effectuent les opérations suivantes:

1. rassembler les paramètres d'entrée et nom de fonction à appeler dans un message de requête (format binaire ou XML ou ... dépendant du protocole utilisé)
2. envoyer et attendre la réponse
3. décortiquer la réponse , y extraire le résultat que l'on retourne à l'appelant

NB: Pour les opérations de rassemblement et encodage des données en paramètres (et en retour) , on parle souvent en terme de **Marshalling/UnMarshalling** ou **Sérialisation/Désérialisation** .

3. Localisation transparente / serveur de noms



Le serveur de nom doit toujours être démarré en premier .
Sa grande importance fait qu'un serveur de secours est recommandé (à l'image des serveurs DNS).

Pour l'application cliente , le serveur de noms ne sert qu'a récupérer les paramètres techniques liés à une connexion avec tel objet précis d'un serveur de traitements.
Une fois la connexion établie , tous les appels sont directement acheminés vers le serveur.

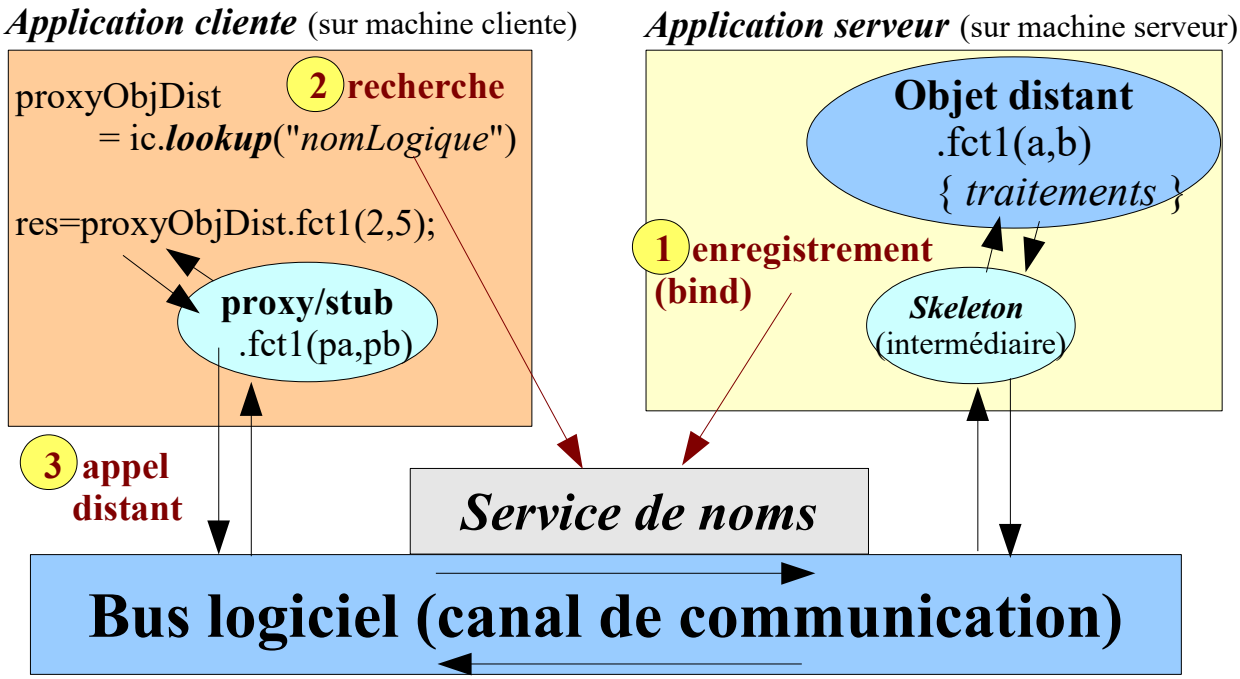
Selon le protocole utilisé le serveur de noms (dans le cadre de RPC orienté objet) peut prendre pleins de formes différentes:

- RmiRegistry (RMI-over-JRMP : RMI de Java à Java)
- TnameServ (Serveur de noms CORBA avec RMI-over-IIOP)
- Base de registre d'un ordinateur Windows (avec DCOM / COM+ de Microsoft)
- Partie interne de JBoss , WebSphere , WebLogic , Jonas ou d'un autre serveur J2EE
- ....

==> Le code client est dépendant du type de serveur de noms qu'il faut utiliser.
Heureusement , certaines API telle que JNDI offrent un paramétrage souple pour configurer les paramètres techniques liés au protocole utilisé .

4. Vue globale sur RPC orienté objet

Bus logiciel *(canal de communication pour objets distribués)*



Le schéma ci dessus montre le dénominateur commun à l'ensemble des technologies existantes.

5. Protocoles et API pour RPC objets synchrones

API et Protocoles pour objets distribués

IIOP:
Internet Inter ORB Protocol
(Protocole lié à l'architecture ouverte ***CORBA*** et à ***TCP/IP***)

RMI (over-JRMP) :
Remote Method Invocation
de Java à Java depuis jdk 1.1

RMI-over-IIOP:
Version de RMI utilisant le protocole IIOP de CORBA comme couche basse. (Java <-> Java ou Java <-> C++, ...) , depuis jdk 1.3

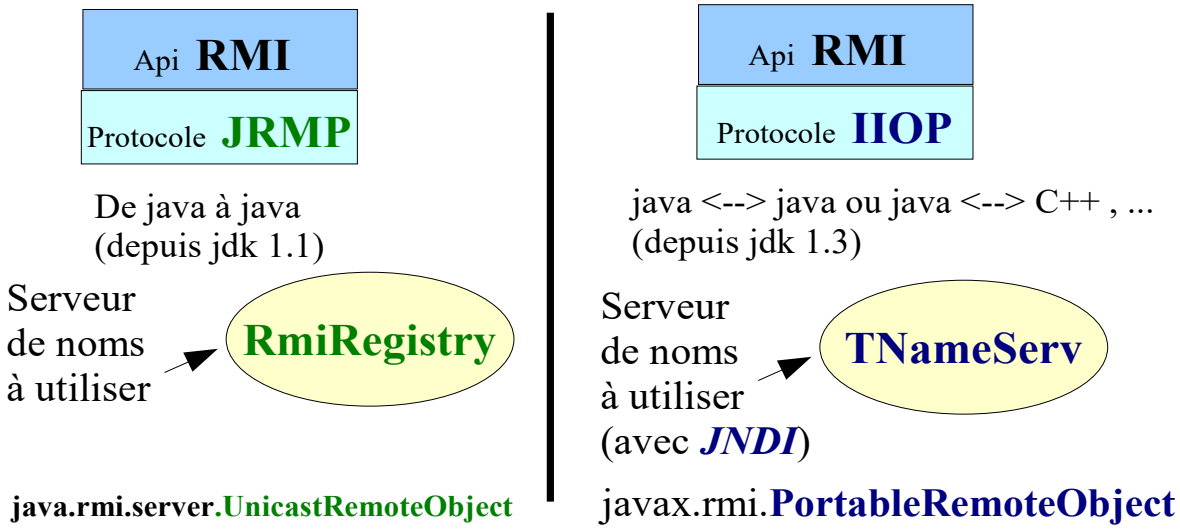
SOAP:
Simple Object Acces Protocol (dialogue Xml sur HTTP)
Interopérabilité totale / Protocole des "service web" .

DCOM/COM+ : Protocole propriétaire de Microsoft .

6. Présentation api RMI (Remote Method Invocation)

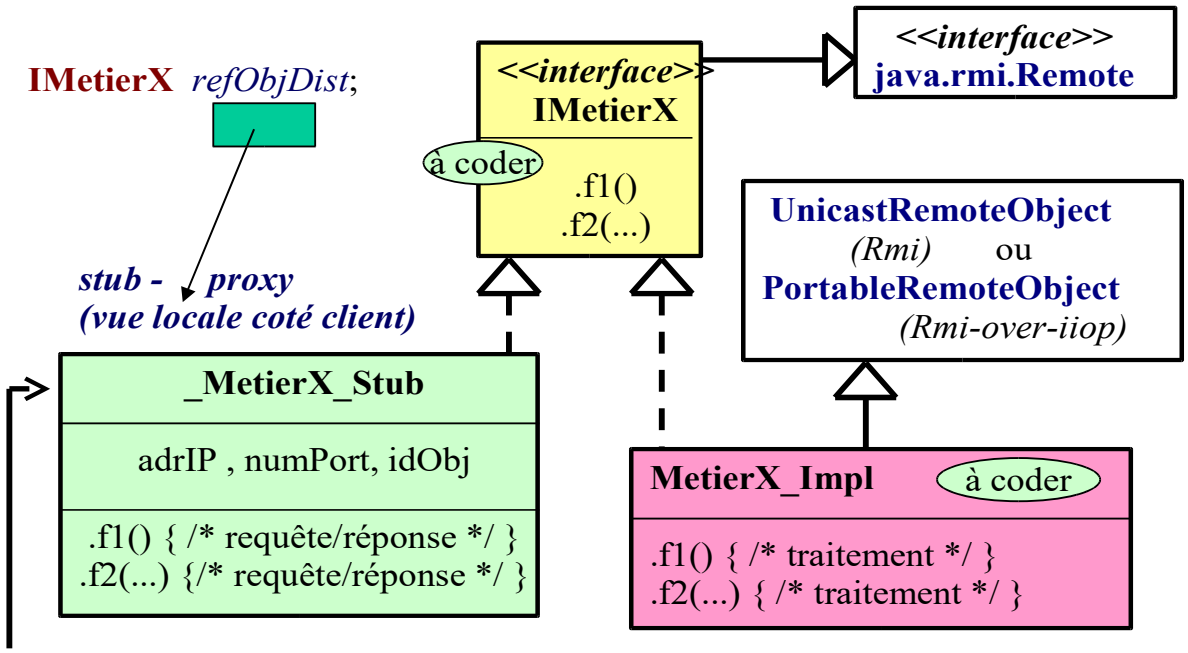
RMI (Remote Method Invocation)

R.M.I. peut s'appuyer sur les protocoles **IIOP** (lié à CORBA) ou **JRMP** (Java Remote Method Protocol).



6.1. Structure du code java (RMI)

R.M.I. – Structure du code java

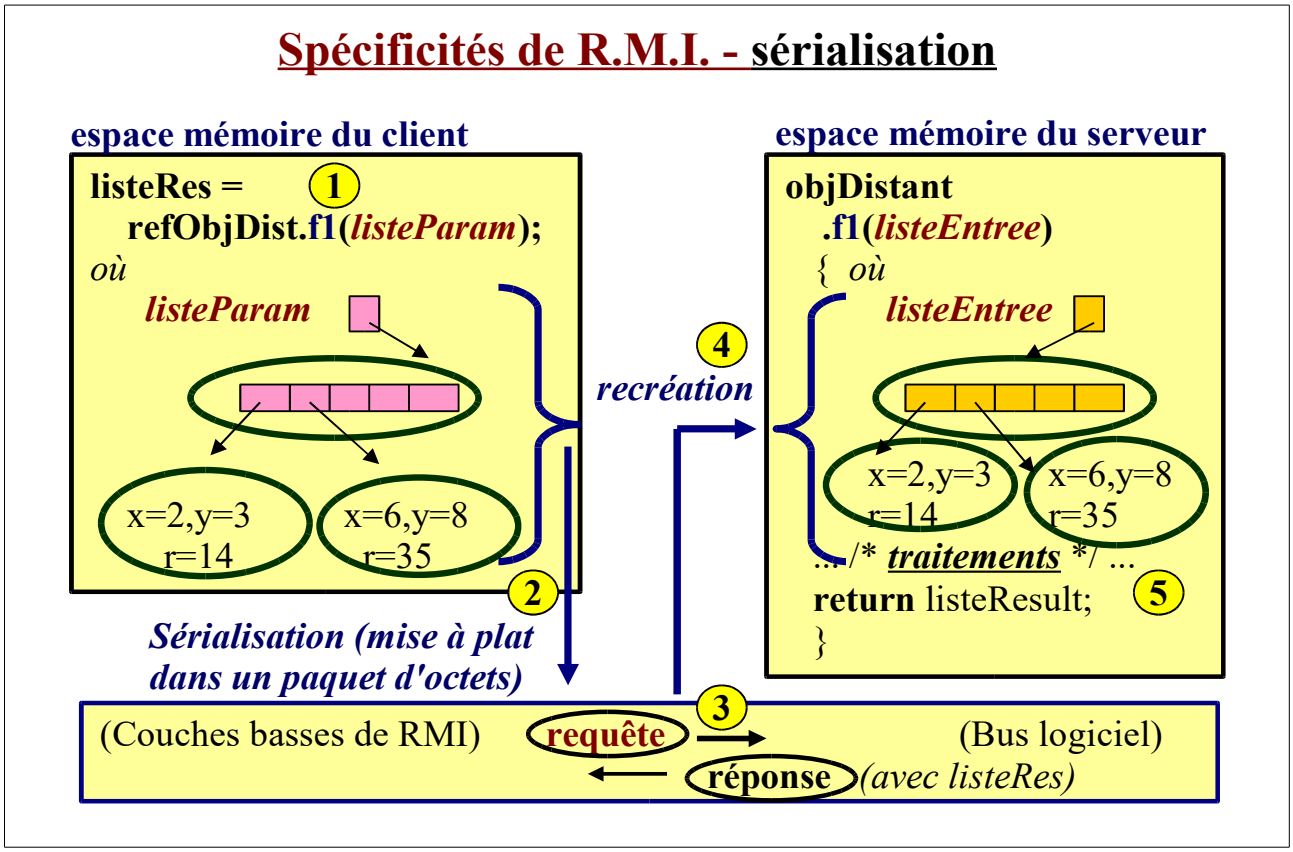


généré automatiquement via **rmic** et ***téléchargé vers le client*** si besoin

6.2. Code type de l'interface de l'objet distant (RMI):

```
package mes_interfaces;
public interface HoraireInfo extends   java.rmi.Remote
{
public String methodeA(int x,String obj) throws java.rmi.RemoteException;
public String getChHeure() throws java.rmi.RemoteException;
}
```

6.3. Passage d'objet en paramètre d'un appel distant RMI:



Très Important:

Lorsque l'on passe en paramètre un objet java local à une fonction distante, celui-ci est entièrement recopié au niveau du serveur.

Il en va de même pour les objets (ou collections d'objets) en "valeur de retour".

Cette recopie automatique fait intervenir les mécanismes de **sérialisation**. Ainsi si l'objet local (en paramètre) comporte des références vers d'autres (sous-)objets locaux, c'est alors toute une grappe d'objets "java" qui est ainsi automatiquement recopiée dans l'espace mémoire du serveur.

7. Code type de l'application cliente (Rmi-JRMP)

```
package clirmi;
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
import mes_interfaces.*;

public class CliRmi  {

public static void main(String[] args) {
  try{
     HoraireInfo  refObjDistant;  //référence sur objet distant (type=interface Remote)
     String chUrl = "rmi://localhost/objetX"; //  "rmi://machineDistanteX/objetX"
    // NB: objetX correspond içi au nom logique de l'objet distant (à enregistrer coté serveur)

   // Pour permettre le téléchargement des "Stub/Proxy" - nécessite "java_rmi_xxx.policy" :
    if(System.getSecurityManager()==null)
       System.setSecurityManager(new RMISecurityManager());

   // Recherche de l'objet distant et tentative de connexion:
   refObjDistant = (HoraireInfo) Naming.lookup(chUrl);

    //Appel transparent d'une méthode distante:
       String chRes = refObjDistant.getChHeure();
     System.out.println("res="+chRes);
    }  catch(Exception ex) { ex.printStackTrace();  }
  }
}
```

variante (utilisant l'api standard JNDI) :

```
...
java.util.Properties props = System.getProperties();
if(props.get("java.naming.factory.initial")==null)
    props.put("java.naming.factory.initial",
                "com.sun.jndi.rmi.registry.RegistryContextFactory");
if(props.get("java.naming.provider.url")==null)
    props.put("java.naming.provider.url",
                "rmi://"+chHostname+":1099");

javax.naming.InitialContext ctx = new javax.naming.InitialContext() ;

 refObjDistant = (HoraireInfo) ctx.lookup("objetX");
        /*Object obj = ctx.lookup("MiniBank");
         refObjDistant = (HoraireInfo) PortableRemoteObject.narrow(obj,HoraireInfo.class);*/
```

8. Code type de l'application serveur:

```
package serveur;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import mes_interfaces.*;

// Classe d'implémentation de l'objet Distant :

public class HoraireImpl  extends UnicastRemoteObject  implements HoraireInfo
{
public HoraireImpl() throws RemoteException
    { super(); // appel obligatoire au de la constructeur classe parente
                // Le constructeur de UnicastRemoteObject exporte l'objet
               // de façon à ce qu'il soit accessible à distance.
    }
public String methodeA(int x,String ch) throws RemoteException
      {
       String res="...";       res=ch+"  " +x;          return res;
      }
public String getChHeure() throws RemoteException   { return "10:50:20"; }
} // fin de HoraireImpl
```

```
package serveur;
public class AppServRMI {
public static void main(String[] args) {
System.out.println("Serveur RMI—AppServRMI ");
try {
   HoraireImpl objet = new HoraireImpl(); // exportation via constructeur
   String nomObjet = "objetX" ;

  java.rmi.Naming.rebind(nomObjet,objet); //enregistre l'objet dans RMIRegistry

 /* Variante JNDI (à paramétrer via propriétés systèmes) :
      javax.naming.InitialContext   ctx =  new   javax.naming.InitialContext() ;
      ctx.rebind(nomObjet,objet);
 */
  } catch(Exception ex) { ex.printStackTrace();}
 }// Fin du main , le processus continue tout de même.
}
```

9. Mise en oeuvre standard (de la rigueur s'impose)

9.1. Scripts ".bat"

InitVars.bat

```
REM     Sous script permettant de fixer les valeurs  de certaines variables d'environnement .
REM   Machine sur laquelle tourne le serveur de nom (%NAME_SVR_HOST%)
          set NAME_SRV_HOST=localhost
REM   répertoire du projet (%DIR_PRJ%)
          set DIR_PRJ=c:\tp\java\workspaces\java_j2se\workspace\tp_rmi
REM   Répertoire ou seront générés les stubs (%DIR_STUB%)
          set DIR_STUB=%DIR_PRJ%\stub
REM   Mise à jour du Path en fonction du jdk installé sur le poste:
          set PATH=%PATH%;C:\Prog\JAVA\j2sdk1.4.2_06\bin
REM   chemins menant au classes compilées (%DIR_BIN%)
          set  DIR_BIN=%DIR_PRJ%\bin
REM   chemins menant aux scripts (%DIR_SCRIPTS%)
          set  DIR_SCRIPTS=%DIR_PRJ%\script
REM   Chemins à ajouter au classpath (%CP%)
          set  CP=%DIR_BIN%;%DIR_STUB%;.
```

LancerRmic.bat (point clef)

```
call initVars.bat
REM   Lancer le compilateur d'amorces RMI pour générer XXX_Stub
rmic -d  %DIR_STUB%  -classpath  %CP%     serveur.XxxxImpl
REM   Recopier le fichier d'interface XXX.class dans ce répertoire des amorces c:\...\stub
copy %DIR_BIN%\yyy\XXX.class  %DIR_STUB%\yyy\XXX.class
```

LancerRmiRegistry.bat

```
call initVars.bat
start rmiregistry -J-Djava.security.policy=%DIR_SCRIPTS%\for_rmi.policy
```

LancerServeurRMI.bat (point clef)

```
call initVars.bat
java  -classpath %CP%
          -Djava.rmi.server.codebase=file:///%DIR_STUB%/  serveur.AppServ
REM   la propriete java.rmi.server.codebase doit finir par /
REM   le classpath doit comporter de quoi accéder aux Stub
```

LancerClientRMI.bat (exemple)

```
call initVars.bat
java   -classpath %CP%
            -Djava.security.policy=%DIR_SCRIPTS%\for_rmi.policy   clirmi.CliRmi
REM :  Le Stub nécessaire au client est normalement automatiquement téléchargé
REM    depuis java.rmi.server.codebase
```

9.2. Script ANT (build.xml)

initVars.properties

```
# nom (ou adresse IP) du serveur distant:
NAME_SRV_HOST=localhost
# repertoire du projet :
DIR_PRJ=c:/tp/java/workspaces/java_j2se/workspace/tp_rmi
# nom du package java contenant les interfaces distantes :
ITF_PACKAGE=finance
# suffixe des classes d'implementation coté serveur (filtre) :
SUFFIXE_IMPL_FILTER=Impl
# repertoire bin du jdk (comportant les commande rmic , rmiregistry et tnameserv)
JDK_BIN=C:/Prog/JAVA/j2sdk1.4.2_06/bin
# classe principale du programme serveur:
SERVER_MAIN_CLASS=serveur.AppServMiniBankV2
# classe principale du programme client:
CLIENT_MAIN_CLASS=client.MiniBankSimpleClientV2
```

build.xml

```
<?xml version="1.0"?>

<project name="project" default="run_Client">
   <description>lancement de rmic, RmiRegistry ,Serveur et client</description>

      <property file="initVars.properties" />
      <property name="DIR_STUB" value="${DIR_PRJ}/stub" />
      <property name="DIR_BIN" value="${DIR_PRJ}/bin" />
      <property name="DIR_ANT_SCRIPT" value="${DIR_PRJ}/script_ant" />

    <!--    target: run_rmic  ====== -->
    <target name="run_rmic"  description="lancement de rmic">
    <echo message="génération des Stubs RMI dans le repertoire ${DIR_STUB}" />
      <rmic base="${DIR_BIN}"  includes="**/*${SUFFIXE_IMPL_FILTER}.class" />
            <copy todir="${DIR_STUB}">
                <fileset dir="${DIR_BIN}">
                     <include name="**/*_Stub.class"/>
                   <include name="${ITF_PACKAGE}/*.class"/>
                </fileset>
             </copy>
    </target>

   <!-- ============ target: run_RmiRegistry  ===== -->
      <target name="run_RmiRegistry"  description="lancement de rmiRegistry">
           <echo message="Lancement du serveur de noms (RmiRegistry)" />
    <exec dir="${JDK_BIN}" executable="rmiregistry.exe" spawn="true">
       <arg line="-J-Djava.security.policy=${DIR_ANT_SCRIPT}/for_rmi.policy" />
    </exec>
            <echo message="Arret via gestionnaire de taches ou ps-ef , kill" />
    </target>

      <!-- ======  target: run_Server ====== -->
      <target name="run_Server"  depends="run_rmic,run_RmiRegistry"
              description="lancement du serveur de traitements">
      <echo message="Lancement du serveur" />
      <property name="shell" location="C:/WINDOWS/system32/cmd.exe"/>
      <property name="startLine"
           value="/C 'start cmd /K  java -classpath ${DIR_BIN};${DIR_STUB}
                 -Djava.rmi.server.codebase=file:///${DIR_STUB}/
                  ${SERVER_MAIN_CLASS}' "/>
        <!-- ou bien -Djava.rmi.server.codebase=
                  http://${NAME_SVR_HOST}:8080/SiteWebAvecStubs/ -->
```

```
                <echo message="${startLine}" />
                  <exec executable="${shell}" spawn="true">
                      <arg line="${startLine}"/>
                  </exec>

      </target>

      <!-- =======  target: run_Client  ======== -->
      <target name="run_Client"  depends="run_rmic,run_RmiRegistry,run_Server"
              description="lancement du client">
      <echo message="Lancement du client" />

      <java fork="true"  classname="${CLIENT_MAIN_CLASS}">
                  <classpath>
                      <pathelement path="${java.class.path}"/>
                      <pathelement  location="${DIR_BIN}"/>
                      <pathelement  location="${DIR_STUB}"/>
                  </classpath>
          <sysproperty key="java.security.policy"
                       value="${DIR_ANT_SCRIPT}/for_rmi.policy" />
          <arg  line="localhost"/>
      </java>

      </target>
</project>
```

9.3. Explications (points clefs):

rmic est l'utilitaire livré avec le jdk qui permet de générer les classes Java correspondant aux stub et squelettes RMI.
Depuis la version 1.2 du jdk, la classe du squelette (intermédiaire coté serveur) n'est plus nécessaire pour les mécanismes internes.
NB:

- La propriété système **java.rmi.server.codebase** (dont la valeur est précisée au lancement du serveur) indique l'endroit depuis lequel on pourra télécharger les classes "stub" nécessaires à RMI. Cette **URL** (en file:// , ftp:// ou http://) est stockée dans le registre Rmi lors de l'enregistrement (bind).
- Lorsqu'un client demande une référence à un objet distant, le registre renvoie le stub qui va bien. Si le client trouve dans son CLASSPATH la classe Java correspondant au stub, il l'utilise alors directement. Si par contre , la classe du Stub (ou une autre) n'est pas disponible au niveau du client, il y a alors un **téléchargement automatique** de celle-ci qui est effectuée depuis la valeur de **java.rmi.server.codebase** que le client récupère depuis Rmiregistry**.**
- La propriété *java.rmi.server.codebase* est également nécessaire pour **rmiregistry** qui est un client Rmi particulier**.**

10. Sécurité avec RMI

Le code d'un client peut éventuellement comporter les instructions suivantes:

```
if( System.getSecurityManager() == null)
      System.setSecurityManager( new  RMISecurityManager() );
```

Ceci permet au client de pouvoir éventuellement télécharger le code de certaines classes Java (Interface remote, Stub) depuis un autre endroit que le CLASSPATH défini localement.

java –Djava.security.policy=*for_rmi.policy* cli.ClientRmi

appletviewer –J–Djava.security.policy=*for_rmi.policy*
rmiregistry –J–Djava.security.policy=*for_rmi.policy*

for_rmi.policy peut être écrit de la façon suivante:

```
grant {
  permission java.net.SocketPermission "*", "connect, resolve";
  permission java.util.PropertyPermission "*", "read, write";
  permission java.io.FilePermission "<<ALL FILES>>", "read";
};
```

On peut également se baser sur les fichiers de régulation par défaut:

%JAVA_HOME%\lib\security\java.policy (lié à la machine virtuelle)
%USER_HOME%\.java.policy (propre à un utilisateur , **policytool.exe**)

11. RMI over IIOP

RMI peut être considéré comme une couche haute pouvant s'appuyer sur JRMP

(Java Remote Method Protocol) ou bien sur IIOP (Internet Inter-ORB Protocol).

- **JRMP** est le protocole par défaut dédié à RMI;
- **IIOP** est le protocole standard pour faire communiquer les objets CORBA.

RMI over IIOP permet à des objets Java distants de communiquer avec des objets CORBA qui peuvent éventuellement être écrits en C++ .

La principale nouveauté liée à "RMI over IIOP" consiste à ce que la classe d'implémentation (coté serveur) hérite de `javax.rmi.PortableRemoteObject` à la place de `java.rmi.server.UnicastRemoteObject`.

D'autre part le serveur de nom (hiérarchisé) à utiliser est **tnameserv.exe** (implémentant CosNaming de CORBA).

Les spécifications de **J2EE** préconisent l'utilisation de **RMI-over-IIOP** pour des raisons d'ouverture: les EJB sont ainsi accessibles depuis un programme C++ ou autre.

Mise en œuvre de RMI-over-IIOP:

rmic –iiop ➔ stub / squelette version RMI-over-IIOP

(possibilité d'implémentation duale JRMP / IIOP) .

rmic –idl ➔ pour éventuel client C++ ou autre.

Le client et le serveur peuvent accéder au service de nom via **JNDI**:

```
java –Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
     -Djava.naming.provider.url=iiop://servnom:900          ClientRmiOverIIOP
```

où **servnom** correspond à la **machine sur laquelle est lancé tnameserv.exe** .

11.1. Fragment de code du client:

```
import javax.rmi.PortableRemoteObject;
import javax.naming.InitialContext;
…
InitialContext ctx = new InitialContext();
Object obj = ctx.lookup("/NomObjet");
HelloInterface myobj =    (HelloInterface)
            PortableRemoteObject.narrow(obj,HelloInterface.class);
…
```

11.2. Fragment de code du serveur:

```
import java.rmi.*;
import javax.naming.*;

HelloServer  obj = new HelloServer() ;  // classe héritant de PortableRemoteObject

/* Fixer des valeurs par défaut pour ce qui concerne les propriétés d'accès au serveur de nom
tnameserv.exe */
java.util.Properties props = System.getProperties();
if(props.get("java.naming.factory.initial")==null)
   props.put("java.naming.factory.initial","com.sun.jndi.cosnaming.CNCtxFactory");
if(props.get("java.naming.provider.url")==null)
  props.put("java.naming.provider.url","iiop://localhost:900");

InitialContext ctx = new InitialContext();
ctx.rebind("/NomObjet",obj);…
```

Essentiel JMS (Java Message Service)

12. Queue & Topic

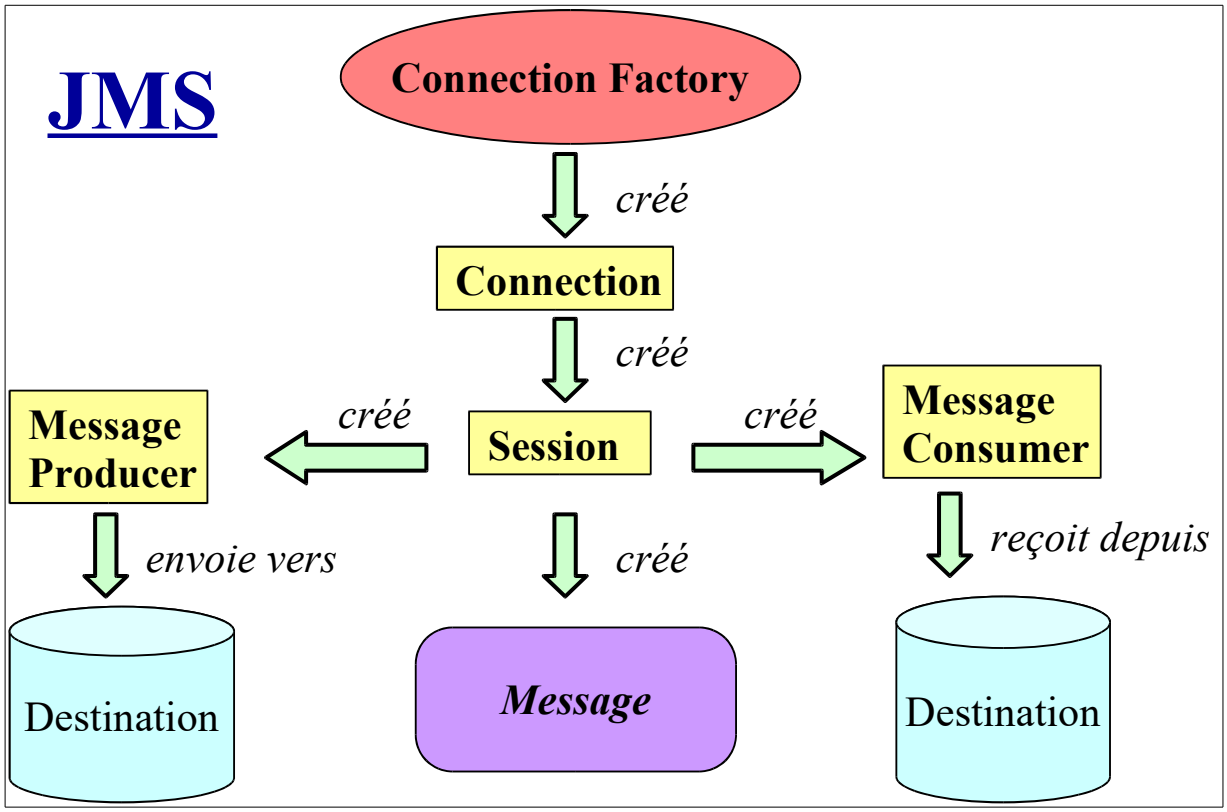
2 modèles de destination :

- **PTP** (Point To Point) - *File de message spécifique à une entité (comparable à une boîte à lettres).*
- **Pub/Sub** (Published & Subscribe) - *Les clients et les serveurs utilisent une même file logique dont le contenu est organisé de façon hiérarchique. On publiera des messages au niveau d'un certain noeud et ceux qui se sont inscrits (abonnés) vis à vis de ce noeud recevront alors ces messages.*

Le tableau ci-dessous résume les différentes **interfaces** utilisées au niveau de l'api JMS:

JSM (Interface générique)	PTP Domain	Pub/Sub Domain
ConnectionFactory *(mt)*	**QueueConnectionFactory**	**TopicConnectionFactory**
Connection *(mt)*	**QueueConnection**	**TopicConnection**
Destination *(mt)*	**Queue**	**Topic**
Session	**QueueSession**	**TopicSession**
MessageProducer	**QueueSender**	**TopicPublisher**
MessageConsumer	**QueueReceiver,QueueBrowser**	**TopicSubscriber**

(mt) : multi-threading support.



13. Exemples (fragments) de code

13.1. Obtention de l'objet ConnectionFactory via JNDI:

```
QueueConnectionFactory queueConnectionFactory;
Context messaging = new InitialContext();
queueConnectionFactory = (QueueConnectionFactory)
messaging.lookup("QueueConnectionFactory");
```

ou bien

```
TopicConnectionFactory topicConnectionFactory;
Context messaging = new InitialContext();
topicConnectionFactory = (TopicConnectionFactory)
messaging.lookup("TopicConnectionFactory");
```

13.2. Obtention d'une file de message via JNDI:

```
Queue stockQueue;
stockQueue = (Queue) messaging.lookup("StockQueue");
```

ou bien

```
Topic stockTopic;
stockTopic = (Topic) messaging.lookup("StockTopic");
```

13.3. Création d'un objet Connection via l'usine:

```
QueueConnection queueConnection;
queueConnection = queueConnectionFactory.createQueueConnection();
```

ou bien

```
TopicConnection topicConnection;
topicConnection = topicConnectionFactory.createTopicConnection();
```

13.4. Création d'une Session à partir de l'objet Connextion:

```
QueueSession session;
session = queueConnection.createQueueSession(false /*trasact. */,
                                    Session.AUTO_ACKNOWLEDGE);
```

ou bien

```
TopicSession session;
session = topicConnection.createTopicSession(false,
                                    Session.CLIENT_ACKNOWLEDGE);
```

13.5. Obtention de l'objet "MessageProducer" pour envois

```
QueueSender sender;
sender = session.createSender(queue);
```

ou bien

```
TopicPublisher publisher;
publisher = session.createPublisher(stockTopic);
```

13.6. Obtention de l'objet "MessageConsumer" pour réceptions

```
QueueReceiver receiver;
receiver = session.createReceiver(queue);
```

ou bien

```
TopicSubscriber subscriber;
subscriber = session.createSubscriber(stockTopic);
```

et

```
StockListener myListener;
subscriber.setMessageListener(myListener);
```

avec

```
public class StockListener implements javax.jms.MessageListener {
void onMessage(Message message) {
// unpack and handle the messages we receive.
    }
}
```

13.7. Déclencher le début possible des réceptions de messages:

queueConnection.start(); *ou bien* **topicConnection.start();**

13.8. Création de messages

Création d'un message binaire:

```
byte[] stockData; // stock information as a byte array
BytesMessage message;
message = session.createByteMessage();
message.writeBytes(stockData);
```

Création d'un message en mode texte:

```
String stockData; // stock information as a String
TextMessage message;
message = session.createTextMessage();
message.setText(stockData);
```

Création d'un message structuré (avec différents champs nommés):

```
MapMessage message;
message = session.createMapMessage();
message.setString("Name", stockName); message.setDouble("Value", stockValue);
message.setLong("Time", stockTime); message.setDouble("Diff", stockDiff);
message.setString("Info", stockInfo);
```

Création d'un message à données à lire séquentiellement:

```
StreamMessage message;
message = session.createStreamMessage();
message.writeString(stockName); ...message.writeDouble(stockValue);
message.writeLong(stockTime); ...
```

Création d'un message contenant les valeurs d'un objet Java (Sérialisation):

```
ObjectMessage  message = session.createObjectMessage();
message.setObject(stockObject);
```

13.9. Envoi et réception

Envoi et réception d'un message en mode PTP:

```
sender.send(message);
```

et

```
StreamMessage stockMessage = (StreamMessage) receiver.receive();
```

Publication et réception d'un message en mode Pub/Sub:

```
publisher.publish(message);
```

et

*appel automatique de la méthode **OnMessage()** de l'abonné*

13.10. Extraction des valeurs d'un message

Extraction des valeurs d'un message binaire:

```
byte[] stockData; // stock information as a byte array
int length;
length = message.readBytes(stockData);
```

Extractions des valeurs d'un message texte:

```
String stockData;
stockData = message.getText();
```

Extractions des valeurs d'un message structuré:

```
stockName = message.getString("Name");
stockValue = message.getDouble("Value");
stockTime = message.getLong("Time");
stockDiff = message.getDouble("Diff");
stockInfo = message.getString("Info");
```

Extractions des valeurs d'un message à valeurs séquentielles:

```
stockName = message.readString();  stockValue = message.readDouble();
stockTime = message.readLong();  stockDiff = message.readDouble();
stockInfo = message.readString();
```

Extractions des valeurs d'un message objet:

```
stockObject = message.getObject();
```

13.11. Filtrage éventuel des messages que l'on souhaite récupérer:

```
String selector;
selector = new String("(name = 'SUNW') OR (name = 'IBM')");
```

et

```
QueueReceiver receiver;
receiver = session.createReceiver(queue, selector);
```

ou

```
TopicSubscriber subscriber;
subscriber = session.createSubscriber(topic, selector);
```

Nb: le package **javax.jms** est intégré dans **j2ee.jar** .

14. Champs des entêtes de message

Champ de l'entête	*signification*	*fixé (affecté) par*
JMSDestination	File de destination	méthode **send**()
JMSDeliveryMode	PERSITENT ou NON_PERSISTENT	méthode **send**()
JMSExpiration	0 : pas d'expiration. sinon *n* **ms** à vivre.	méthode **send**()
JMSPriority	priorité de 0 à 9 (0-4: normal) (5-9: high)	méthode **send**()
JMSMessageID	***ID:xxx*** identifiant du message	méthode **send**()
JMSTimestamp	estampillage de temps	méthode **send**()
JMSCorrelationID	identifiant de la requête associée à la réponse	Client
JMSReplyTo	File où il faut placer la réponse.	Client
JMSType	selon le contexte , catégorie , ...	Client
JMSRedelivered	si réception multiple d'un même message	Provider

Le champ **JMSReplyTo** peut comporter le nom d'une file (éventuellement temporaire) que l'émetteur de la requête a préalablement créé pour récupérer la réponse..

15. Acquittement des messages reçus.

Principe de base (*lié aux mécanismes internes*): Il faut **acquitter un message une fois consommé pour ne pas le recevoir une nouvelle fois** .

mode d'acquittement	*caractéristiques*
Session.**CLIENT_ACKNOWLEDGE**	il faut appeler explicitement message.**acknowlegde**() .
Session.**AUTO_ACKNOWLEDGE**	automatique en fin de ***OnMessage*** ou en fin de ***receive***()
Session.**DUPS_OK_ACKNOWLEGDE**	Si recevoir un message dupliqué n'est pas 1 pb

Nb: L'acquittement ne revient pas vers l'émetteur . Il n'est pris en compte que par les mécanismes internes qui doivent normalement garantir q'un même message (identifié via un ID) envoyés plusieurs fois ou bien dupliqué ne sera reçu qu'une seule fois.

16. Liste des principaux "Provider JMS"

La liste des principales implémentations disponibles se trouve au bout de l'url suivante:
http://java.sun.com/products/jms/vendors.html

Parmi les implémentations les plus connues, on peut citer:

- **MQSeries** (→ renommé **WebSphere_MQ**) d' **IBM**
- **Open Message Queue** de **Sun**
- Services "JMS" intégrés dans les serveurs J2EE (JBoss, WebLogic, WebSphere).

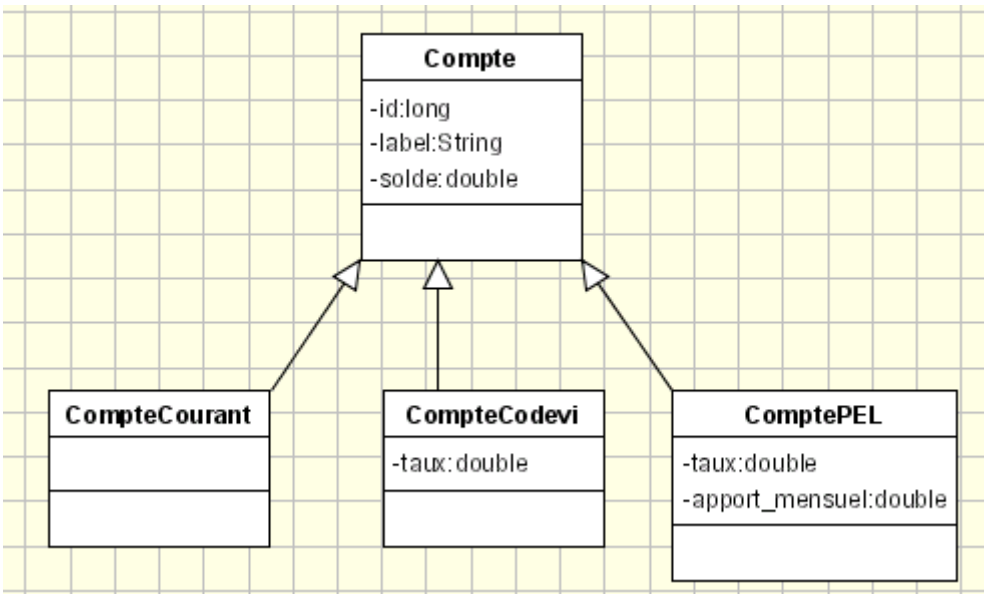
Produits (FreeWare - OpenSource):

- **OpenJMS**
- ***ActiveMQ (d'Apache Software)***
- **Joram de OW2**
- **…**

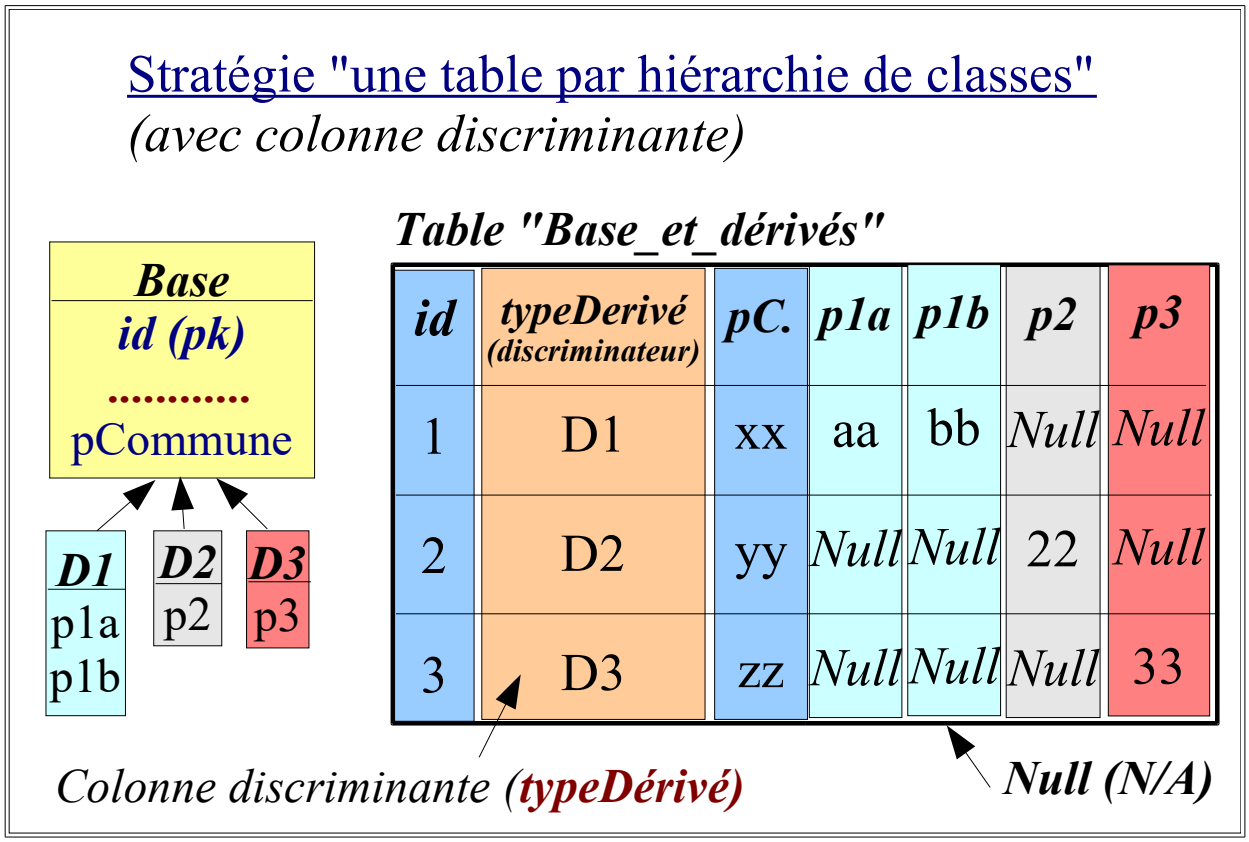
XVIII - Annexe – Détails sur JPA

1. Relations d'héritage & polymorphisme

UML:



1.1. stratégie "une seule table par hiérarchie de classes"



Une seule grande table permet d'héberger les instances de toute une hiérarchie de classe.

Pour distinguer les instances des différentes sous classes , on utilise une propriété discriminante (à telle valeur correspond telle sous classe).

Cette stratégie (relativement simple) est assez pratique et adaptée dans le cas où il y a peu de différences structurelles ente les sous classes .

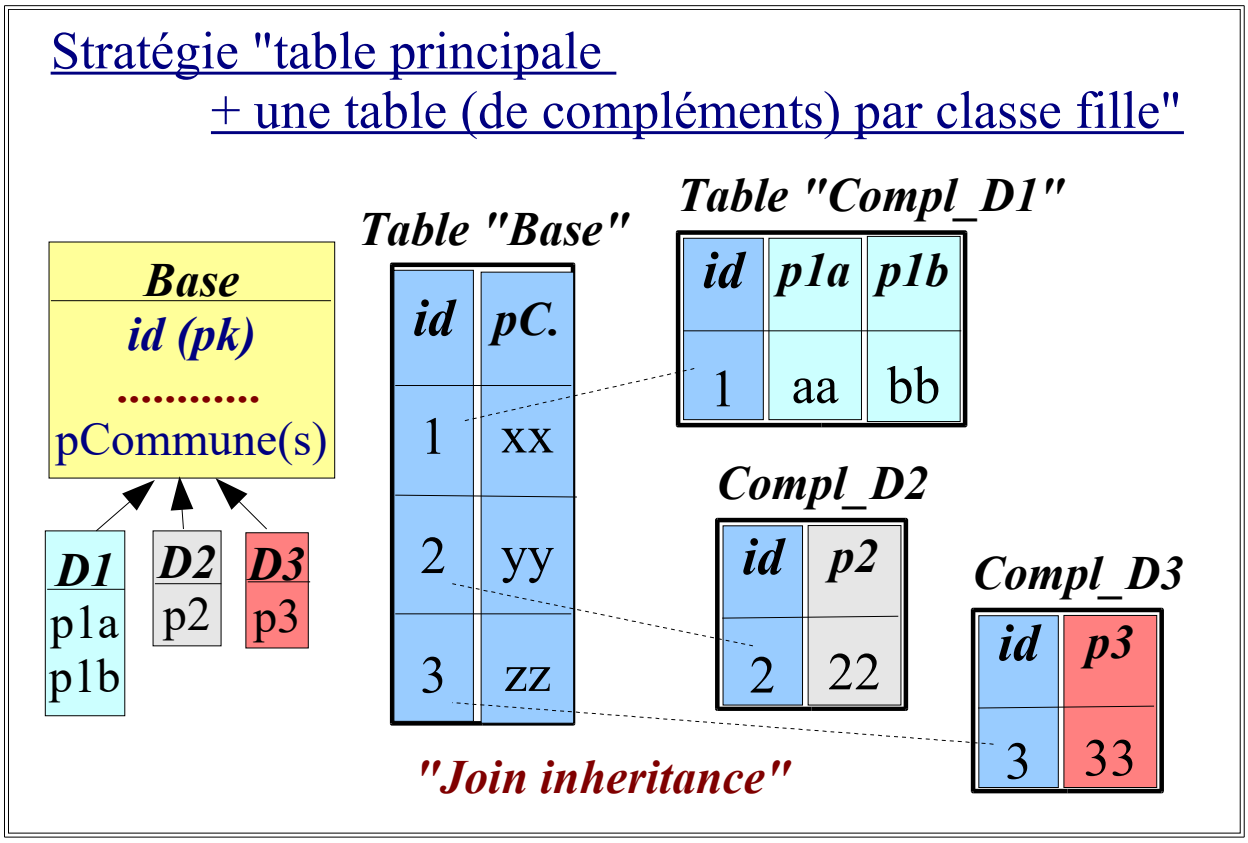
Contrainte : les colonnes liées aux attributs des sous classes ne peuvent pas avoir la contrainte "NOT NULL" . (***Pour une instance de la sous classe D1 , les colonnes inutilisées de la sous classe D2 doivent avoir la valeur NULL)***.

Pour la classe de base (JPA):

```
@Entity
@Table(name="Compte")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "typeCompte",
                     discriminatorType = DiscriminatorType.STRING)
/* @DiscriminatorValue("CompteCourant") //éventuellement en tant que valeur par défaut
   // et si pas de sous classe CompteCourant mais utilisation directe du parent "Compte" */
public class Compte
{
  @Id @GeneratedValue(strategy=GenerationType.AUTO)
  private long id;
  private String label;
  private double solde;

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

  public String getLabel()  {  return label;  }
  public void setLabel(String label)  { this.label = label;  }

  public double getSolde() { return solde;}
  public void setSolde(double solde) {this.solde = solde;  }
}
```

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorValue("PEL")
public class ComptePEL extends Compte
{
  private double tauxInteret;    //+get/set
  private double apportMensuel;  //+get/set
  ...
}
```

1.2. Stratégie "Join_inheritance"




```
@Entity
@Table(name="Compte")
@Inheritance(strategy=InheritanceType.JOINED )
public class Compte
{
  @Id @GeneratedValue(strategy=GenerationType.AUTO)
  private long id;
  private String label; // +get/set
  private double solde; // +get/set

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

NB : Bien que "pas indispensable" , une éventuelle colonne discriminante pourrait éventuellement être ajoutée si l'implémentation de JPA en tient compte du point de vue des performances.

```
@Entity
@Table(name="ComptePEL")
@Inheritance(strategy=InheritanceType.JOINED )
public class ComptePEL extends Compte
{
  private double tauxInteret;    //+get/set
  private double apportMensuel;  //+get/set
  ...
}
```

1.3. Stratégie (assez rare) "une table par classe"

Avec tout un tas de restrictions (voir documentation de référence si besoin)

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Compte
{
   @Id
   @GeneratedValue(strategy=GenerationType.TABLE)
 ...
  public int getId()   {    return id;  }

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

   ...
}
```

```
@Entity
public class ComptePEL extends Compte
{
   private double tauxInteret; // +get/set
   ...
}
```

1.4. Polymorphisme

Une requête JPQL/HQL exprimée avec un type correspondant à une classe parente retournera (par défaut / sans restriction explicite) **toutes les instances de toute une hiérarchie de classes (classe mère + classe fille)**. Le type exact de chacune des instances retournées sera précis (bien que compatible avec le type de la sur-classe) ==> **polymorphisme complet** (java/mémoire + au niveau O.R.M.).

Autrement dit :

select c from Compte as c" retournera tous les types de Compte (courant , codevi , pel).

"select c from ComptePEL as c" ne retournera que des comptes de types "PEL" .

1.5. Une entité répartie dans 2 tables (principale,secondaire)

NB: avec JPA, il est également possible d'associer quelques propriétés d'une entité à une table secondaire :

```
@Entity
@Table(name = "CUSTOMER")
@SecondaryTable(name = "EMBEDDED_ADDRESS")
public class Customer implements java.io.Serializable
{
  @Column(name = "STREET", table = "EMBEDDED_ADDRESS")
  private String street ;

  @Column(name = "CITY", table = "EMBEDDED_ADDRESS")
  private String city ;
 ...
  public String getStreet()   { return street;  }
  public void setStreet(String street)   {   this.street = street;  }

  public String getCity()   {      return city;   }
  public void setCity(String city)   {      this.city = city;   }

}
```

==> dans ce cas (clef primaire table_secondaire = clef_primaire_table_principale)
[cas du <one-to-one> des .hbm.xml de hibernate]

1.6. Eléments requis sur une classe d'entité (JPA):

- vraie classe (pas interface ni enum) non finale (pas de mot clef "final" , on doit pouvoir en hériter).
- un constructeur par défaut (sans argument) obligatoire et devant être public ou protégé (d'autres constructeurs sont possibles mais non indispensables).
- annotation "@Entity" ou bien <entity ...> dans un descripteur Xml.
- éventuelle implémentation de *java.io.Serializable* si besoin d'un transfert distant à l'état détaché.

NB:

- Des héritages sont possibles (entre "entité" et "non entité" et vice versa) et le polymorphisme peut s'appliquer sans problème.
- L'état d'une entité est liée à l'ensemble des valeurs de ses attributs (devant être privés ou protégés) .
- L'accès aux propriétés d'une entité doit s'effectuer via des "getter/setter" (conventions

JavaBean).

La persistance (mapping ORM) s'applique sur toutes les propriétés non "transientes" d'une entité.

Sachant qu'une éventuelle propriété non persistante doit être marquée via "private transient ...." et par l'annotation "@Transient" .

1.7. Verrous (optimistes et pessimistes)

```
@Version
   @Column(name = "OPTLOCK")
   public Integer getVersion()
   {
      return version;
   }

   public void setVersion(Integer i)
   {
      version = i;
   }
```

2. Cycle de persistance (pour @Entity) et annotations/callbacks associées pour "Listener"

Annotations (callback)	*déclenchement (étape du cycle de persistance)*
@PrePersist	juste avant passage à l'état persistant
@PostPersist	juste après passage à l'état persistant (clef primaire souvent affectée)
@PreRemove	juste avant suppression
@PostRemove	juste après suppression et destruction
@PreUpdate	juste avant mise à jour
@PostUpdate	juste après mise à jour
@PostLoad	juste après chargement en mémoire (suite à une recherche)

```
@Entity
@Table(name = "CUSTOMER")
@EntityListeners(CustomerCallbackListener.class)
public class Customer implements java.io.Serializable
{
...
}
```

```
public class CustomerCallbackListener
{
   @PrePersist
   public void doPrePersist(Customer customer) {
      System.out.println("doPrePersist: About to create Customer: " + ...);
   }

   @PostPersist public void doPostPersist(Customer  customer) {
      System.out.println("doPostPersist: Created Customer: " + ...);
   }

   @PreRemove public void doPreRemove(Customer customer) {
      System.out.println("doPreRemove: About to delete Customer: " + ...);
   }

   @PostRemove public void doPostRemove(Customer customer)  {
      System.out.println("doPostRemove: Deleted Customer: " +...);
   }

   @PreUpdate public void doPreUpdate(Customer customer) {
      System.out.println("doPreUpdate: About to update Customer: " +...);
   }

   @PostUpdate public void doPostUpdate(Customer customer) {
      System.out.println("doPostUpdate: Updated Customer: " +...);
   }

   @PostLoad public void doPostLoad(Customer customer) {
      System.out.println("doPostLoad: Loaded Customer: " + ...);
   }
}
```

Annexe - Essentiel JNDI

3. JNDI (pour se connecter à un EJB ou ...)

Présentation de JNDI

API Java ...

JNDI = Java Naming & Directory Interface

Permettant d'accéder à ...

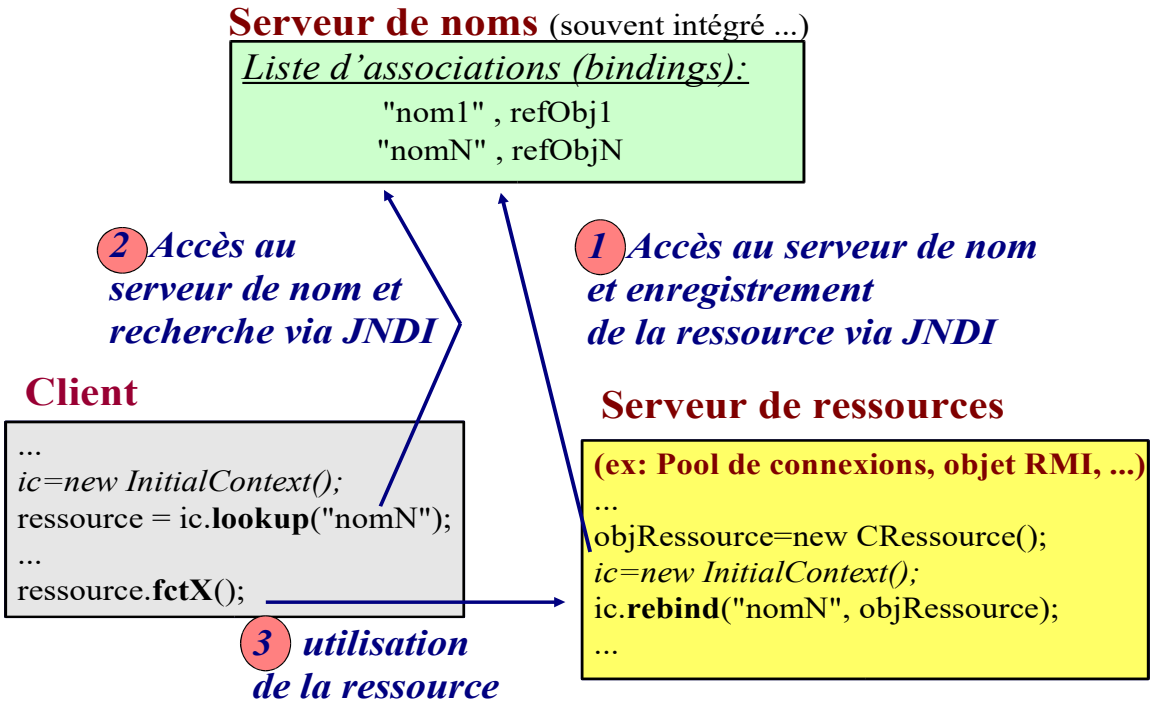
- Des **services de noms** : liste d'associations (bindings) entre des ressources et des noms logiques.

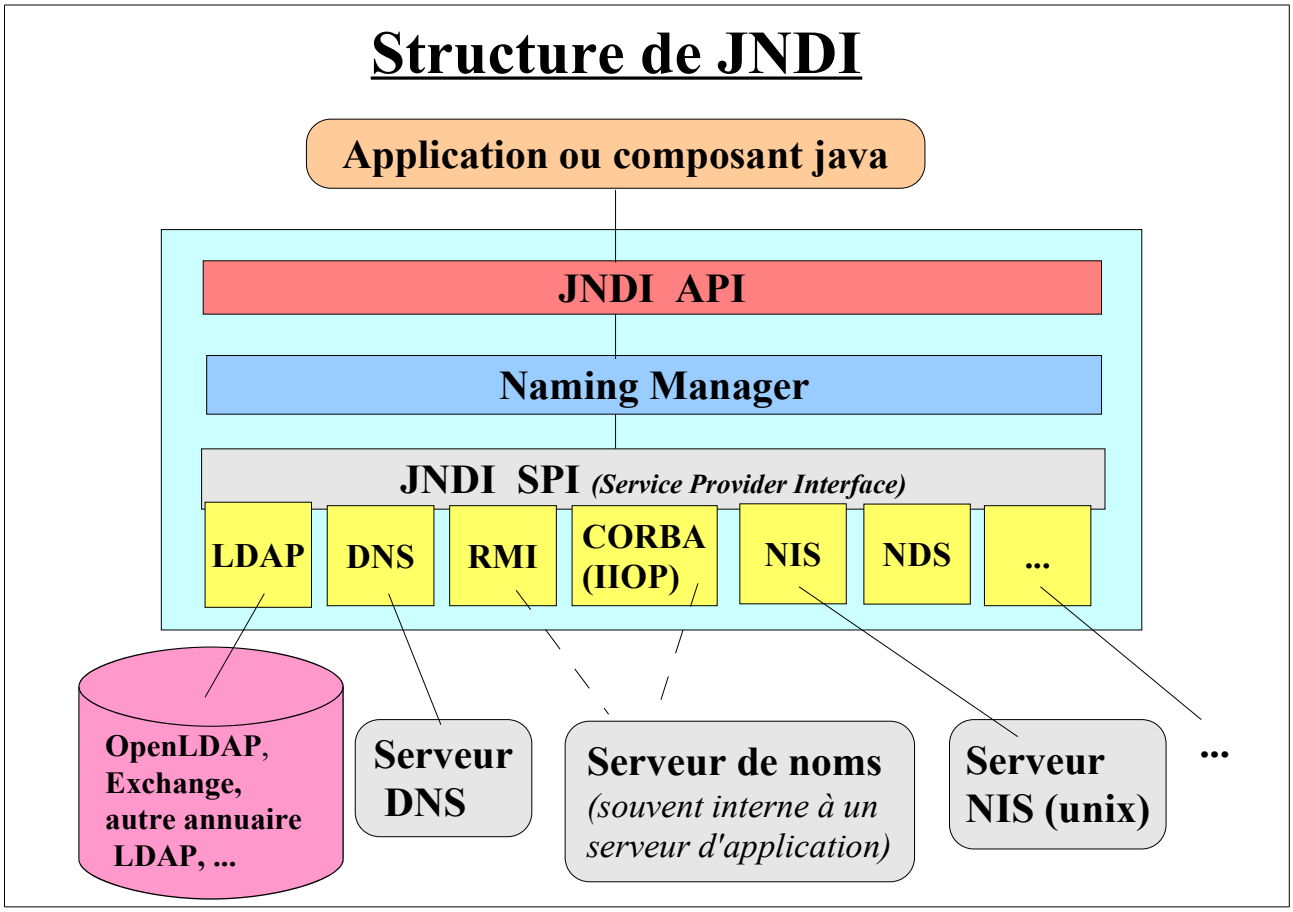
Ex: Liste de composants (EJB, ...) et de ressources techniques (pool, ...)

- Des **services d'annuaire (Directory)** : chaque entrée de l'annuaire comporte plusieurs caractéristiques (ex: nom, email , … pour une Personne) et il est possible d'effectuer des recherches portant sur ces différents critères.

Ex: annuaire LDAP (employés , ...)

Recherche (via JNDI) de ressources enregistrées avec des noms logiques





Paramétrage du « Initial Context » de JNDI

De façon à préciser les ***protocoles*** & ***adresses des serveurs de noms*** que les couches basses de JNDI doivent utiliser, il faut renseigner quelques ***propriétés systèmes***:

Paramétrage généralement effectué au sein de ***fichiers ".properties"*** *:*

(ex1: jndi.properties *[pour rmi-over-iiop / tnameserv]*):

```
java.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
java.naming.provider.url=iiop://servnom:900
```

(ex2: JbossClientJndi.properties):

```
java.naming.factory.initial=
            org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming.client
```

XIX - Annexe: tests sans serveur, aspects divers

1. Tests d'EJB sans serveur

Certains **conteneurs d'EJB3** (Jboss, Jonas, OpenEjb) existent en version "**embbeded**" et peuvent alors être embarqués et utilisés dans une simple JVM java (ex: IDE Eclipse ou maven).
Intérêts:

- possibilité d'intégrer le conteneur "Embedded_EJB3" dans le conteneur Web et le tout fonctionne dans un simple "tomcat"
- possibilité d'effectuer des tests unitaires (via JUnit) directement dans eclipse ou maven (sans avoir besoin de démarrer un gros serveur d'application)

Attention:

- La version "Embedded_Ejb3" de Jboss évolue sans cesse et n'est pas très bien documentée.
- La version "Embedded" de "**OpenEJB**" est bien plus pratique à utiliser durant la phase des tests unitaires (ex: Junit / maven)

1.1. Exemple avec OpenEjb + maven + Junit

Extraits de pom.xml (maven)

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
 <modelVersion>4.0.0</modelVersion>  <parent> ....   </parent>
 <groupId>com.mycompany.jee5app1</groupId>  <artifactId>my-jee5app1-ejb</artifactId>
 <packaging>ejb</packaging>  <version>1.0-SNAPSHOT</version>
 <name>my-jee5app1-ejb Maven JEE5 EJB</name>

 <properties>
       <!-- valeur (sans profile test) pour le fonctionnement dans JBoss -->
       <persistence.datasource>java:/produits_db_DS</persistence.datasource>
       <persistence.provider></persistence.provider>
 </properties>

 <dependencies>
  <dependency>
    <groupId>org.apache.openejb</groupId>
    <artifactId>openejb-core</artifactId>
    <version>3.1.4</version>
    <scope>test</scope>
    <exclusions>
      <exclusion>
        <groupId>org.apache.openjpa</groupId>
        <artifactId>openjpa</artifactId>
      </exclusion>
    </exclusions>
  </dependency>

 <dependency>
    <groupId>javaee</groupId>
    <artifactId>javaee-api</artifactId>
    <version>5</version>
    <scope>provided</scope>
  </dependency>
```

```
....
...
<plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>     <!-- plugin de test  (maven) -->
        <configuration>
            <!-- skip test à true par défaut (pour Jboss ou ...) et à false dans profile test -->
            <skip>true</skip> <!--  équivalent à mvn -Dmaven.test.skip=true -->
        </configuration>
      </plugin>
   <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-ejb-plugin</artifactId>
      <configuration>
       <ejbVersion>3.0</ejbVersion>
      </configuration>
     </plugin>
....
     <resources>
      <resource>
                <directory>src/main/resources</directory>
                <filtering>true</filtering>
       </resource>
   </resources>
....
<profiles>
   <profile>
    <id>test</id>
    <properties>
      <!-- valeur (avec profile test) pour le fonctionnement avec Embedded OpenEJB -->
      <persistence.datasource>produits_db_TestDS</persistence.datasource>
      <persistence.provider><!
[CDATA[<provider>org.hibernate.ejb.HibernatePersistence</provider>]]></persistence.provider>
    </properties>
    <build>
      <plugins>
           <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
              <configuration>
              <skip>false</skip>
            </configuration>
           </plugin>
      </plugins>
    </build>
   </profile>
  </profiles>
..
</project>
```

```
mvn  -P test clean test   > ..\resBuild.txt
```

(avec -P nomDuProfile) .

src/main/resources/META-INF/persistence.xml

```
<persistence ....>
        <persistence-unit name="myPersistenceUnit" >
        <jta-data-source>${persistence.datasource}</jta-data-source>
    <!-- "produits_db_DS" dans JBOSS_HOME/server/default/deploy/produits_db_DS.xml -->
         ${persistence.provider}
```

```
        <properties>
          <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
        </properties>
        </persistence-unit>
</persistence>
```

NB: le filtrage de ressources de maven remplacera ${persistence.datasource} et ${persistence.provider} par les propriétés maven de mêmes noms dont les valeurs dépendent du profil activé (par défaut ou bien test) .

Test "Junit" pour Embedded OpenEjb:

```
...
public class TestGestionProduits {
private static Context context; // jndi context for open-ejb

@BeforeClass
 public static void initializeEmbeddedContainer() throws Exception {
  Properties properties = new Properties();
  properties.setProperty(Context.INITIAL_CONTEXT_FACTORY,
     "org.apache.openejb.client.LocalInitialContextFactory");
  properties.put("produits_db_TestDS", "new://Resource?type=DataSource");
  properties.put("produits_db_TestDS.JdbcDriver", "com.mysql.jdbc.Driver");
  properties.put("produits_db_TestDS.JdbcUrl", "jdbc:mysql://localhost/produits_db");
  properties.put("produits_db_TestDS.username", "root");
  properties.put("produits_db_TestDS.password", "root");
  //properties.put("produits_db_TestDS.JtaManaged", "false");
  properties.put("openejb.embedded.initialcontext.close", "destroy");
  context = new InitialContext(properties);
 }

 private GestionProduitsLocal service = null; // service métier  (ejb) à tester
 //NB: en testant les méthodes de l'EJB session , on teste également
 //si JPA fonctionne bien en arrière plan

   @Before  //ou bien constructeur
  public void initService(){
   if(service==null){
    try{  String openEjbJndiName="GestionProduitsBean" + "Local";
          service= (GestionProduitsLocal )
         context.lookup(openEjbJndiName);
              }catch(Exception ex){ex.printStackTrace();}
     }
   }

 @Test
 public void test_getListCategories() {
       List<String> listeCategories = service.getListCategories();
       TestCase.assertTrue(listeCategories.size()>0);
       System.out.println("liste des categories -->");
       for(String c : listeCategories)  System.out.println(c);
       }
}
```

1.2. Exemple 2 (sans maven , compatible TomEE) :

MyAbstractOpenEjbTest.java

```
package test.service;

import java.util.Properties; import org.junit.BeforeClass;
import javax.naming.Context;  import javax.naming.InitialContext;

public abstract class MyAbstractOpenEjbTest {
        protected static Context context; // jndi context for open-ejb
        private static final String TEST_OPENEJB_DS="java:openejb/Resource/XyDS";//"XyDS";

        @BeforeClass
        public static void initializeEmbeddedContainer() throws Exception {
           Properties properties = new Properties();
           properties.setProperty(Context.INITIAL_CONTEXT_FACTORY,
              "org.apache.openejb.client.LocalInitialContextFactory");

           properties.put(TEST_OPENEJB_DS, "new://Resource?type=DataSource");

           properties.put(TEST_OPENEJB_DS+".JdbcDriver", "com.mysql.jdbc.Driver");
           properties
              .put(TEST_OPENEJB_DS+".JdbcUrl", "jdbc:mysql://localhost:3306/xy");
           properties.put(TEST_OPENEJB_DS+".username", "root");
           properties.put(TEST_OPENEJB_DS+".password", "afcepf");
           //properties.put("produits_db_TestDS.JtaManaged", "false");

           properties.put("openejb.embedded.initialcontext.close", "destroy");
           context = new InitialContext(properties);
          }

          public abstract void initService();
}
```

TestEjbServiceXySansServeur.java

```
package test.service;

import org.junit.Assert;    import org.junit.Before;   import org.junit.Test;
import tp.dto.Xy;    import tp.service.ServiceXy;

public class TestEjbServiceXySansServeur extends MyAbstractOpenEjbTest {
        private ServiceXy service;      //service (ejb) a  tester

         @Before
           public void initService(){
            if(service==null){
             try{
               String openEjbJndiName="ServiceXyImpl" + "Local";
               //String openEjbJndiName="ServiceXyImpl" + "Remote";
               service= (ServiceXy )
                   context.lookup(openEjbJndiName);
                        }catch(Exception ex){ex.printStackTrace();}
              }
          }

        @Test
        public void testRechercherXyByNum(){
                try {
                        Xy objXy = service.rechercherXyByNum(1L);
                        System.out.println("objXY=" + objXy);
                        Assert.assertTrue(objXy.getNum() == 1L);
                } catch (Exception e) {
                        e.printStackTrace();
                        Assert.fail(e.getMessage());
                }
        }
}
```

2. EJB3 avec TomEE (Tomcat EE)

2.1. Présentation de TomEE

TomEE (Tomcat EE) est une version de Tomcat qui est packagée avec tout un tas de **librairies (.jar)** permettant de prendre en charge des **EJB3** , des **services web** (soap et/ou REST) et des injections de dépendances selon la norme **CDI** de JEE6 .

TomEE est un serveur d'application qui respecte une grande partie des spécifications **JEE6** (et qui est "certifié" sur ce point).

TomEE est entièrement implémenté avec des technologies de "**Apache** group" :

- La partie JPA2 est prise en charge par **OpenJPA**
- La partie EJB3 est implémentée via **OpenEJB**
- La partie Web-service (JAX-WS et JAX-RS) est gérée par **CXF**
- La partie CDI est prise en charge par **OpenWebBean**
- La partie JSF2 est implémentée via **MyFaces**

Il existe également une version "**-plus**" comporte une implémentation de la partie "EJB MDB" en s'appuyant sur activeMQ.

2.2. Configuration d'un "dataSource" pour TomEE

Le moyen le plus simple de configurer un "dataSource" qui sera pris en charge par TomEE (dans le cadre d'une prise en charge d'application JEE) consiste à embarquer (dans l'application à déployer) un fichier **WEB-INF/resources.xml** ayant la structure suivante :

```
<resources>
<Resource id="XyDS" type="DataSource">
   JdbcDriver  com.mysql.jdbc.Driver
   JdbcUrl     jdbc:mysql://localhost/xy
   UserName    root
   Password    mypwd
</Resource>
</resources>
```

NB: il faudra penser à recopier ***mysql-connector-java-5.1.31.jar*** dans le répertoire "**lib**" de **TomEE**.

Nom JNDI à configurer dans (src ou src/main/resources) **META-INF/persistence.xml**

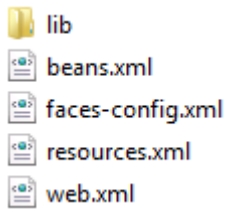
<jta-data-source>**java:openejb/Resource/*XyDS***</jta-data-source>

2.3. Structure d'une application avec TomEE comme cible

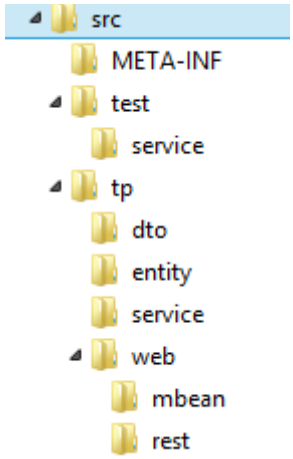
L'application à déployée doit être structurée comme une "dynamic web application" (**.war**) comportant éventuellement des sous-parties "xy.jar" dans **WEB-INF/lib** .

==> pas d'ear (et pas de sous projet "ejb" imposé).

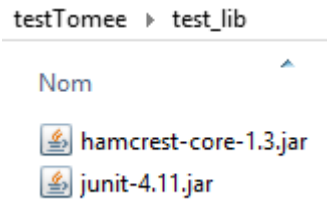
Contenu typique de WEB-INF (dans WebContent ou src/main/webapp) :



Contenu typique de src :

	persistence.xml Tests_unitaires_avec_openEjb_et_dataSource_sans_serveur entités persistantes services métiers en tant que "stateless" ManagedBean pour JSF Adaptateur "REST / JAX-RS" pour communication avec Angular-Js

Librairies (non fournies par TomEE) à prévoir pour les tests unitaires :



Tout le reste est fourni par TomEE (et peut être en quasi-totalité être en mode "provided" au sein d'une configuration maven).

XX - Annexe – Bibliographie, Liens WEB + TP

1. Bibliographie et liens vers sites "internet"

http://www.oracle.com/technetwork/java/javaee (anciennement sun)	La Référence (**spécifications** , exemples , ...)
http://www.oracle.com/technetwork/java (anciennement www.javasoft.com)	Site de référence sur le langage JAVA
www.jboss.org	Site pour accéder au serveur Jboss (download, documentation, exemples , ...)
http://wiki.jonas.ow2.org	Site pour accéder au serveur Jonas (OW2)
http://www.application-servers.com/	Site de news sur les serveurs d'applications et les technologies java/jee/web .
http://www.eyrolles.com/Informatique/	Site pour trouver des livres informatiques

2. TP

Préliminaires:

- Installer si besoin le jdk 1.6 et le serveur de base de données MySql 5
- Installer le serveur Jboss_5.1 (en version compatible avec le jdk 6)
- Lancer la commande Jboss5.1/bin/run.bat et vérifier le bon fonctionnement de Jboss via l'URL http://localhost:8080
- Installer si besoin eclispe (3.5 ou 3.6) avec les plugins intégrés pour JEE .
- Configurer eclipse pour qu'il utilise le jdk 1.6 et Jboss 5.1 (window/preferences/Server/Runtimes env, ...)
- Créer une nouvelle application de type "JEE/Enterprise Application Project" de nom "myJeeApp" (ou autre) et comportant des sous projet "EJB" , "Web" et "Client" .

Succession de Tp (progressifs):

- Ejb session stateless de type "calcul" (ex: calculTva ou calculatrice ou ....) puis tests/invocation via "client externe distant" et "client web" .
- Éventuel ajout de @WebService et test via soapUi
- Créer une petite base de données [ex: minibankDB avec une table Compte(numCpt,label,solde,numClient) ou bien deviseDB avec une table Devise(monnaie,dChange)] puis paramétrer un pool de connexions JDBC vers cette base.
- Coder ensuite un ejb session stateless "gestionComptes" ou "gestionDevises" qui va (dans une première version) directement accéder à la base via @Resource / DataSource JDBC.
- Basculer ensuite sur une implémentation basée sur JPA et (re-)tester
- Bien tester l'aspect transactionnel (par exemple en essayant de transférer un montant d'un compte à débiter valider vers un compte à créditer invalide et vérifier le "rollback").
- Coder éventuellement un "ejb session à état" (ex: Caddy) + test
- Coder éventuellement un "ejb MDB" + test via un client JMS
- coder et tester la sécurité JEE (Roles , Realm , ...)
- ....