

EJB 3

*(avec JPA,
RMI et JMS)*

Table des matières

I - Présentation (EJB 3 et JEE).....	7
1.EJB (Enterprise JavaBean) - présentation.....	7
1. Fabrication automatique.....	10
2. Persistance des EJB "entity".....	11
3. Grands traits des EJB3.....	11
3.1. Terminologie.....	11
3.2. Catégorisation via annotations java5.....	12
3.3. Un petit exemple (très simple):.....	13
4. JEE en tant qu'ensemble d'API & conteneur JEE.....	14
5. Structure d'un serveur JEE.....	14
6. Aspects pratiques (projets dans l'IDE eclipse).....	16
6.1. Types des projets eclipse.....	16
6.2. Relations entre les projets d'une même application JEE.....	17
6.3. Lancement des tests (sous eclipse).....	17

6.4. Paramétrage du serveur Jboss (sous eclipse).....	17
7. Un Serveur JEE fonctionne avec une "JVM".....	18
8. Utilisation de "Spring" à la place des EJB.....	20
9. Evolution de JEE.....	21
10. Principaux serveurs d'applications (JEE).....	22
11. Ressources propriétaires et références standards.....	22
12. Descripteur de déploiement de l'application.....	24
13. Eventuel fichier "META-INF/ejb-jar.xml".....	25

II - EJB session sans état (et invocations).....26

1. EJB Session sans état apparent (stateless).....	26
1.1. cycle de vie d'un EJB3 session sans état.....	26
2. Conventions de noms.....	28
3. Client distant et externe vis à vis d'un EJB3.....	29
4. Client Web/J2EE vis à vis d'un EJB3.....	30
5. Références de ressources.....	30
5.1. Référence vers ejb (depuis client web).....	31
6. Liens entre différents EJB.....	31

III - EJB vu/invoqué comme un service web.....32

1. EJB3 session sans état vu comme un service WEB.....	32
1.1. Paramétrage de l'aspect "service web" d'un EJB Session	32
1.2. Client externe JAX-WS (java6).....	33

IV - Source de données JDBC (et EJB).....35

1. Sources de données JDBC.....	35
1.1. Api JDBC (Java DataBase Connectivity).....	35
1.2. Pool de connexions et DataSource.....	36
2. Architecture JCA (Connecteurs).....	38
3. Accès à un DataSource JDBC depuis un Ejb session.....	39
3.1. Récupération d'une source de données via recherche JNDI.....	39
3.2. injection directe d'une dépendance vers une ressource	39
4. éventuelle utilisation coté web (DataSource).....	39
5. Configuration d'une source de données (pool).....	40

V - Ejb "Entity" & JPA (présentation).....41

1. Problématique "O.R.M.".....	41
1.1. Objectif & contraintes:.....	41
1.2. Eléments techniques devant être bien gérés.....	42
2. JPA (Java Persistence Api).....	42

VI - JPA : architecture & configuration.....	44
1. Présentation de JPA (Java Persistence Api).....	44
2. Unité de Persistance (configuration+packaging).....	45
3. Unité de Persistance (META-INF/persistence.xml).....	46
4. Configuration du mapping JPA via annotations.....	48
4.1. Exemple d'entité persistante (@Entity).....	48
VII - EntityManager et entités persistantes.....	50
1. Entity Manager et son contexte de persistance.....	50
2. Transaction JPA.....	52
3. Différents états - objet potentiellement persistant.....	53
4. Cycle de vie d'un objet JPA/Hibernate.....	53
5. Synchronisation automatique dans l'état persistant.....	54
6. objet persistant et architecture n-tiers.....	54
7. Principales méthodes JPA / EntityManager et Query.....	55
8. Contexte de persistance et proxy-ing	56
VIII - Langage de requêtes JPQL.....	58
1.1. Présentation générale de HQL et JPQL.....	58
1.2. Principaux éléments de syntaxe de JPQL et HQL.....	58
1.3. Quelques exemples simples de requêtes JPQL/HQL:.....	59
1.4. Lancement d'une requête JPQL (JPA).....	59
IX - O.R.M. JPA (généralités).....	61
1. Vue d'ensemble sur les entités, valeurs et relations.....	61
1.1. Problématique.....	61
1.2. Entités	61
1.3. Objet "Valeurs" ou bien "imbriqué/incorporé (embedded)"	62
1.4. Collections et constituants.....	63
1.5. FetchType (lazy & eager).....	64
1.6. Operations en cascade.....	66
2. Identité d'une entité (clef primaire et générateurs).....	67
3. Clef primaire composée.....	68
4. Structure générale d'une classe d'entité persistante	69
5. Propriétés d'une colonne.....	71
6. Relations (1-1, n-1, 1-n et n-n).....	71
X - O.R.M. JPA – détails (1-n , 1-1 , n-n, ...).....	72
1.1. Relations 1-n et n-1 (entité-entités).....	72
1.2. Relation 1-1.....	73

1.3. Sous objets imbriquables au sein d'une entité.....	74
1.4. Relation n-n.....	75

XI - Gestion des transactions (niveau EJB).....78

1. Transactions distribuées et commit à 2 phases.....	78
1.1. Qualités (A.C.I.D.) d'une transaction distribuée basique.....	78
1.2. Protocole XA pour le commit à 2 phases.....	79
2. Infrastructure transactionnelle de JEE.....	80
3. Gestion déclarative des transactions.....	81
4. Propagation du contexte transactionnel.....	82
5. Effets du contexte transactionnel sur les EJB.....	83
5.1. Annulation implicite d'une transaction en cas d'exception.....	83
5.2. Annulation explicite d'une transaction :.....	84
6. Gestion déclarative des transactions / EJB3.....	84
6.1. Attributs transactionnels sur EJB (approche déclarative).....	84
6.2. Annotations sur EJB3 concernant les transactions.....	84

XII - EJB session à état (Stateful).....85

1. EJB Session à état conversationnel (stateful).....	85
1.1. cycle de vie d'un EJB3 session à état.....	85
1.2. Initialisation et terminaison explicites (@Init et @Remove).....	86

XIII - EJB "M.D.B." et invocation asynchrone.....87

1. Présentation de JMS	87
2. Configuration d'une file d'attente au sein de JBoss5.....	88
3. EJB3 de type MDB.....	89
3.1. cycle de vie d'un EJB3 MDB.....	89
3.2. Annotations et interfaces pour EJB3 de type MDB.....	90
3.3. Exemple d' EJB3 de type MDB.....	90
3.4. client (externe) de test (envoyant un message dans une file).....	91

XIV - Sécurité JEE (au niveau des EJB).....93

1. D.M.Z. et Firewalls.....	93
2. Sécurité J2EE/JEE5.....	93
3. Rôles associés aux EJB3 (via annotations).....	95
4. Domaine de sécurité (Jboss).....	96
5. Tests.....	97
5.1. Via une authentification coté "Web".....	97
5.2. Depuis un client externe (via Jaas et Jndi).....	97

XV - Aspects divers (timer, aop, ...)	100
1. Timer sur EJB3 (déclenchement différé)	100
2. Intercepteurs pour EJB3 / extension AOP	101
XVI - Essentiel RMI (Remote Method Invocation)	103
1. Principes "RPC" (Remote Procedure Call)	103
2. Principe général des RPC	104
3. Localisation transparente / serveur de noms	105
4. Vue globale sur RPC orienté objet	106
5. Protocoles et API pour RPC objets synchrones	106
6. Présentation api RMI (Remote Method Invocation)	107
6.1. Structure du code java (RMI)	107
6.2. Code type de l'interface de l'objet distant (RMI):	108
6.3. Passage d'objet en paramètre d'un appel distant RMI:	108
7. Code type de l'application cliente (Rmi-JRMP)	109
8. Code type de l'application serveur:	110
9. Mise en oeuvre standard (de la rigueur s'impose)	111
9.1. Scripts ".bat"	111
9.2. Script ANT (build.xml)	112
9.3. Explications (points clefs):	113
10. Sécurité avec RMI	114
11. RMI over IIOP	114
11.1. Fragment de code du client:	115
11.2. Fragment de code du serveur:	115
Essentiel JMS (Java Message Service)	116
12. Queue & Topic	116
13. Exemples (fragments) de code	117
13.1. Obtention de l'objet ConnectionFactory via JNDI:	117
13.2. Obtention d'une file de message via JNDI:	117
13.3. Création d'un objet Connection via l'usine:	117
13.4. Création d'une Session à partir de l'objet Connexion:	117
13.5. Obtention de l'objet "MessageProducer" pour envois	118
13.6. Obtention de l'objet "MessageConsumer" pour réceptions	118
13.7. Déclencher le début possible des réceptions de messages:	118
13.8. Création de messages	118
13.9. Envoi et réception	119
13.10. Extraction des valeurs d'un message	119
13.11. Filtrage éventuel des messages que l'on souhaite récupérer:	120
14. Champs des entêtes de message	120
15. Acquiescement des messages reçus	121

16. Liste des principaux "Provider JMS"	121
---	-----

XVII - Annexe – Détails sur JPA.....	122
---	------------

1. Relations d'héritage & polymorphisme.....	122
1.1. stratégie "une seule table par hiérarchie de classes"	122
1.2. Stratégie "Join_inheritance".....	124
1.3. Stratégie (assez rare) "une table par classe"	125
1.4. Polymorphisme.....	125
1.5. Une entité répartie dans 2 tables (principale,secondaire).....	126
1.6. Eléments requis sur une classe d'entité (JPA):.....	126
1.7. Verrous (optimistes et pessimistes).....	127
2. Cycle de persistance (pour @Entity) et annotations/callbacks associées pour "Listener"	128

Annexe - Essentiel JNDI	129
--------------------------------------	------------

3. JNDI (pour se connecter à un EJB ou ...).....	129
--	-----

XVIII - Annexe: tests sans serveur, aspects divers.....	131
--	------------

1. Tests d'EJB sans serveur.....	131
1.1. Exemple avec OpenEjb + maven + Junit	131

XIX - Annexe – Bibliographie, Liens WEB + TP.....	134
--	------------

1. Bibliographie et liens vers sites "internet"	134
2. TP.....	134

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, ...)

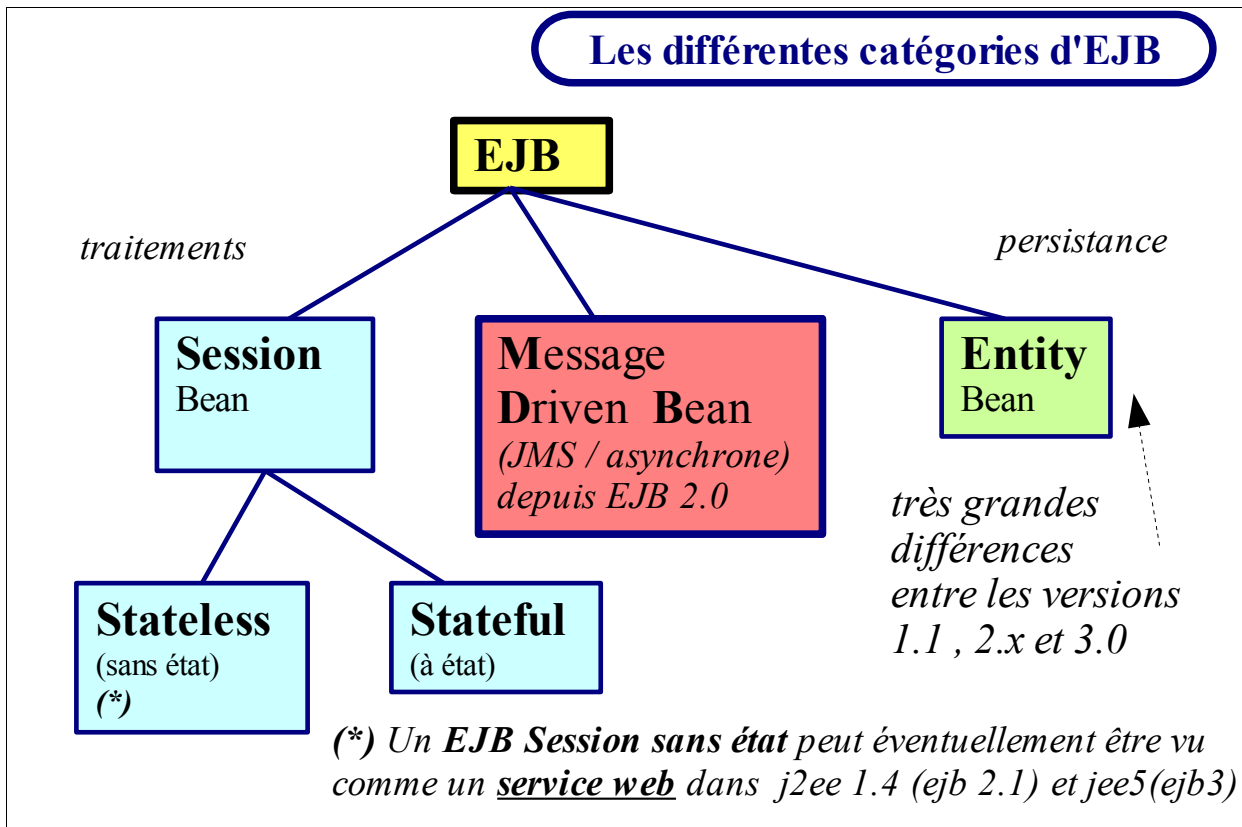
Présentation des EJB

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":

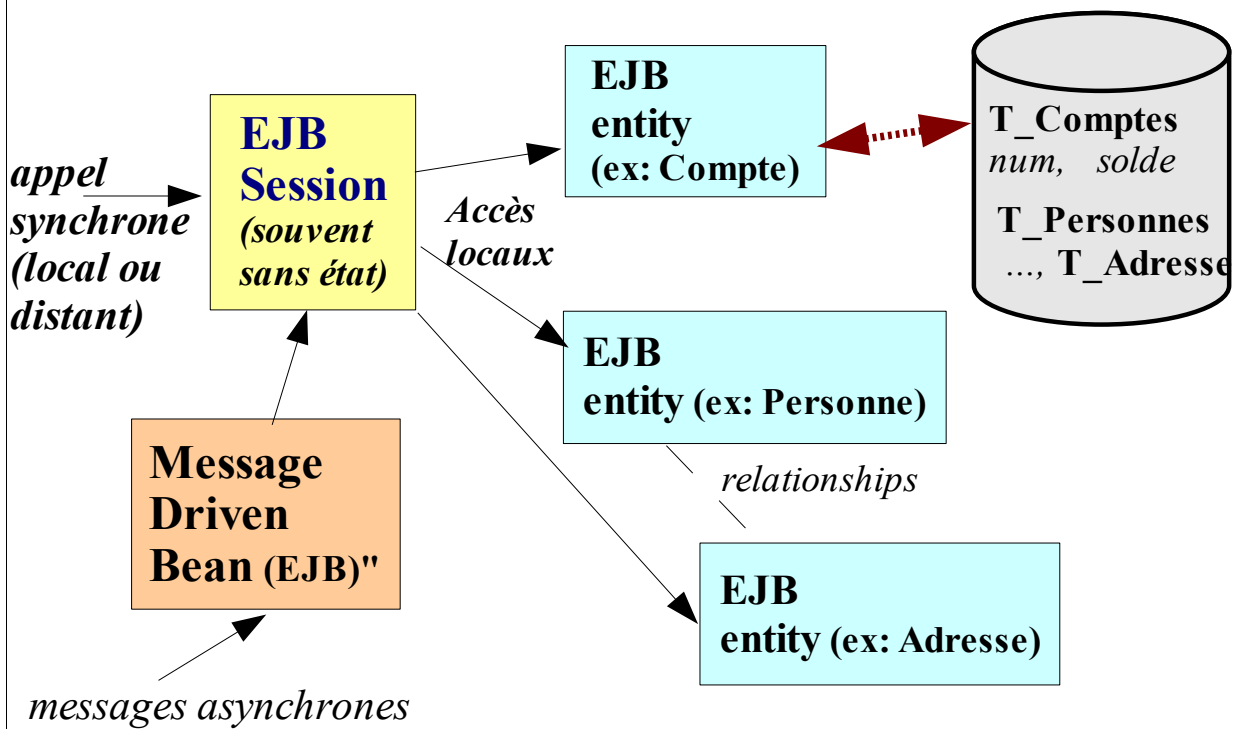
-ne programmer que l'aspect "métier" des composants.
le serveur d'application (avec son container d'EJB) **prend en charge les aspects techniques** (sécurité, multi-tâches, transactions).



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)

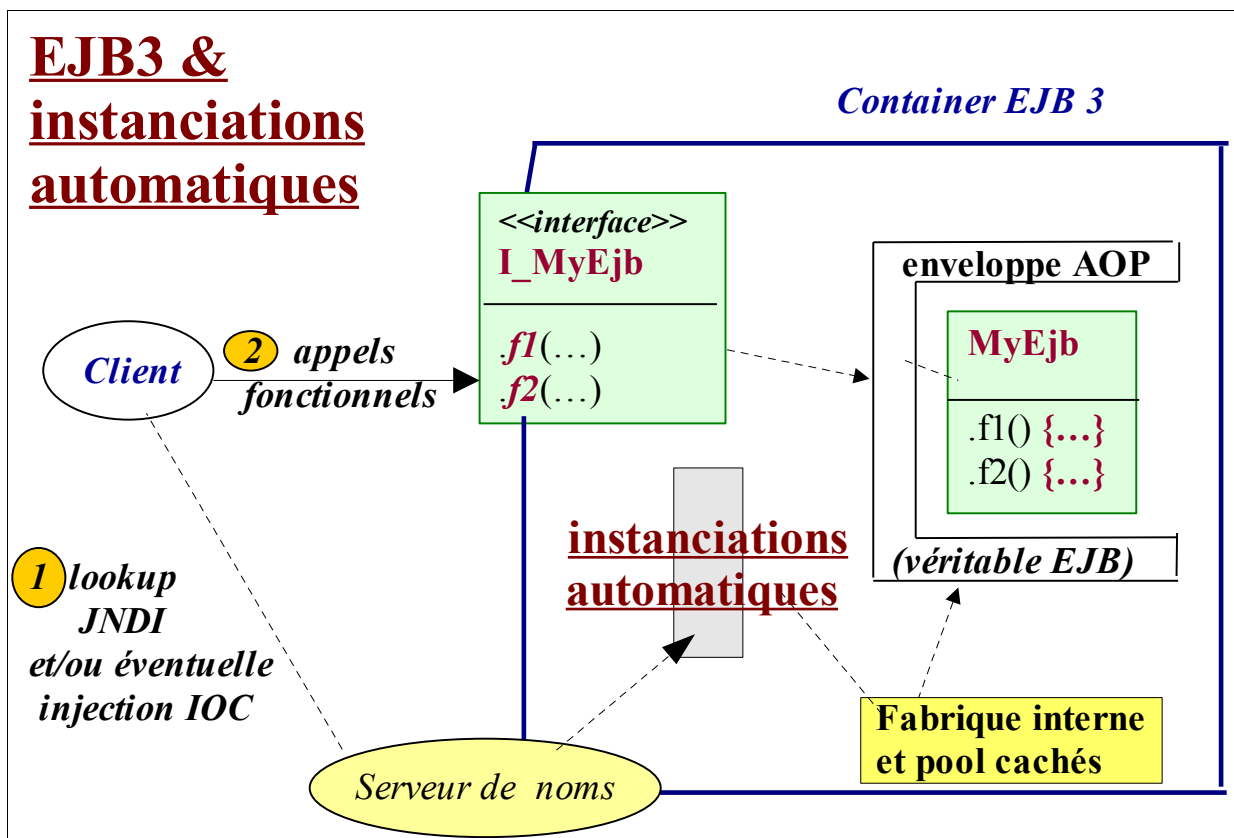
- (Ejb Session) stateless :
 - sans état, aucune donnée métier n'est conservée dans la mémoire de l'EJB entre deux appels (successifs) émanant d'un même client .
 - sert à factoriser des traitements atomiques entre les clients . Tous les paramètres nécessaires doivent être précisés d'un coup lors d'un 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 Java5.

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**.

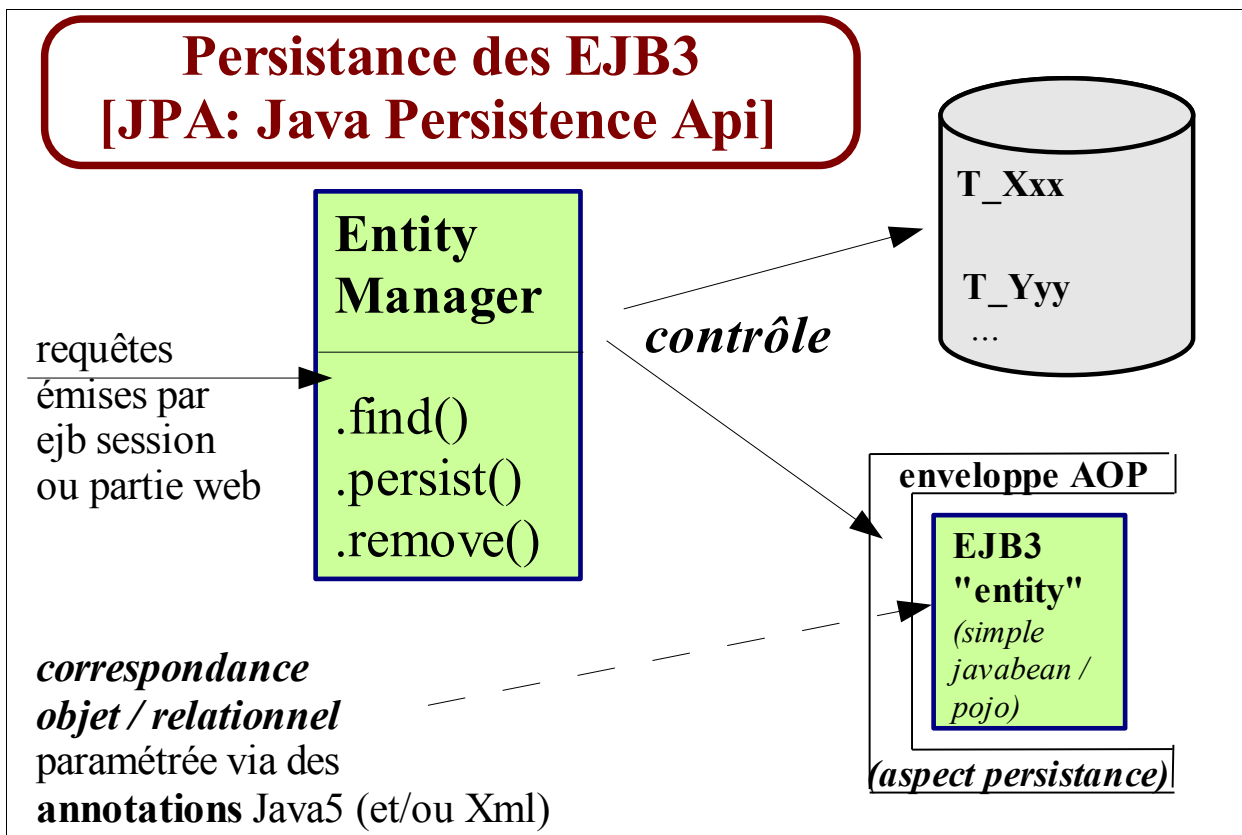
Les EJB entités ont été complètement chamboulés ==> nouvelle Api **JPA** (Java Persistence Api) très proche de Hibernate .



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. Persistence 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 (<i>lié à JPA et contrôlé par PersistenceManager</i>)
@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 conséquent, 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 différencier 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");

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

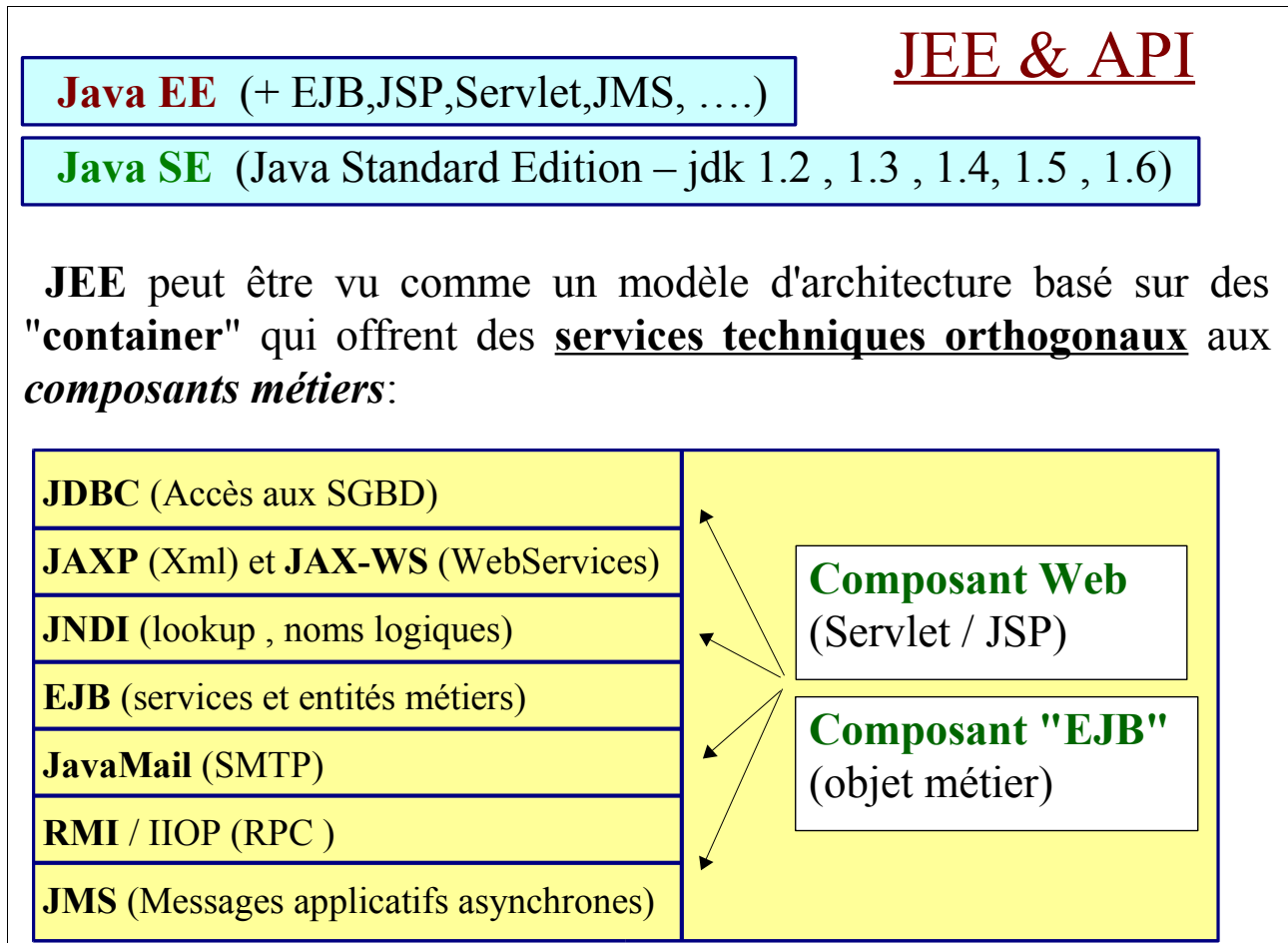
NB:

- Bien qu'ayant la sémantique d'un appel distant , l'interface de l'EJB (vue cliente) est une simple interface locale .
- Un lookup suivi d'un casting Java suffit (plus besoin de PortableRemoteObject.narrow() pour les EJB3).

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" .



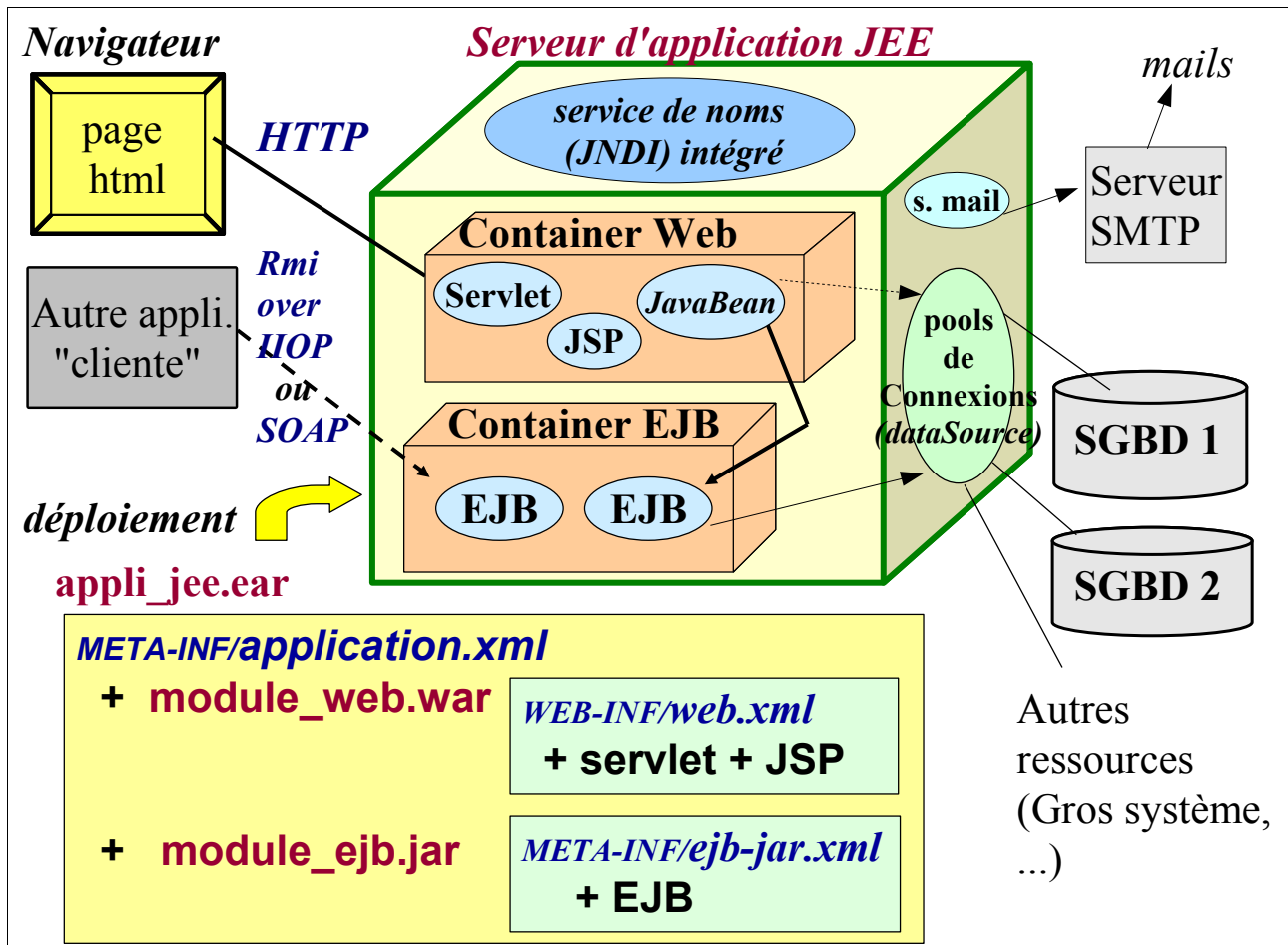
NB: La nouvelle version de J2EE est JEE5 , il vaut mieux utiliser maintenant le terme générique "JEE" pour s'adapter aux évolutions récentes.

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 "conteneur" : 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** la nouvelle version des spécifications de sa plate-forme Java de niveau entreprise .

Les principaux apports de cette nouvelle version sont les suivants:

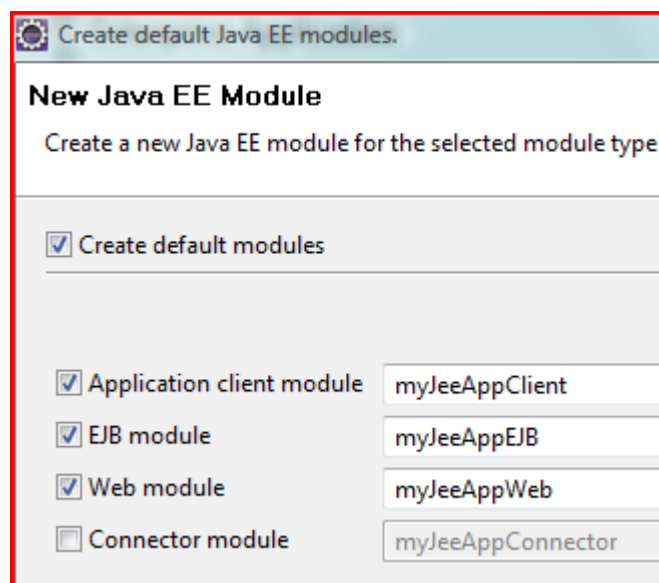
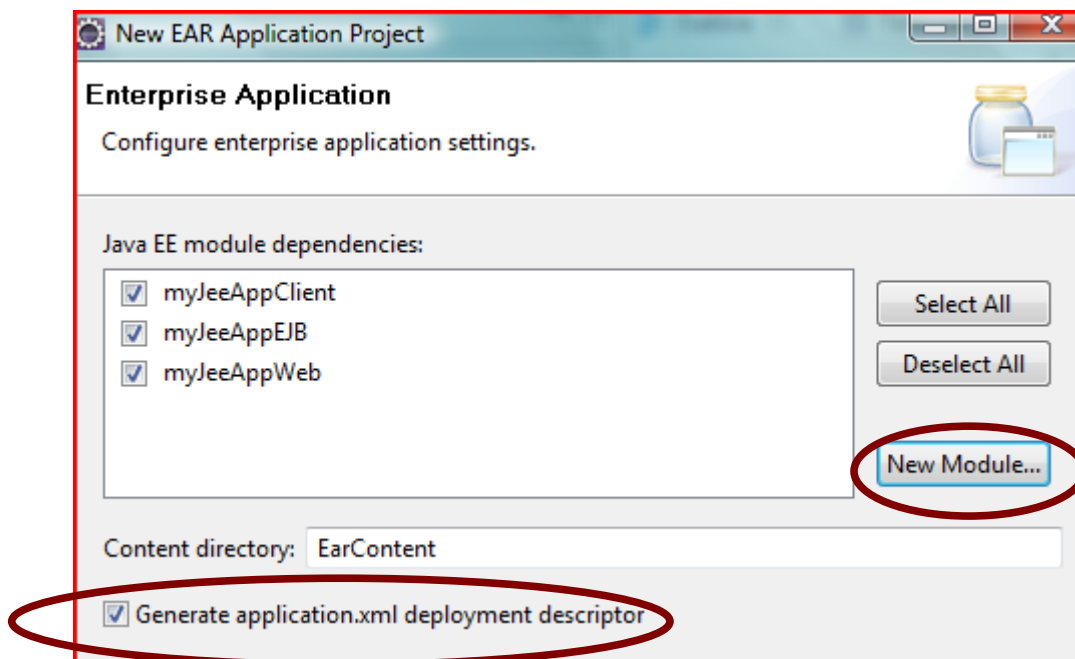
- **EJB3** (avec api **JPA** pour la persistance des données).
- Nouveau support des **services WEB** via l'api **JAX-WS** (mieux que JAX-RPC)
- Intégration du framework **JSF** dans la partie WEB
- **Partie Web** de niveau "Servlet 1.5 / Jsp 2.1" *supportant l'injection IOC des EJB3 et des ressources JEE5.*

6. Aspects pratiques (projets dans l'IDE eclipse)

6.1. Types des projets eclipse

NB: Avant de créer un projet JEE sous Eclipse, il est très fortement conseillé de paramétrer dans "*Window/préférences/Servers/runtime environnement*" le serveur (Jboss ou ...) qui sera utilisé pour effectuer les tests.

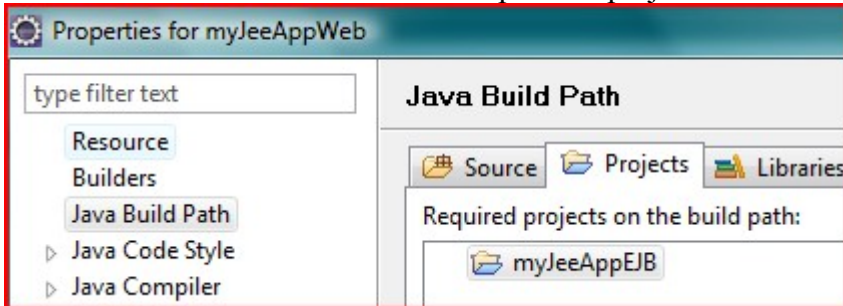
Au sein de l'IDE eclipse , une application complète "JEE" correspond à un projet enveloppe de type "Java EE/**Enterprise Application Project**" référençant divers sous projets de type "Web/**Dynamic Web Project**" ou "EJB/**Ejb Project**".



NB: Le projet "*myJeeAppClient*" correspondra à un projet de test ("prog client" externe à Jboss pour tester les EJB en mode "remote"). Ce projet pourra comporter des classes "*TestXxApp*" avec une méthode "*main()*" à lancer via "**run as /java application**" après que le serveur d'application ait eu le temps de préalablement démarrer.

6.2. Relations entre les projets d'une même application JEE

Pour que le code d'un projet "Web" ou "Client" puisse accéder aux interfaces des EJB situées dans le projet "EJB" on peut éventuellement paramétrer le projet "Web" et/ou "Client" pour qu'il puisse directement accéder aux éléments compilés du projet "EJB":



via Add "myJeeAppEjb project".

Autre solution: recopier les interfaces des EJB d'un projet à l'autre (à refaire si évolutions).

6.3. Lancement des tests (sous eclipse)

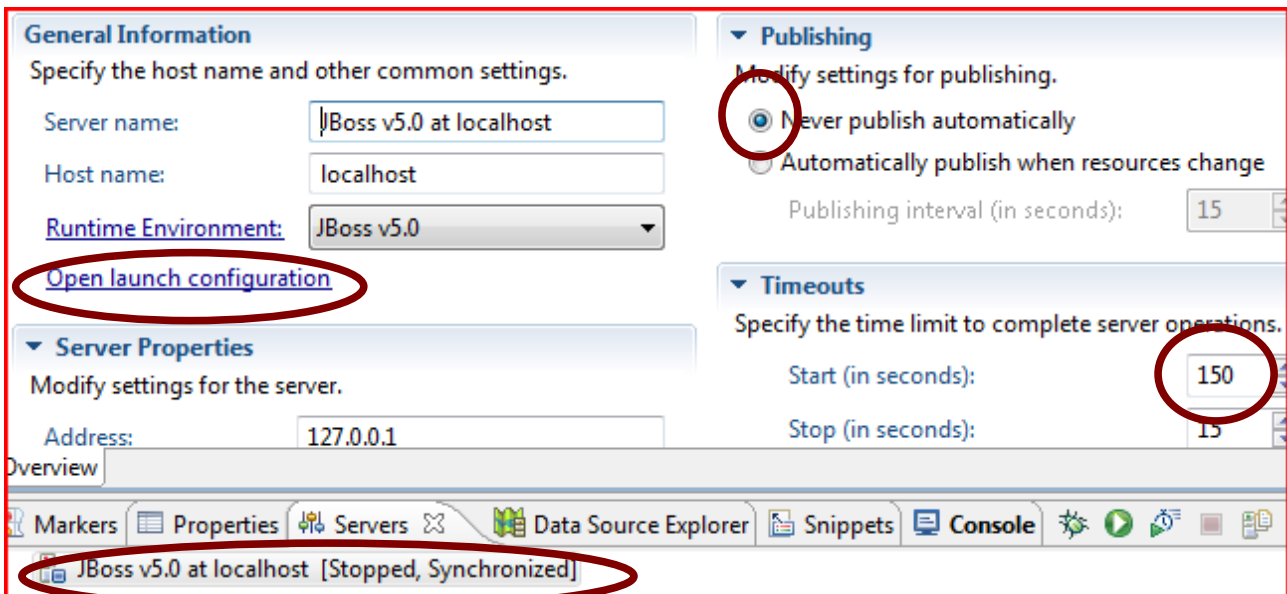
Run As / Run on server depuis:

- page index du projet Web
- (ou bien) projet "ear"

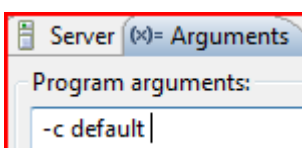
NB: Depuis l'icône du serveur de test (dans la vue ou onglet "server") , on peut déclencher via un click droit :

- start / stop
- **publish** (pour packager et déployer le code de l'application vers le serveur).

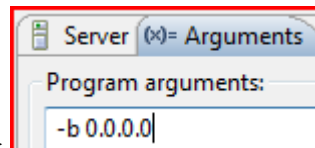
6.4. Paramétrage du serveur Jboss (sous eclipse)



(via click droit /Open sur le serveur JBoss de test) .



==>

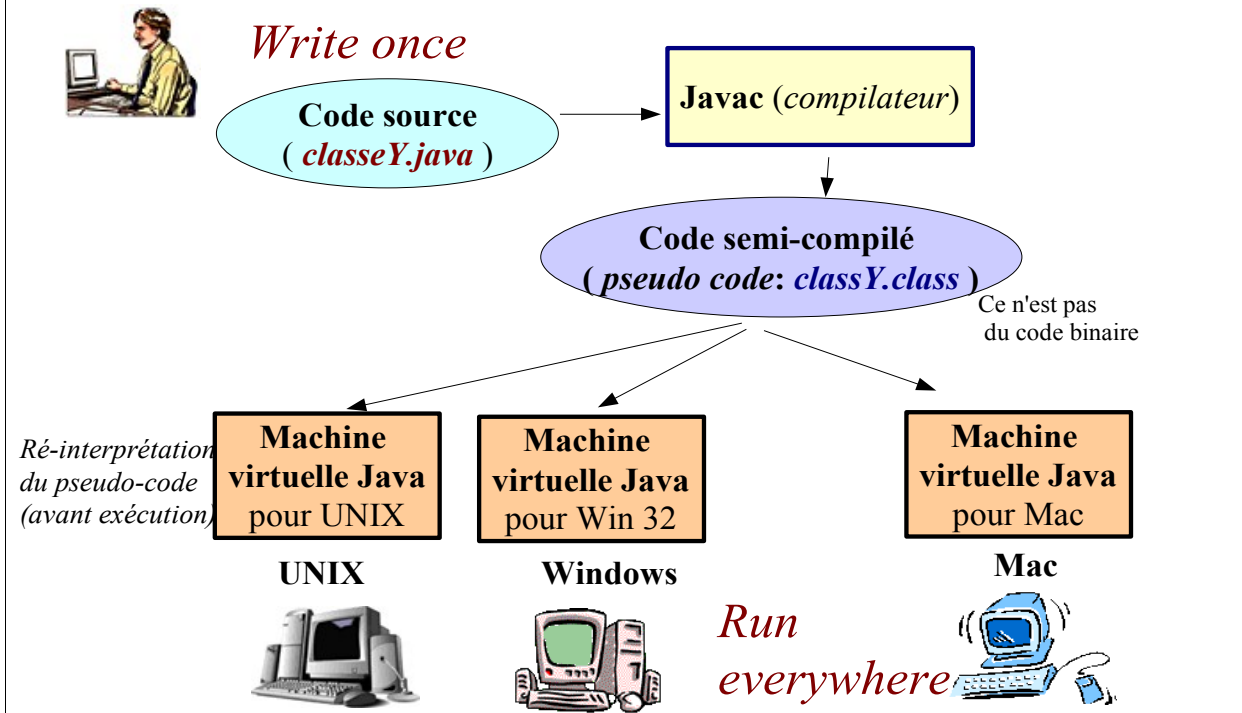


à paramétrer dans "launch configuration"

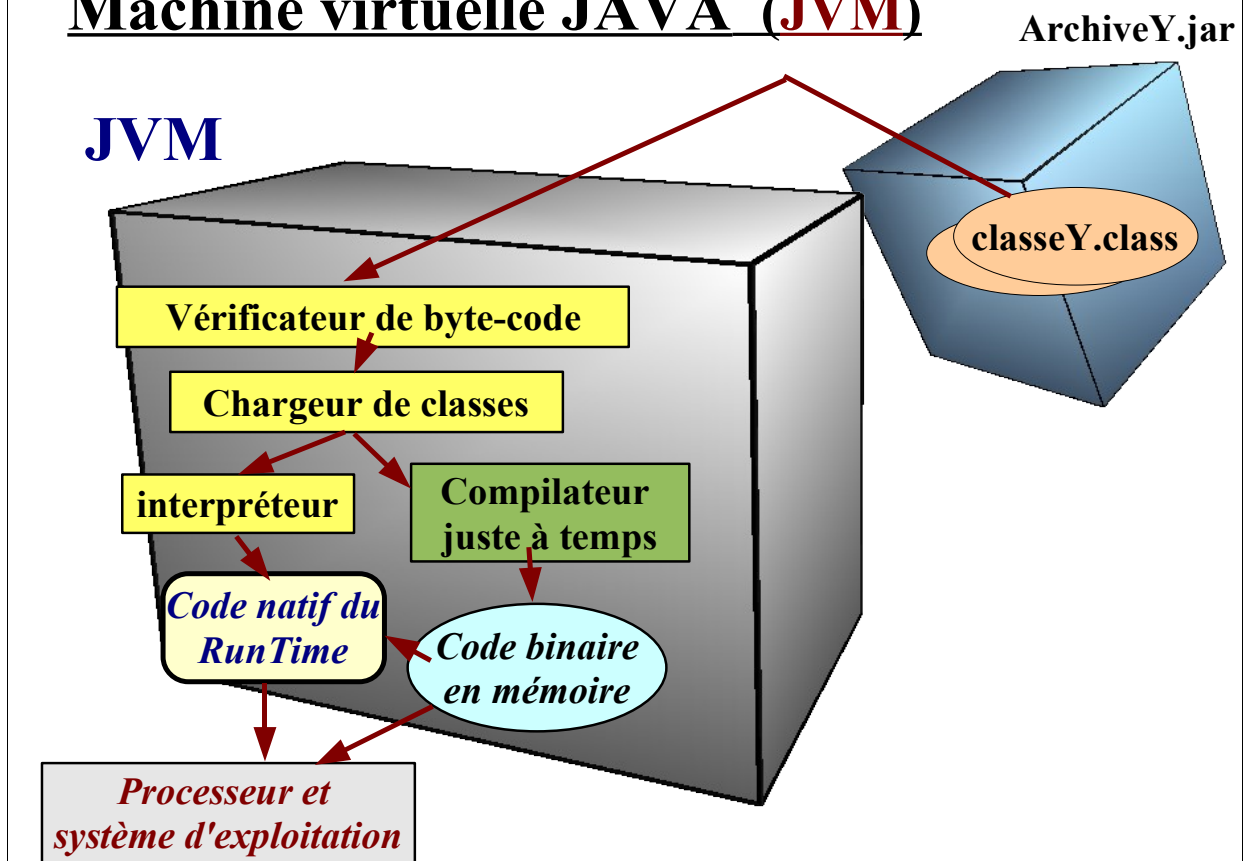
pour permettre (en mode développement non sécurisé) à accès au serveur Jboss depuis d'autre machine que localhost .

7. Un Serveur JEE fonctionne avec une "JVM"

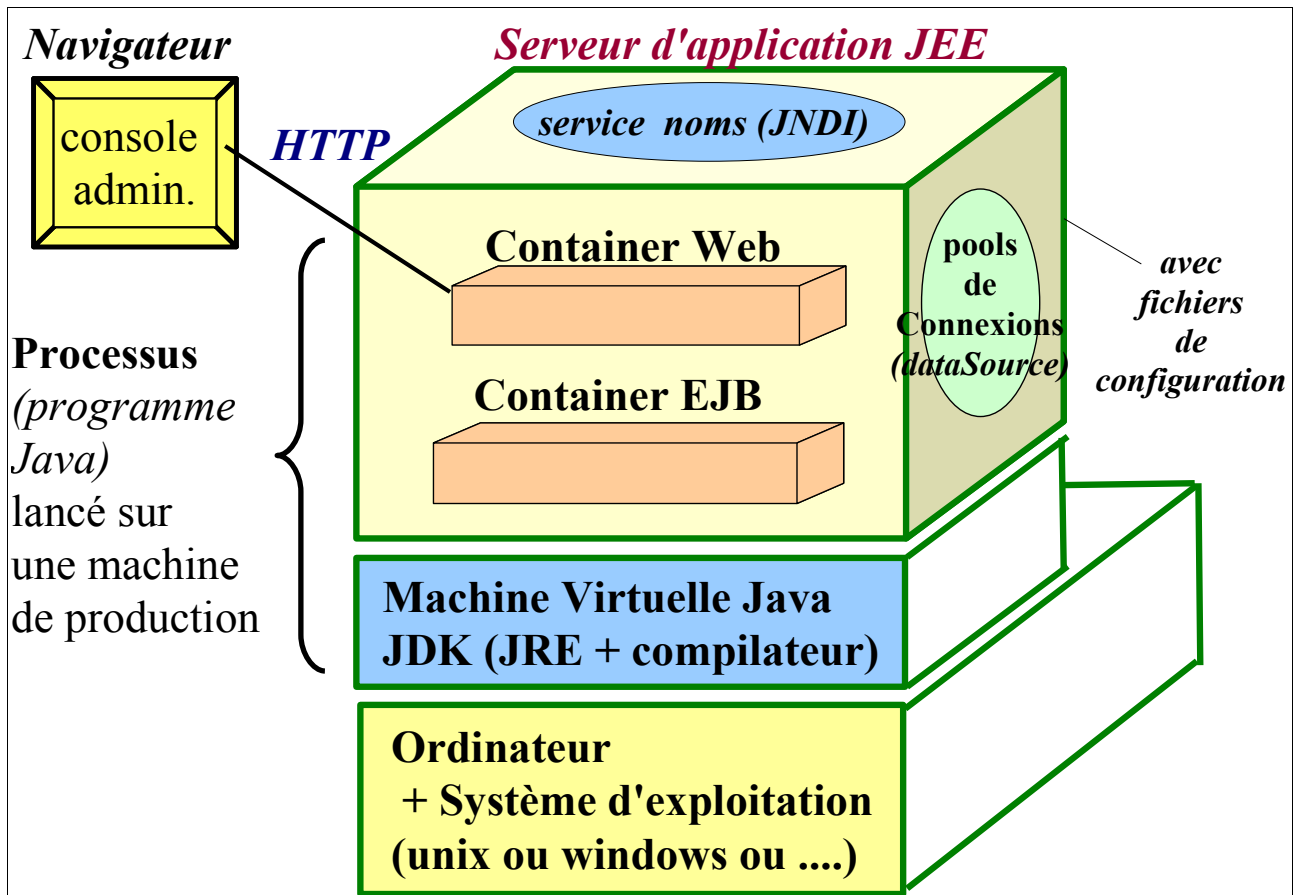
Java: langage *partiellement compilé* et *interprété* (via JVM)



Machine virtuelle JAVA (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) .



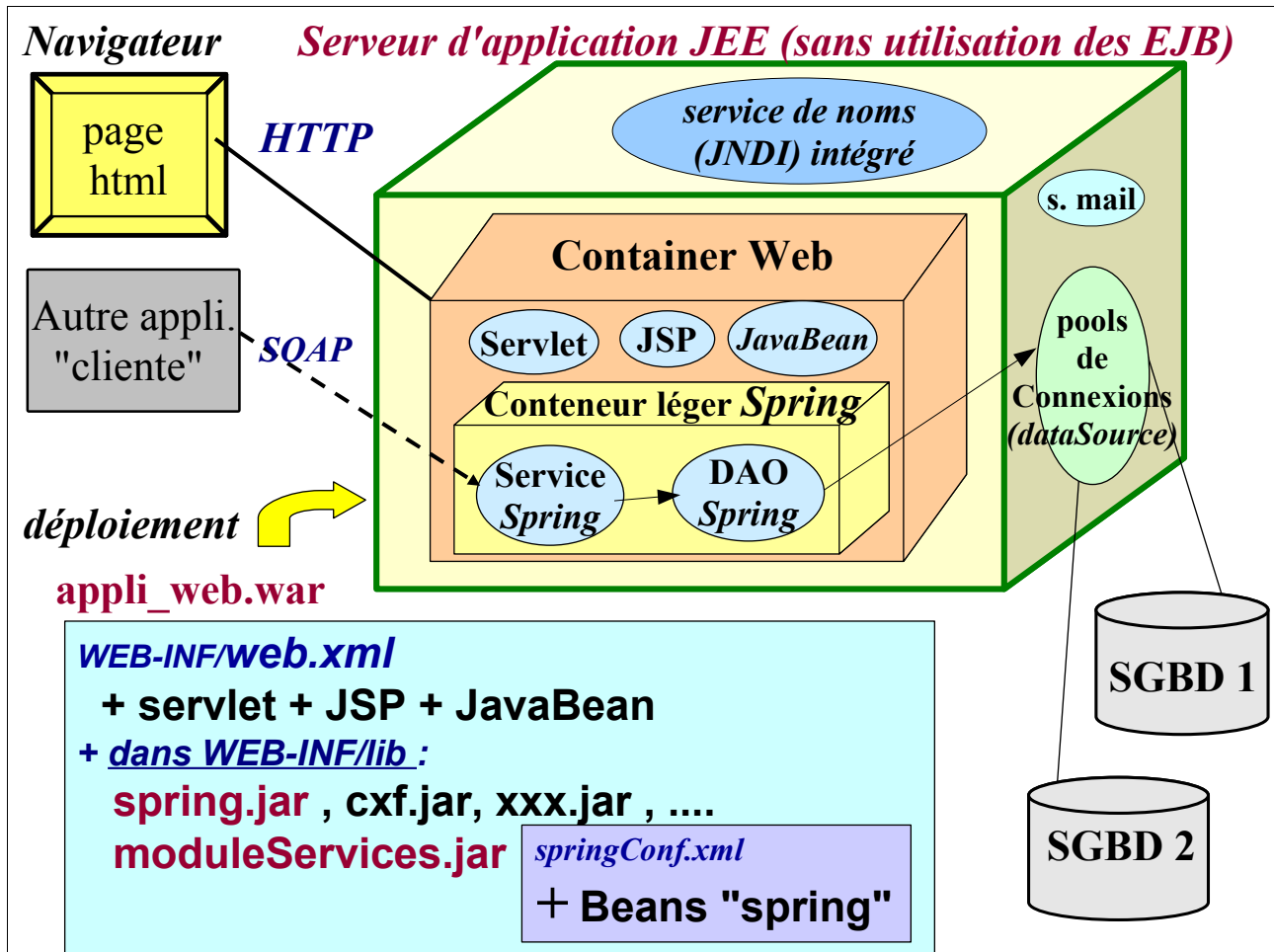
Certains mécanismes internes des serveurs d'applications JEE ont besoin de déclencher des compilations (ex: pages JSP transformées en Servlet à compiler) . Il faut donc s'appuyer sur le JDK complet (compilateur + JRE = Java Runtime Environment) .

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

8. Utilisation de "Spring" à la place des EJB

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

On peut préférer utiliser un conteneur léger tel que Spring (qui offre des avantages certains sur le plan de la modularité et qui rend les tests plus faciles du fait d'une meilleure indépendance avec le serveur d'application).



9. Evolution de JEE

Evolutions de J2EE, JEE5, JEE6

J2EE 1.0 & 1.1

Socle architecture = Servlet/JSP + JNDI + RMI + EJB

J2EE 1.2 (*WebSphere 4 , WebLogic 6*)

Formalisation des archives (.war, .jar, .ear) – Standard de déploiement , Références de ressources (indirections)

J2EE 1.3 (*WebSphere 5 , WebLogic 7 et 8 , Jboss 3.2*)

EJB 2.0 (**MDB**, ..., Interfaces locales, ...) , Connecteurs **JCA** (et .rar)

J2EE 1.4 (*WebSphere 6 , WebLogic 9 , Jboss 4 , Jonas 4*)

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

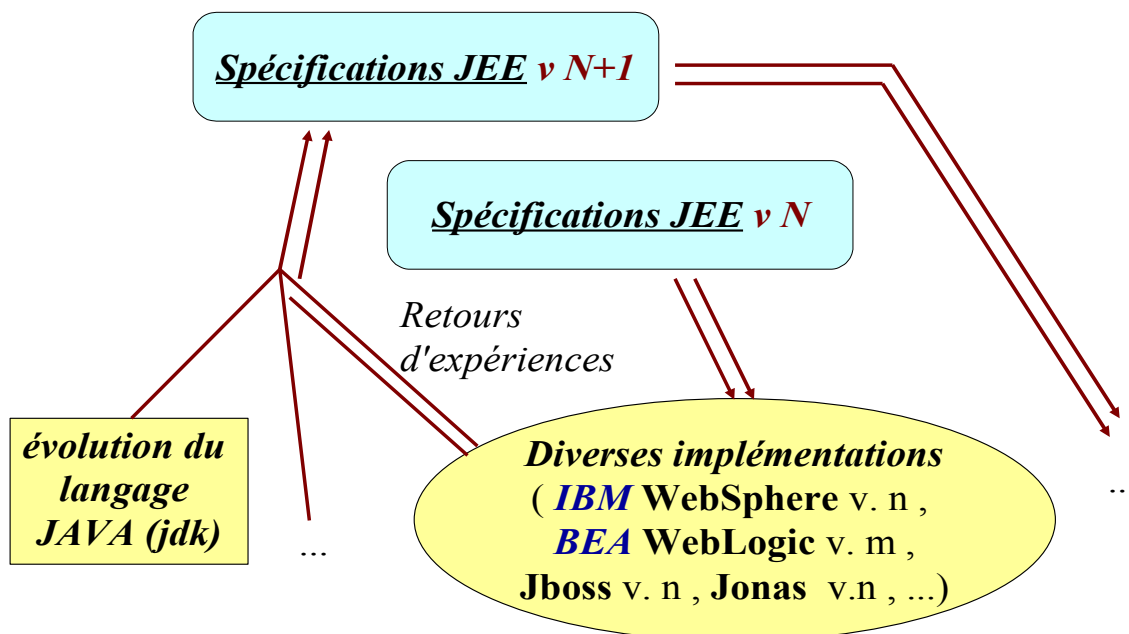
EJB3 (config. via annotations , **Java Persistence Api**)

Framework web **JSF** (Java Server Faces) , **IOC** ,

JAX-WS (Api simple et efficace pour Services Web)

JEE6 (*jdk >=1.6 , Jboss 7 , ...*) : annotations coté Web, ...

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



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

10. Principaux serveurs d'applications (JEE)

<i>Serveurs d'applications</i>	<i>Marques/Editeurs</i>	<i>Caractéristiques</i>
WebSphere	IBM	<ul style="list-style-type: none"> - 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	<ul style="list-style-type: none"> - Autre bon produit commercial (à peu près aussi sophistiqué que WebSphere)
Jboss (4.2 , 5.1 , 7.0)	Jboss / Red Hat	<ul style="list-style-type: none"> - 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.
Jonas	OW2 (INRIA + Bull +)	<ul style="list-style-type: none"> - Open source (produit stable et sérieux) <p>moins utilisé que Jboss car un peu en retard à l'époque des premières versions.</p>
Geronimo	Apache Group	<ul style="list-style-type: none"> - Open source - Serveur récent (assez peu de recul)
Tomcat (*)	Apache Group	<ul style="list-style-type: none"> - Open source faisant office de référence sur la partie "conteneur Web". - Serveur JEE simplifié (partie "conteneur web" seulement (sans EJB)).
GlassFish	SUN	<ul style="list-style-type: none"> - Serveur JEE de SUN (en partie open source) assez complet et assez innovant sur certaines technologies (BPEL, ESB/JBI ,) - Serveur récent (assez peu de retour/recul en production)

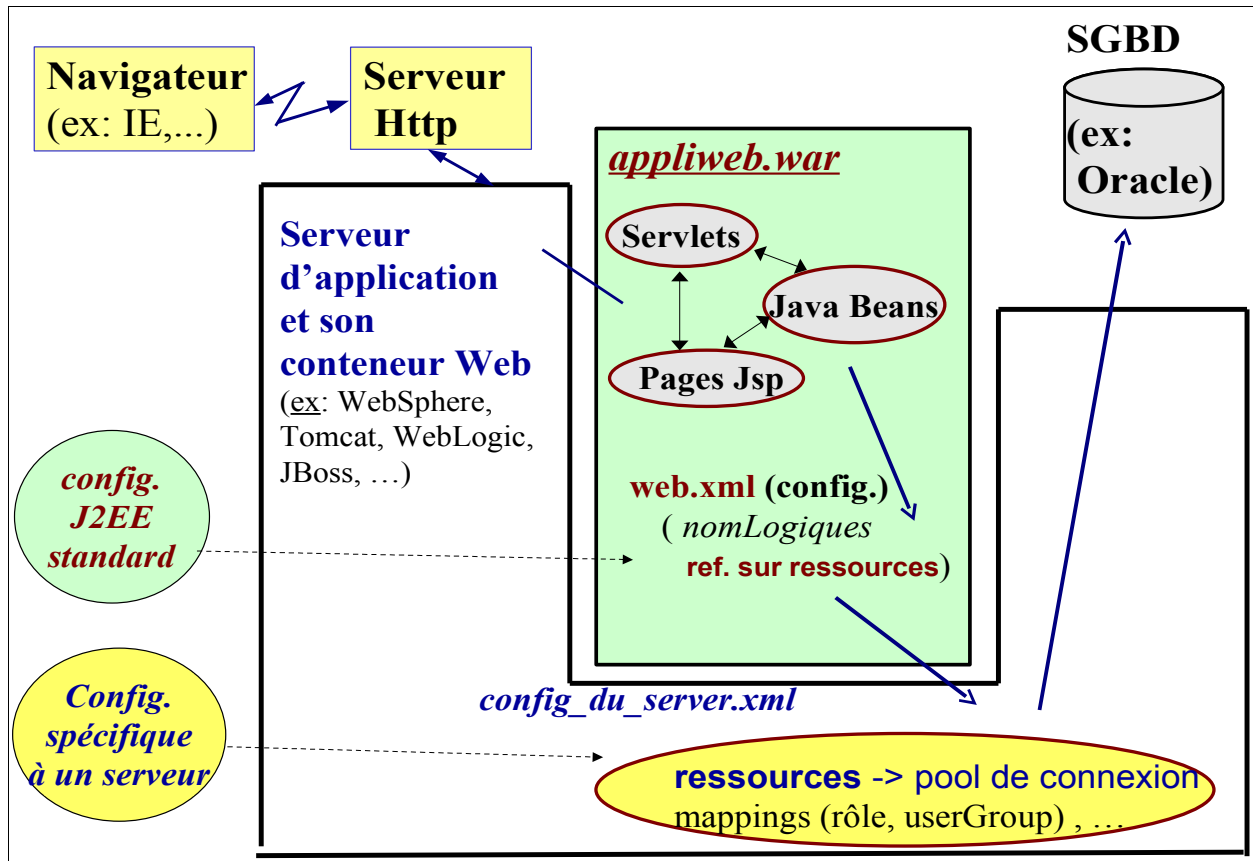
(*) **NB:** Tomcat peut être utilisé :

–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 (Jboss , Jonas ,). En règle générale on utilise/télécharge quasiment toujours une version pré-assemblée "Jboss + Tomcat" ou "Jonas + Tomcat".

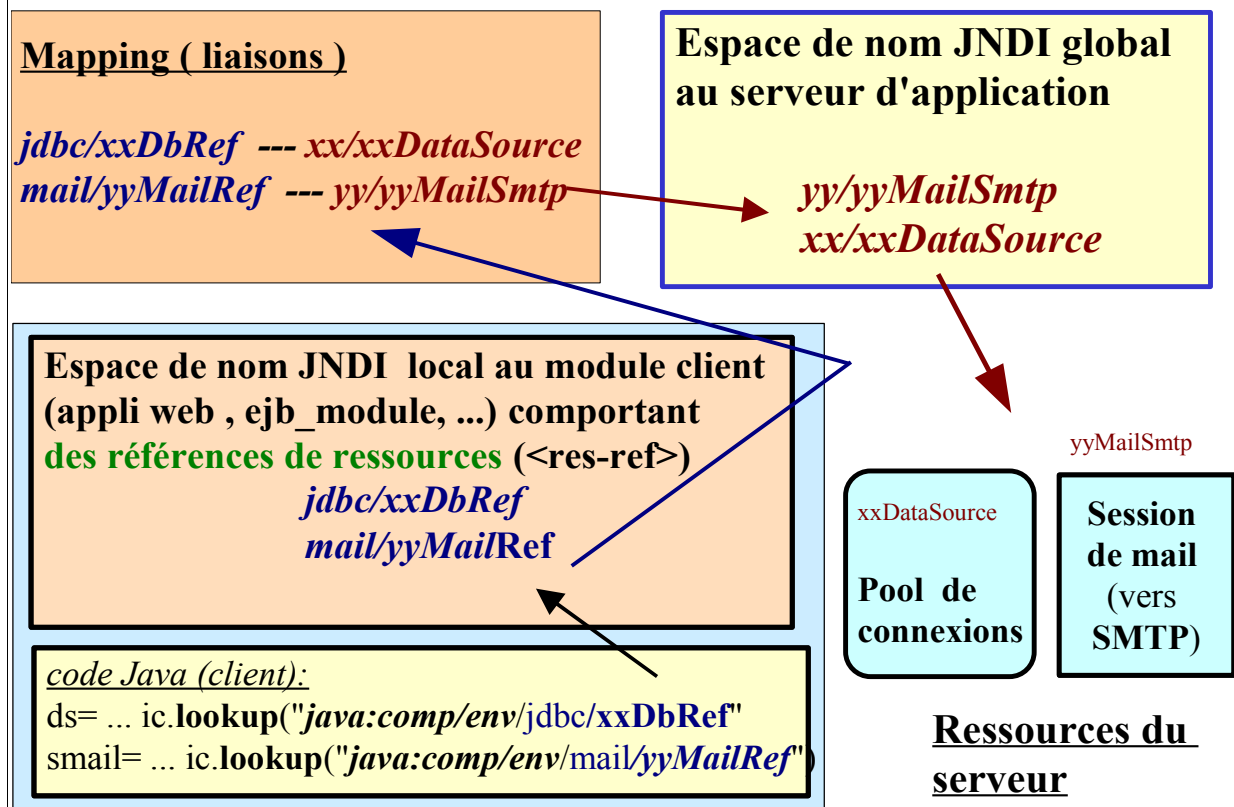
11. Ressources propriétaires et références standards

90 % des aspects "développement" et "déploiement" d'une application JEE sont tout à fait standards.

Les **principales différences** entre les **différents serveurs d'application (WebLogic, WebSphere , ...)** se situent au niveau de la **gestion des ressources internes (pool de connexions , clusters , logs ,)** et au niveau de la façon de les **configurer** et les **administrer** .



Références de ressources (ENC) & liaisons JNDI



12. 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_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/application_5.xsd"
  id="Application_ID" version="5">
  <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_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  id="WebApp_ID" version="2.5">
  <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>
```


13. 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>
```

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

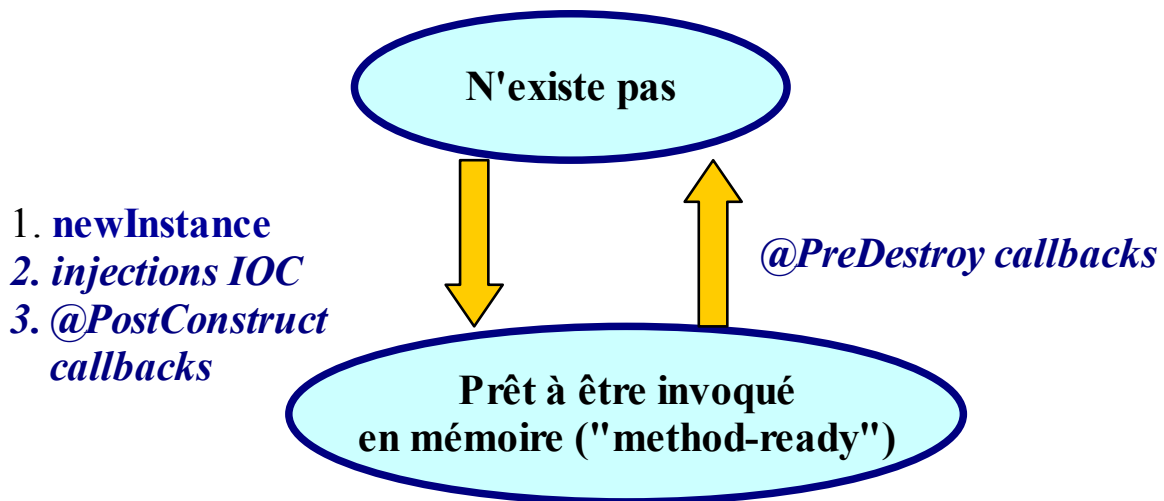
Cette éventuelle présence d'éléments techniques (références de ressources, logique transactionnelle, contrôle d'accès selon authentification, ...) fait qu'un EJB Session sans état est :

- 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) :

Cycle de vie - EJB3 Session sans état (stateless)



Nb: une même instance (initialisée dans un pool interne) peut, après avoir géré une certaine requête, être tout de suite réutilisée pour traiter une autre requête (provenant éventuellement d'un autre client).

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.

Attention (bug Eclipse/JBoss 5.1):

Pour pouvoir démarrer Jboss 5.1 sans problème depuis eclipse, il faut modifier le fichier

jboss-5.1.0.GA\server\default\conf\bootstrap\profile.xml

en ajoutant *class="java.io.File"* au niveau de la balise suivante :

```
<!-- The attachment store -->
<bean name="AttachmentStore"
      class="org.jboss.system.server.profileservice.repository.AbstractAttachmentStore">
    <constructor><parameter class="java.io.File">
        <inject bean="BootstrapProfileFactory" property="attachmentStoreRoot" />
    </parameter></constructor> ...
```

2. Conventions de noms

Attention :

Les conventions de noms qui suivent ont été vérifiées avec **JBoss 5.1**

Des tests et/ou des ajustements avec d'autres serveurs peuvent s'avérer nécessaires.

Si l'*application J2EE* s'appelle "*myJeeApp*" et si la *classe java interne de l'EJB* est dénommée *XxxBean* alors (par défaut):

- le nom JNDI de l'EJB (en accès distant) est "*myJeeApp/XxxBean/remote*"
- le nom JNDI de l'EJB (en accès local) est "*myJeeApp/XxxBean/local*"

D'autre part, un *EJB3 Session sans état* configuré de la façon suivante

```
@Stateless
@WebService(endpointInterface="myejb.Convertisseur")
public class ConvertisseurBean implements Convertisseur {
    ...
}
```

et appartenant au module d'EJB "*myJeeAppEJB*"

sera vu comme un service web ayant les références suivantes:

- URL wsdlUrl = new
URL("*http://localhost:8080/myJeeApp-myJeeAppEJB/ConvertisseurBean?wsdl*");
QName qname = new QName("*http://myejb/jaws*", "*ConvertisseurService*");
- EndPoint URL ==> *http://localhost:8080/myJeeApp-myJeeAppEJB/ConvertisseurBean*

NB: les descriptions "*WSDL*" des services web déployés par Jboss sont placés dans le répertoire
jboss-5.1.0.GA\server\default\data\wsdl

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

Un client distant et externe au serveur J2EE doit avant tout préciser les *paramètres JNDI* permettant de se connecter au serveur de noms interne au serveur d'application.

Dans le cas du serveur **JBoss 4 et 5** ces paramètres techniques sont les suivants :

clientJndi.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
#j2ee.clientName=myJeeAppClient
#java.security.auth.login.config=C:\Prog\...\jboss-...\client\auth.conf
```

avec:

```
public class Client {
private java.util.Properties props = null;

    private void init_client_env()
    {
        try {
            InputStream inStream = Thread.currentThread().getContextClassLoader()
                .getResourceAsStream("clientJndi.properties");
            this.props = new java.util.Properties();
            props.load(inStream);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    ....
}
```

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

```
Manifest-Version: 1.0
Main-Class: myclient.Client
```

(classe avec méthode main())

```
package myclient;

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

public class Client
{
    private java.util.Properties props = null;
    public static void main(String[] args) throws Exception
    {
        test_calculator();
    }
}
```

```

}

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

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

```

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("myJeeApp/CalculatorBean/local");
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="myJeeApp/CalculatorBean/local")
private Calculator calculator;

```

5. Références de ressources

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.

5.1. 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</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;
```

6. 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;
    ...
}
```

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(targetNamespace="http://xxx/serv/")
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",
            targetNamespace="http://xxx/serv/")
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/Run 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") utilisent en interne l'API JAX-WS.

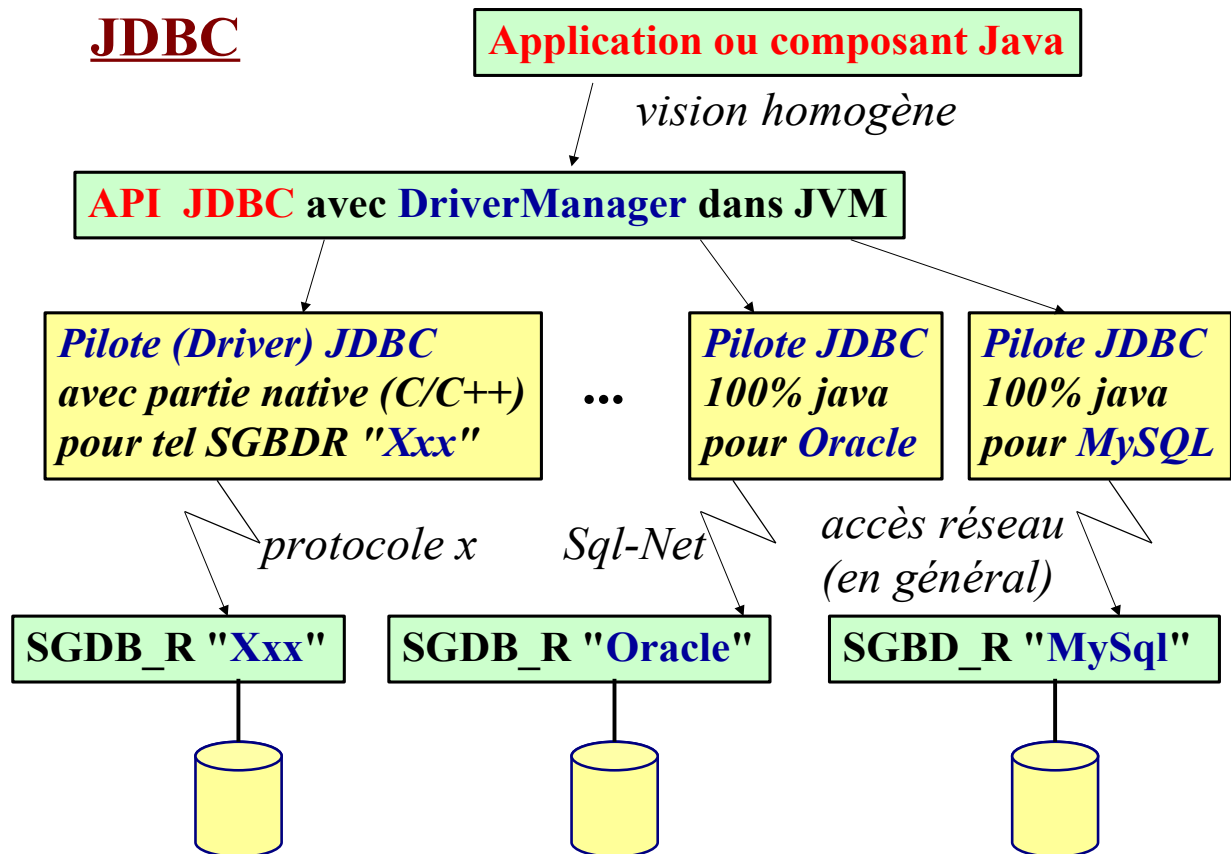
==> Etudier si besoin l'api JAX-WS pour approfondir le sujet.

Attention: JAX-WS n'est bien implémenté qu'au sein des versions récentes du jdk1.6 .

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

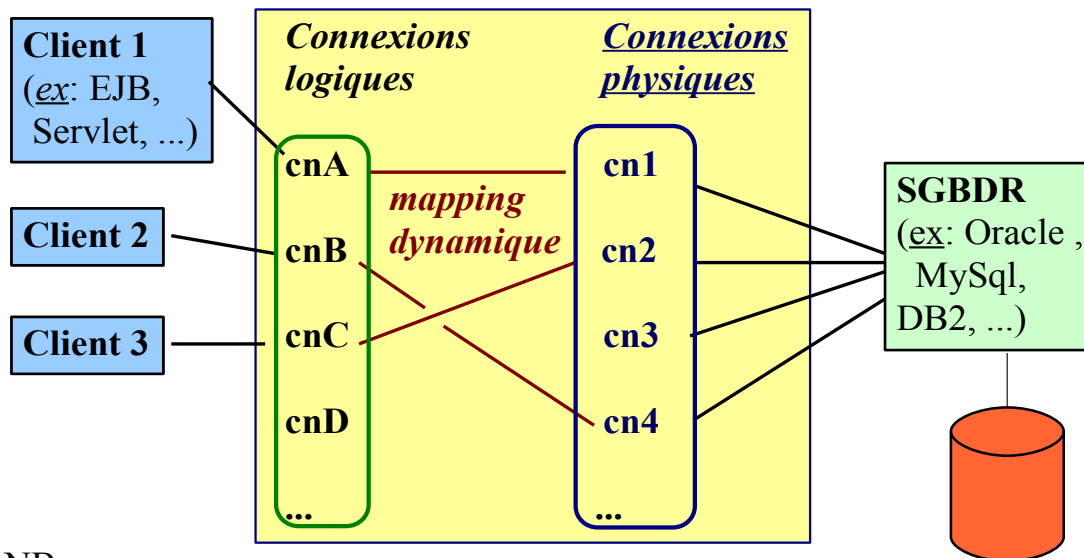
Rôles (utilités) des pools de connexions:

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 (Java Connector Architecture) est une API assez générique permettant à un composant déployé dans un serveur d'application J2EE de communiquer avec des systèmes d'informations très divers d'une entreprise (**E.I.S.**) :

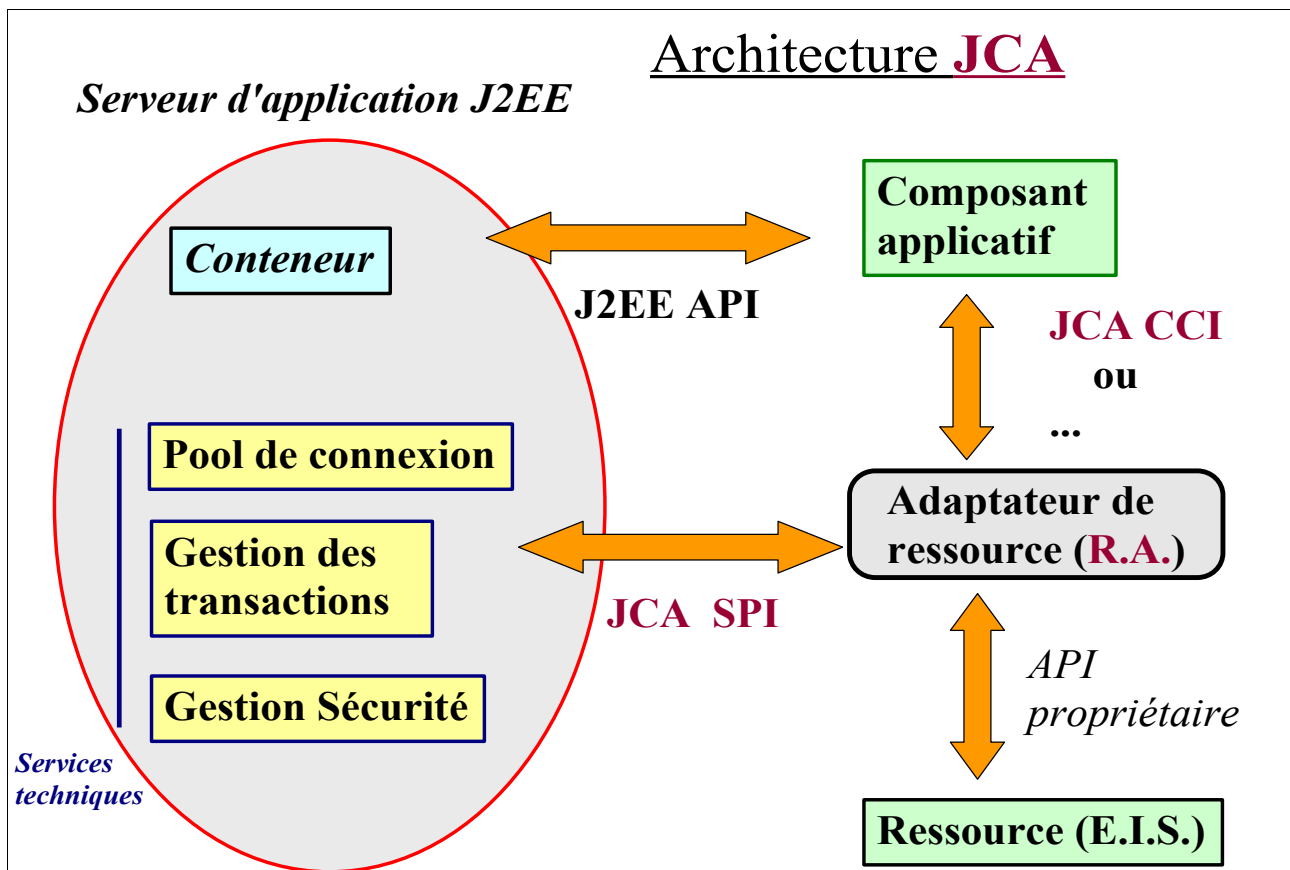
- **SGBDR** (Oracle, Sybase, DB2, SQL-server, ...) – via JDBC.
- **Moniteur transactionnel** (Tuxedo, CICS, ...).
- **Système propriétaire** (SAP, ..., ERP divers, ...).

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:/MyDB");
```

3.2. injection directe d'une dépendance vers une ressource

```
@Stateless
public class MySessionBean implements MySession {
    ...
    @Resource(mappedName="java:/myDB")
    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 du serveur JBoss (en version 3, 4 , 5 et 6) , une source de données JDBC se configure via un fichier "**xxx-ds.xml**" qu'il faut placer dans le répertoire **JBoss/server/default/deploy** .

NB: le répertoire **Jboss/doc/examples/jca** comporte un exemple de fichier "xxx-ds.xml" pour chaque type de base de données connue (DB2 , MySQL , Oracle , SqlServer ,).

Ceci est souvent la base d'un copier/coller.

Mode opératoire conseillé:

- 1) recopier un fichier "xxx-ds.xml" dans un répertoire eclipse de type "config" et renommer éventuellement ce fichier en "nomDataBase-ds.xml"
- 2) éditer ce fichier avec tous les paramétrages nécessaires (URL_DB , username, password).
- 3) recopier ensuite le fichier "...-ds.xml" au sein du répertoire server/default/deploy de Jboss.

Important: Ne pas oublier de placer une copie du driver JDBC adéquat (ex: **mysqlConnector...jar**) au sein du répertoire **server/default/lib** de Jboss .

Exemple (mysql-ds.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- fichier à recopier dans Jboss/server/default/deploy (config DataSource)
nécessite mysqlconnector...jar dans server/default/lib -->
<datasources>
  <local-tx-datasource>
    <jndi-name>MyDbDS</jndi-name>
    <connection-url>jdbc:mysql://localhost/mydb</connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <user-name>root</user-name>
    <password>root</password>
  </local-tx-datasource>
</datasources>
```

NB: Un redémarrage complet du serveur (Jboss ou WebSphere ou ...) est souvent nécessaire pour que la configuration du pool de connexions soit bien prise en compte.

Dans le cas du serveur Jboss, on peut vérifier la configuration en:

1. activant la console d'administration (<http://localhost:8080>) et la sous console "**JMX**"
2. sélectionnant le service Jboss/**service=JNDIView**
3. sélectionnant la méthode "**list**" et en l'invoquant via "**invoke**" .
4. La partie "**Global JNDI Name**" ou "**Java**" doit normalement comporter [java:/]MyDbDS .

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

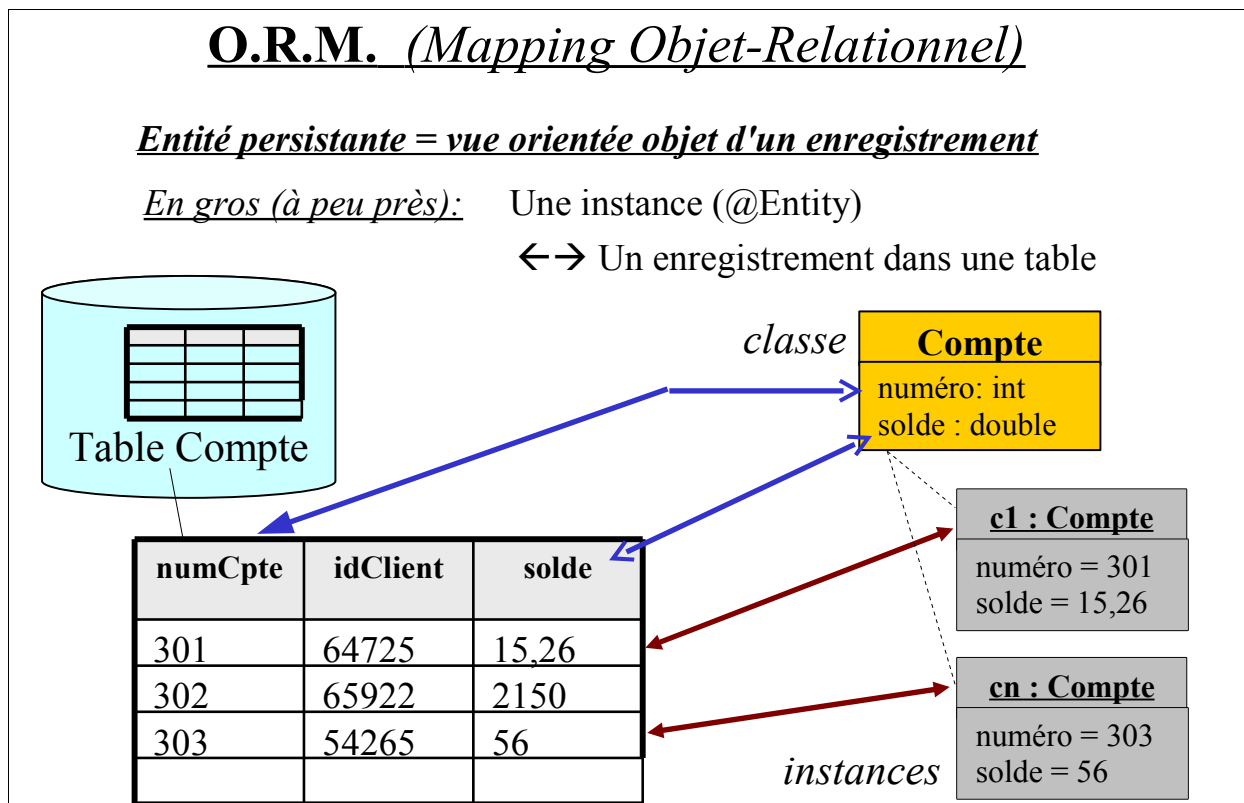
un ensemble d'enregistrements d'une base relationnelle.

O.R.M. (Mapping Objet-Relationnel)

Entité persistante = *vue orientée objet d'un enregistrement*

En gros (à peu près): Une instance (@Entity)

↔ Un enregistrement dans une table



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. Éléments 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** :

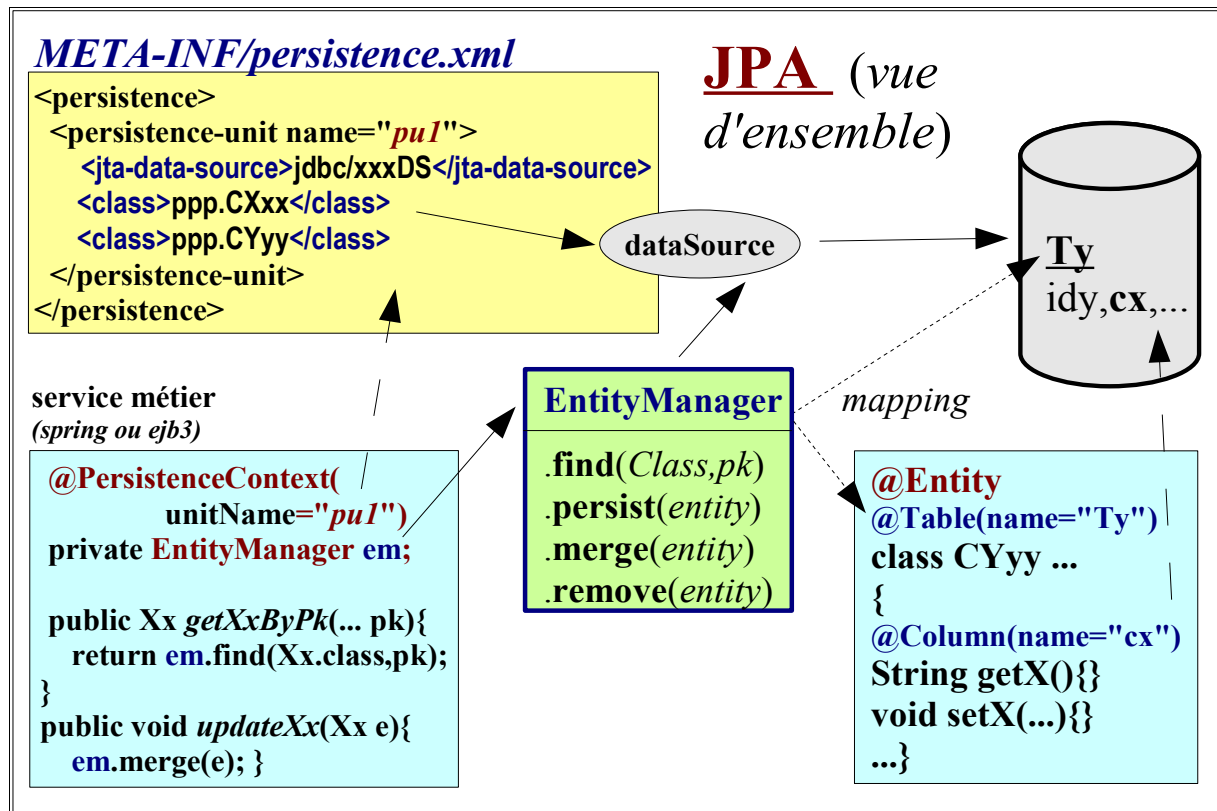
- **Associations "1-1" , "1-n" , "n-n"** (jointures entre tables <---> relations entres objets)
- **Objets "valeur"** (sous parties d'enregistrements , tables secondaires , ..)
- **Généralisation/Héritage/Polymorphisme** (<---> schéma relationnel ?)
- ...

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

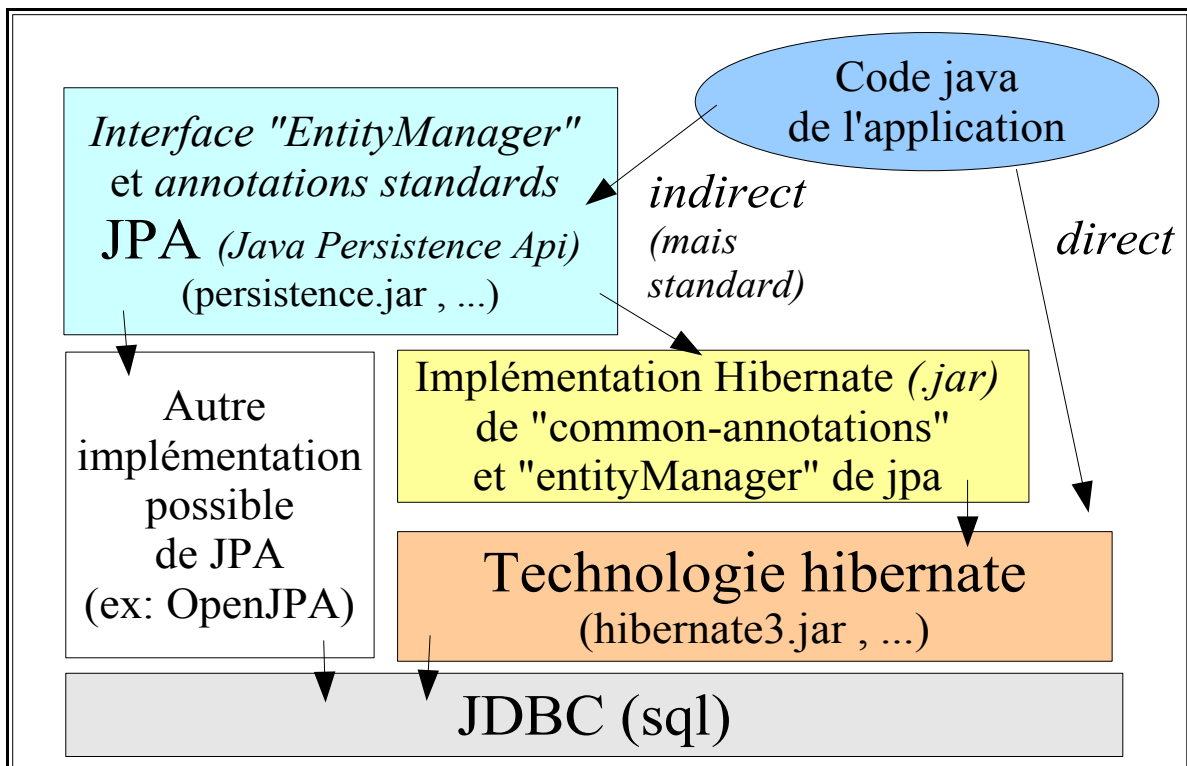
2. JPA (Java Persistence Api)

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 ,)



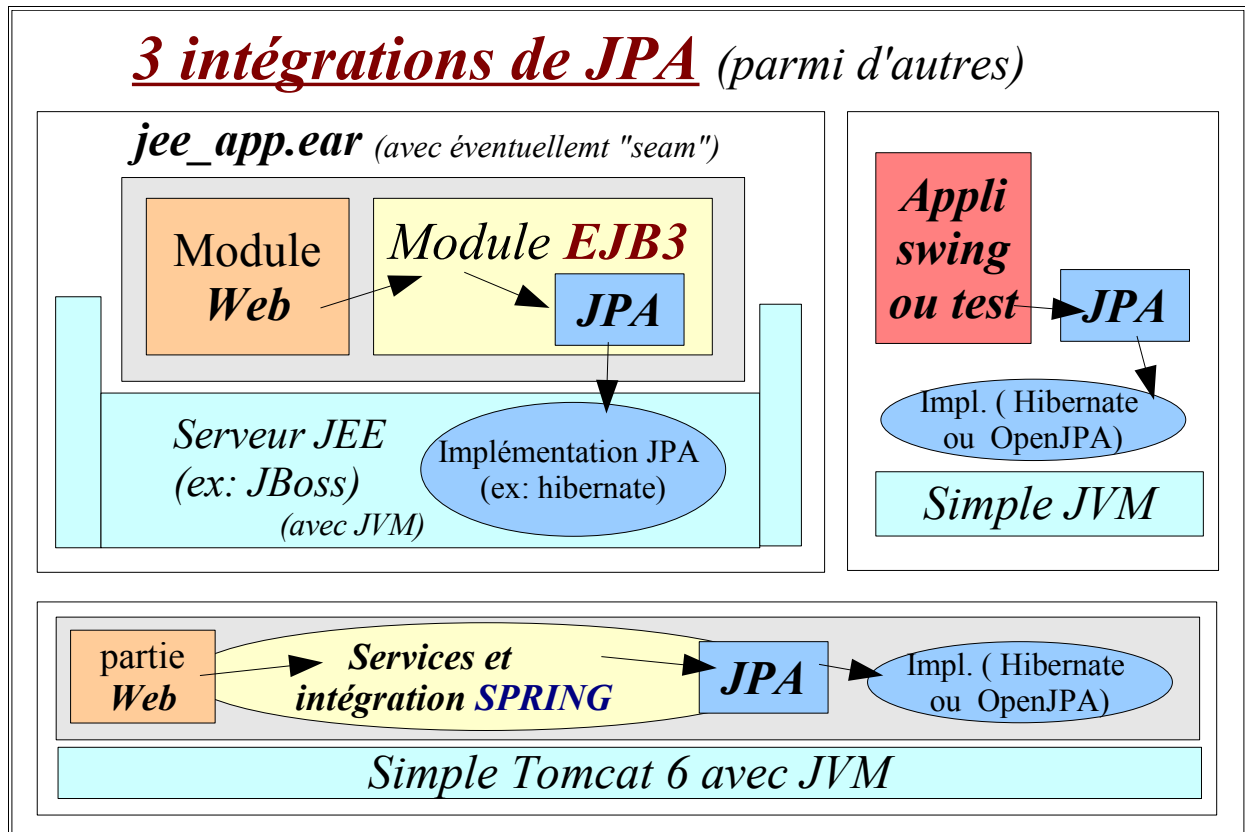
Hibernate est quelquefois utilisé en tant qu'implémentation interne de JPA.
C'est le cas au sein du serveur Jboss :



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:

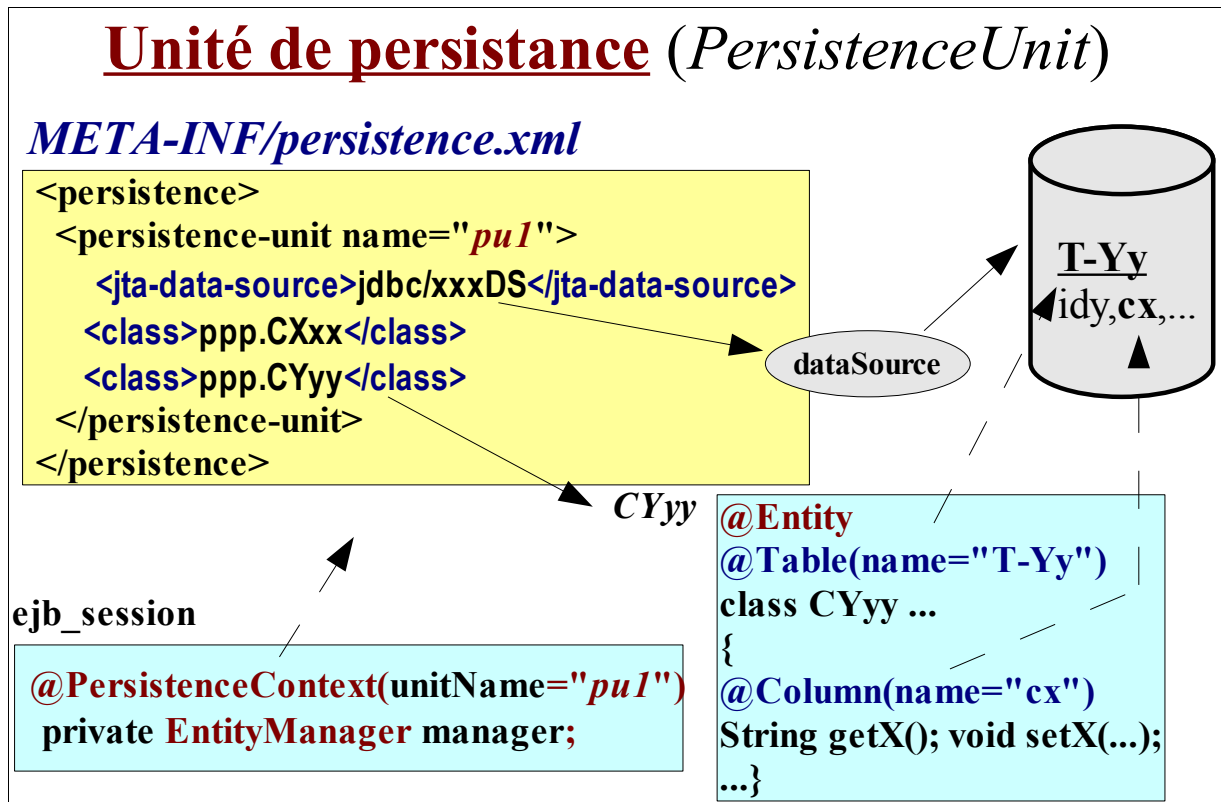
- Lorsqu'une unité de persistance est incorporée dans un module d'EJB , celle ci est alors considérée comme "privée" et ne peut pas être utilisée par d'autres modules. D'autre part, ci celle-ci est unique, il n'est pas indispensable de préciser son nom.
- Lorsqu'une unité de persistance est packagée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

<persitence> ... </persitence> 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

```
<persistence>
  <persistence-unit name="OrderManagement">
    <description>....</description>
    <jta-data-source>jdbc/MyOrderDB</jta-data-source>
    <!-- <mapping-file>ormap.xml</mapping-file> -->
    <!-- <jar-file>MyOrderApp.jar</jar-file> -->
    <class>com.widgets.Order</class>
    <class>com.widgets.Customer</class>
    <properties>
      <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 env 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).

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>`)
- une ou plusieurs archives (.jar) dans lesquelles seront scrutées toutes les classes (balises `<jar-file>`) pour rechercher les `"@Entity"`
- Les classes annotées situées à la racine du module de persistance (sauf si l'élément `exclude-unlisted-classes` est spécifié)

Dans l'exemple ci-dessus , seront prises en compte les classes de POJO qui sont:

- définies dans `ormap.xml`
- + celles comportant des annotations de persistance et situées dans `MyOrderApp.jar`
- + celles explicitement listées.

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

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">
  <!-- with JNDI lookup inside JBoss App Serv -->
  <!-- <jta-data-source>java:/DefaultDS</jta-data-source> -->

  <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <!-- <class>entity.persistence.jpa.Compte</class>
    <class>entity.persistence.jpa.XxxYyy</class> -->
  <properties>
    <property name="hibernate.dialect"
      value="org.hibernate.dialect.MySQLDialect" />

    <!-- pour éviter <class> , <class> , <class> en mode J2SE -->
    <property name="hibernate.archive.autodetection" value="class, hbm"/>

    <!-- <property name="hibernate.hbm2ddl.auto" value="create-drop" /> -->

    <property name="hibernate.connection.driver_class"
      value="com.mysql.jdbc.Driver" />
    <property name="hibernate.connection.username" value="mydbuser" />
    <property name="hibernate.connection.password" value="mypwd" />
    <property name="hibernate.connection.url"
      value="jdbc:mysql://localhost/test_hibernate_db" />
  </properties>
</persistence-unit>
</persistence>
```

4. Configuration du mapping JPA via annotations

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

4.1. Exemple d'entité persistante (@Entity)

```
@Entity
@Table(name = "T_CUSTOMER")
public class Customer implements Serializable {
    private Long id;
    private String name;
    private Address address;
    private Collection<Order> orders = new HashSet();
    private Set<PhoneNumber> phones = new HashSet();

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

    @Id
    public Long getId() {return id;} // pk
    public void setId(Long id) {this.id = id;}

    @Column(name = "customer_name")
    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;}

    @OneToMany(...)
    public Collection<Order> getOrders() {return orders;}
    public void setOrders(Collection<Order> orders) {
        this.orders = orders;}

    @ManyToMany(...)
    public Set<PhoneNumber> getPhones() {return phones;}
    public void setPhones(Set<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 "id" (qui aurait pu s'appeler "customer_id") comme **identifiant de l'entité (et comme clef primaire de la table)**.
- les autres annotations (**@OneToMany** , ...) seront approfondies dans un chapitre ultérieur.

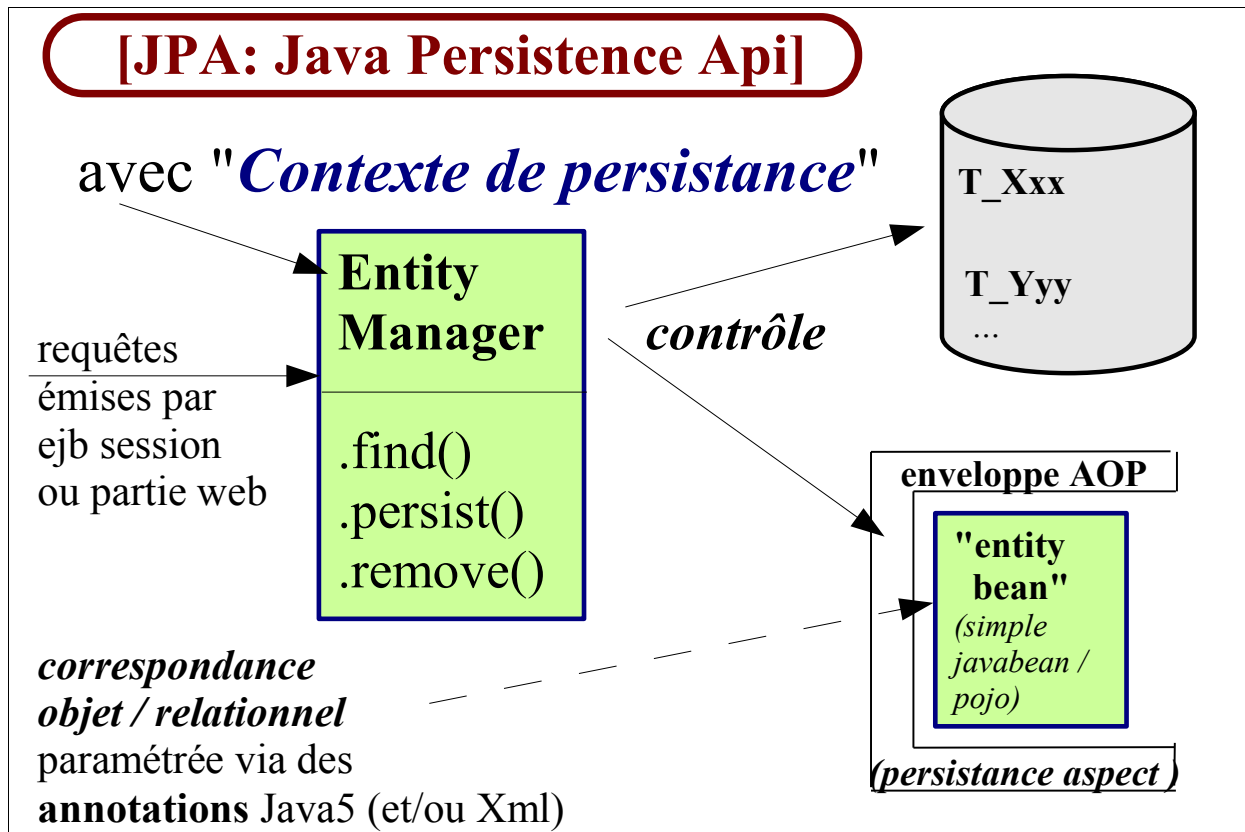
NB: Lorsqu'une annotation (@Column ou ...) **n'est pas présente , les noms sont censés coïncider** (sachant que les minuscules et majuscules n'ont pas d'importance du coté SQL/relationnel).

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 !!!)
- 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).

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 qlString);
public Query createNativeQuery(String qlString, Class resultClass);
public Query createNativeQuery(String qlString, 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)

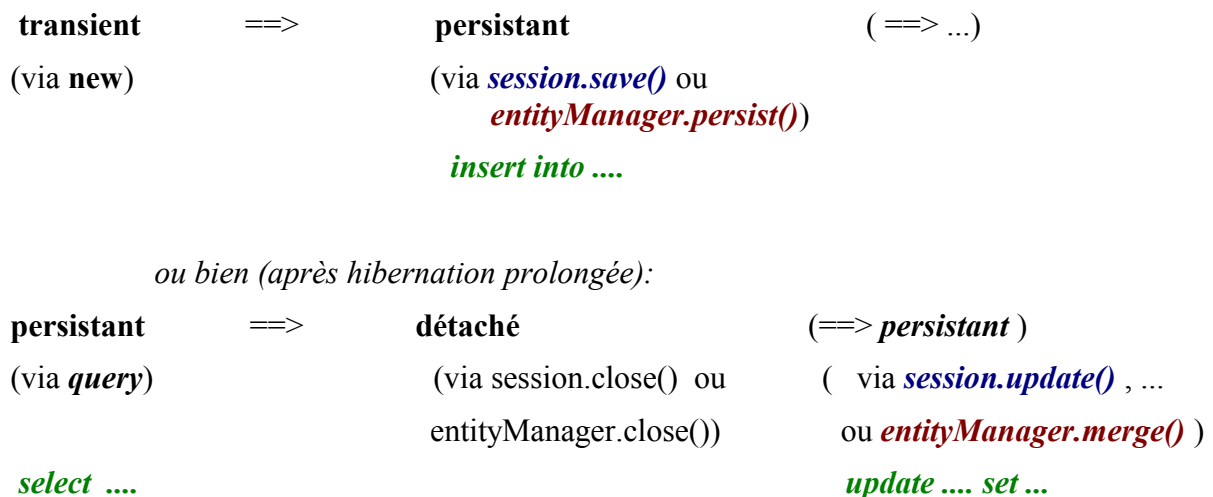
Exemple (avec gestion explicite):

```
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

<i>Etats objets</i>	<i>Caractéristiques</i>
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 (<i>avant session.close() ou entityManager.close()</i>). 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 (<i>après session.close() ou entityManager.close()</i>). 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.

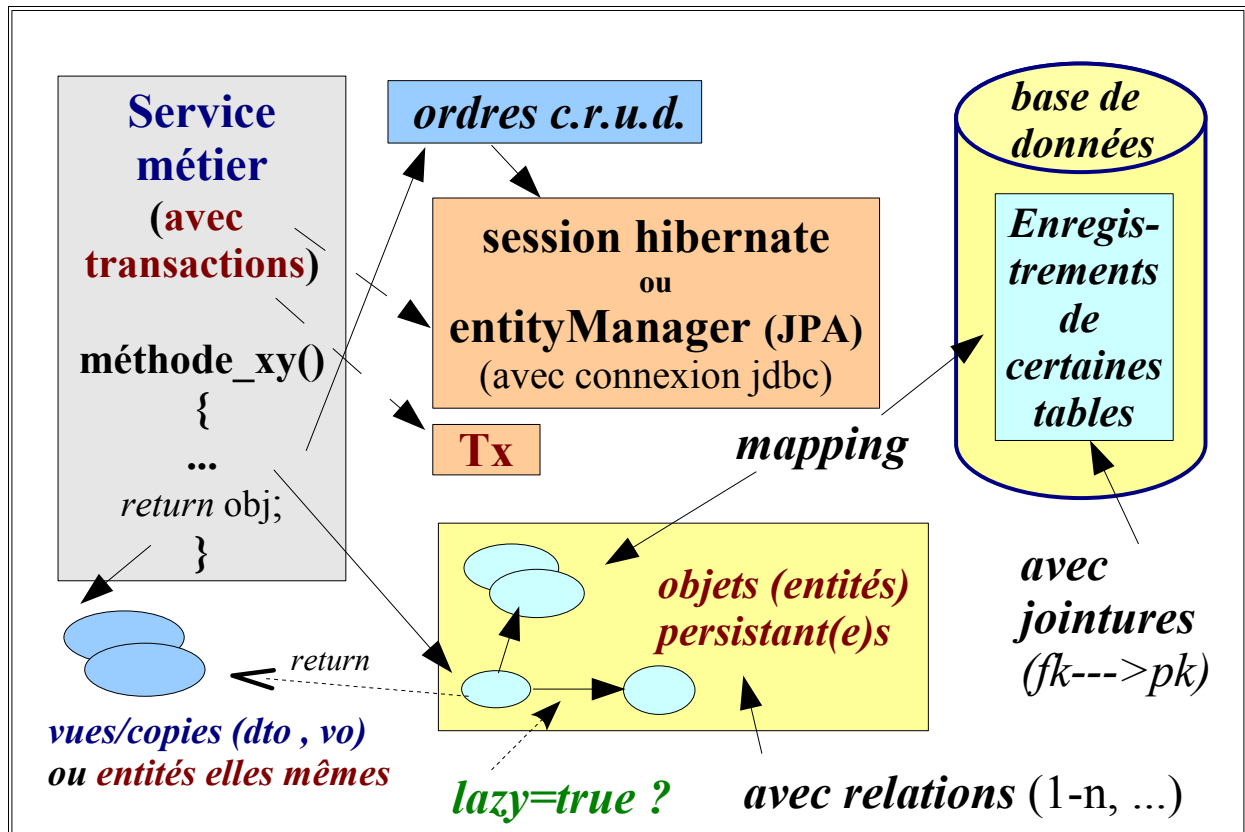
4. Cycle de vie d'un objet JPA/Hibernate



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) <i>ou bien assez inutilement</i> .persist (obj_déjà_persistant)	<u>Rend persistant un objet détaché :</u> 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) <i>ou bien assez inutilement</i> .merge (obj_déjà_dans_contexte_persistence)	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**

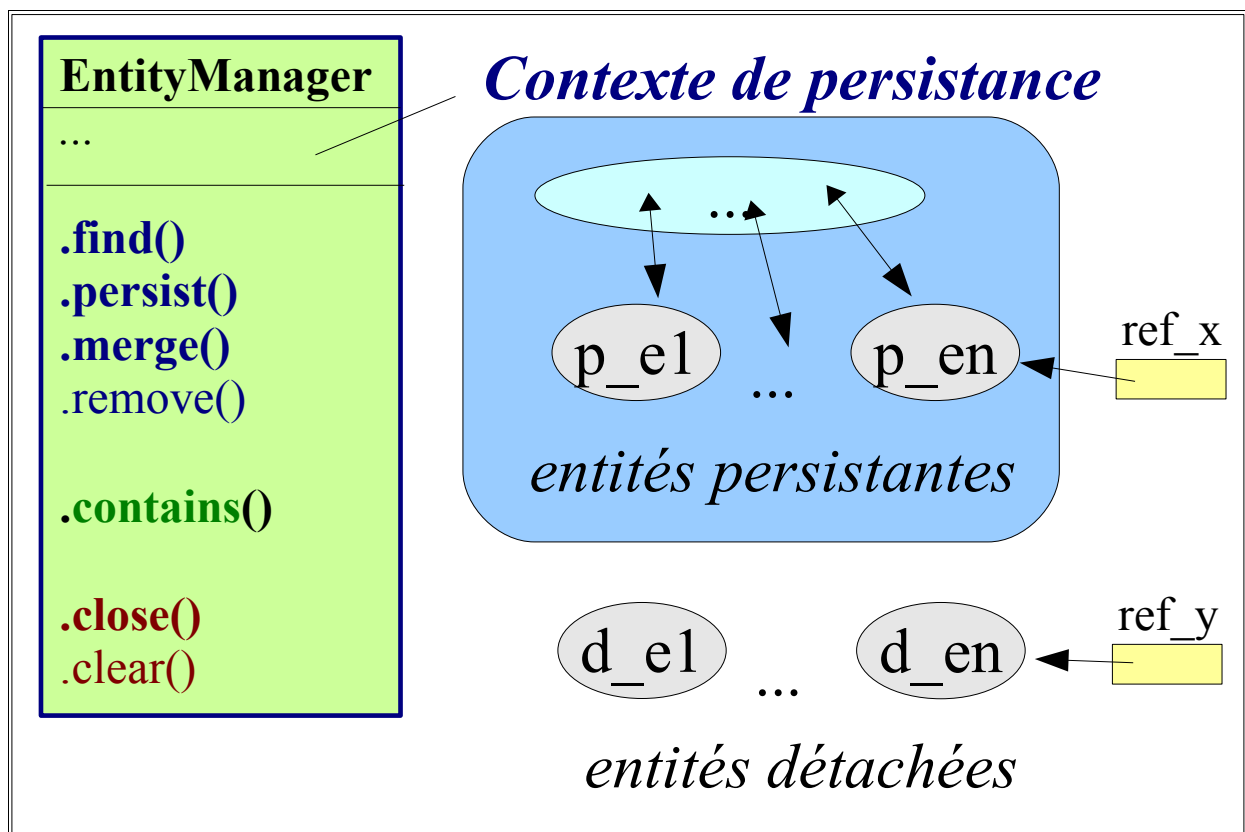
NB:

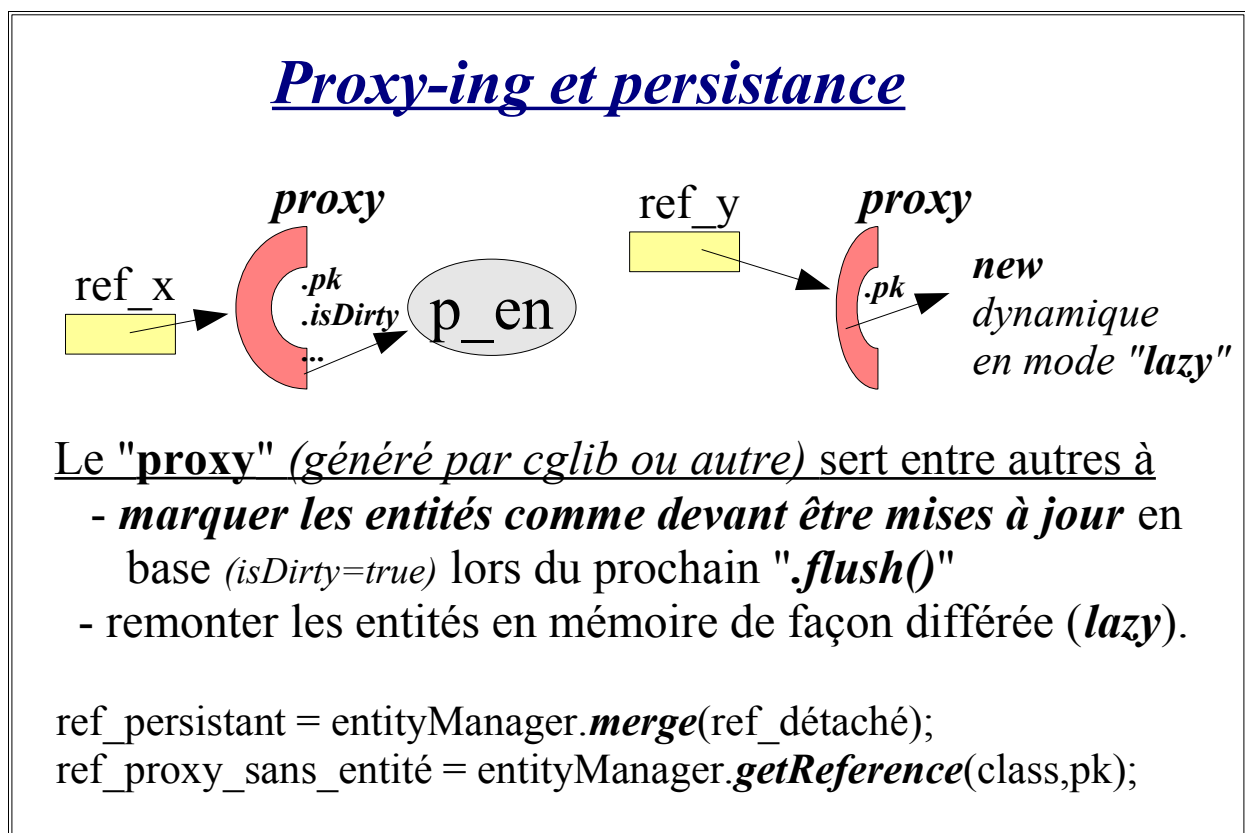
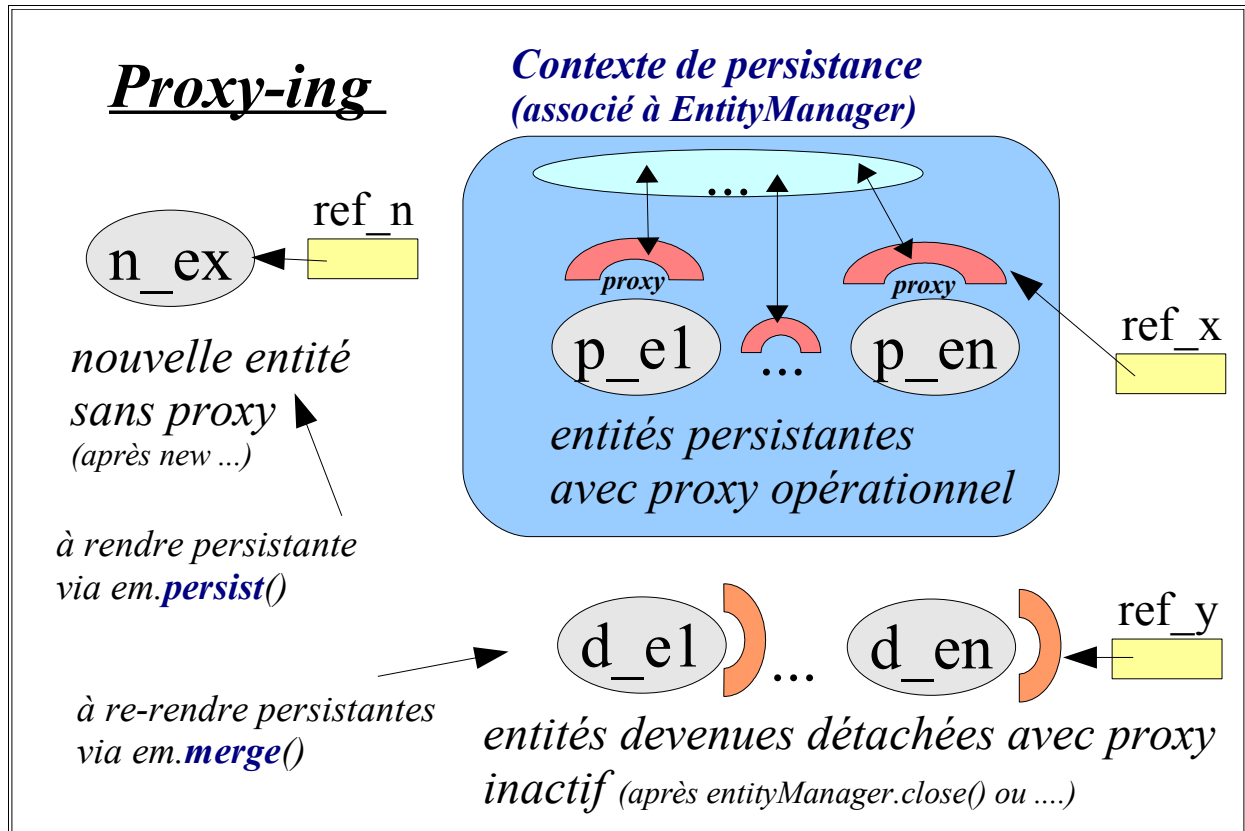
entityManager.getTransaction().commit() déclenche automatiquement *entityManager.flush()*

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

Principales méthodes de l'objet *Query*:

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

8. Contexte de persistance et proxy-ing



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 classes* (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 simples 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[]
```

```
select foo from Foo foo, Bar bar where foo.startDate = bar.date
```

```
from Cat cat where cat.mate.name is not null
```

1.4. Lancement d'une requête JPQL (JPA)

Exemples:

```
import javax.persistence.Query;

...
public Collection<Client> getAllClient()
{
    Query query = entityManager.createQuery("Select c from Client as c");
    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 Collection<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();  
}
```

IX - O.R.M. JPA (généralités)

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

1.1. Problématique

Une technologie de **mapping objet – relationnel** telle qu'**hibernate** ou **JPA** doit pouvoir s'appuyer sur un **modèle objet** qui soit :

- **naturel d'un point de vue "langage orienté objet"** tel que **java** (==> utilisation de **JavaBean** et de collections classiques)
- **compatible avec la structure générale d'une base de données relationnelle** (jointures entre *entités* : clefs étrangères référant des *clefs primaires*).

1.2. Entités

Entités

Client
num_cli (pk)
 nom , adresse, ...

Commande
num_cmde (pk)
 date , ...

Une entité (avec id/pk) existe même si elle n'est pas référencée .

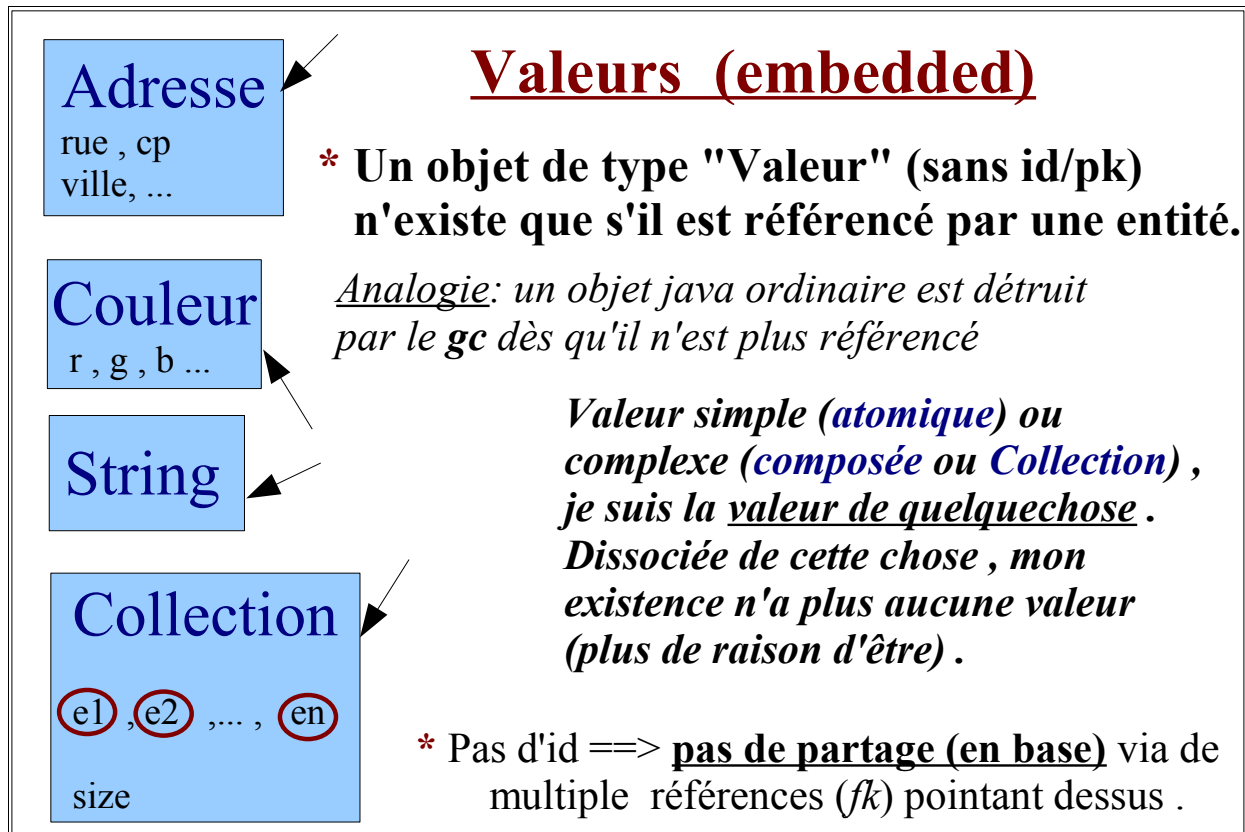
***Je ne pense pas
 mais j'ai une identité
 (clef primaire)
 donc je suis .***

To be or not to be : that is the question .

Pour un objet informatique, l'essence précède l'existence .

Sous **hibernate** et **JPA**, un objet java de type "entité" sera un simple **JavaBean** comportant obligatoirement une **propriété "id"** (ou "numéro" ou "...") faisant office d'identifiant et correspondra généralement en base à un enregistrement identifié par une **clef primaire** .

1.3. Objet "Valeurs" ou bien "imbriqué/incorporé (embedded)"



Concrètement, un objet hibernate/JPA de type "Valeur"/"Embedded" peut correspondre en base :

- à un ensemble de champs (colonnes) d'une certaine table [Valeur composée/component]
- pour Hibernate (mais pas pour JPA 1.0) : à des enregistrements secondaires (non partagés et référencés par une seule clef étrangère) [Collection de "component"] . Bien que pouvant être techniquement associés à des clefs primaires cachées, ces enregistrements secondaires n'ont pas de réelles identités propres : ce ne sont que des détails d'une et une seule entité principale [cascade-delete] .

Au niveau des mécanismes internes et automatiques de hibernate/JPA, on peut aussi considérer comme étant implicitement un sous objet de type "Valeur/Embedded" un objet "Collection de références" "purement Java" qui n'est pas directement associé à un enregistrements en base mais qui est associé à un ensemble de clefs étrangères qui référencent d'autres entités (relation 1-n ou un des sens d'une relation n-n).

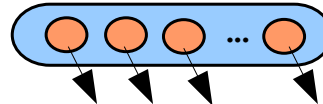
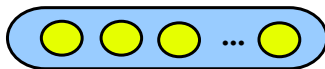
1.4. Collections et constituants

Collection et constituants

* Une collection est **globalement** un élément de type **valeur** (n'existant que si référencé) → 

* Les éléments d'une collection sont :

- soit des **valeurs** (**element** , **composite-element**)
- soit des **références vers d'autres entités**
(**one-to-many** , **many-to-many**, **many-to-any**).



* Une **clef de collection** (liée à une **clef étrangère** en base) permet de déterminer l'appartenance des éléments à une collection



Types de collections "hibernate" --> **map**, **set** , **list** , **bag** ,**array**

Les **list** , **array** et **map** ont des **index**

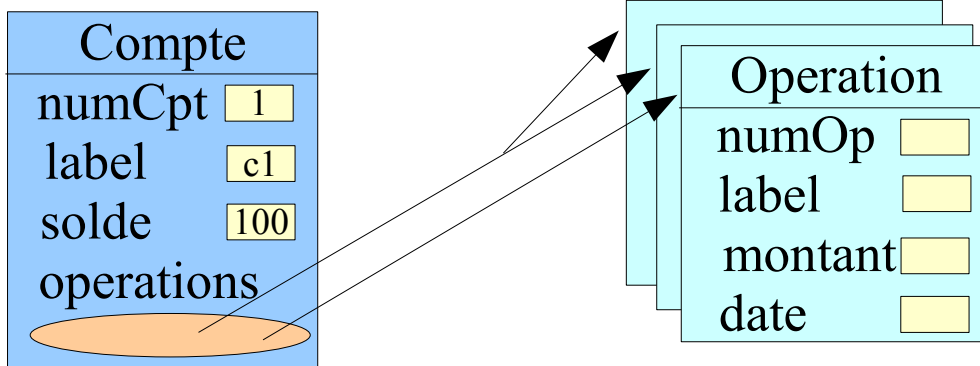
Les **set** et **bag** n'ont pas d'**index**

NB: Une collection de type **java.util.Set** pose quelquefois de gros problèmes d'affichage (ex: impossible de générer un tableau (h:dataTable) dans JSF) . La notion de "**Bag**" correspond en fait à une **simple Collection d'éléments** (implémentée en interne à partir de **java.util.List**) mais n'ayant pas nécessairement toute la sémantique des listes (pas d'ordre/pas d'index) .

Dans le cadre d'une **association 1-n ordinaire** Hibernate et JPA choisiront généralement (par défaut, sans indication particulière) des "**Bag**" (compatibles avec **java.util.List**) comme implémentations de **java.util.Collection** .

1.5. FetchType (lazy & eager)

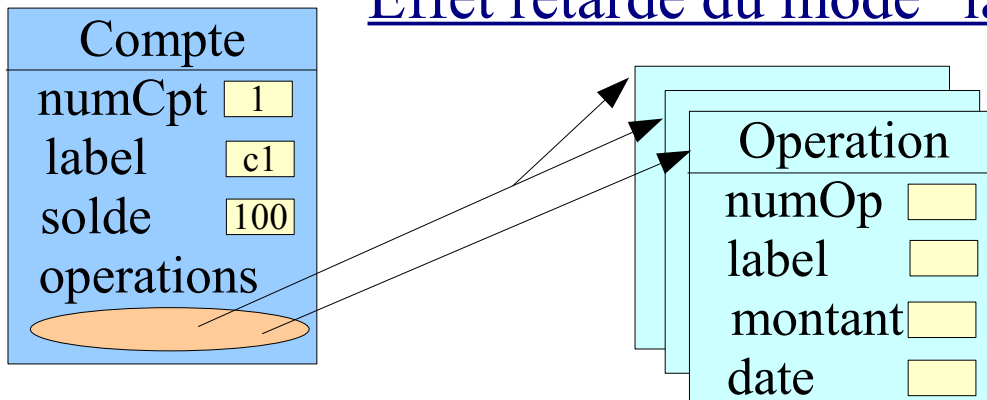
FetchType (Lazy & Eager)



Suite à *"select c from Compte as c where c.numCpt=1"* ,
remonte(nt) alors immédiatement en mémoire:

- **uniquement le compte1 en mode "lazy"**
 - **le compte1 et toutes ses opérations en mode "eager"**
- (via un 2^{ème} "select" SQL implicitement déclenché sur l'ensemble des opérations associées au compte1)

Effet retardé du mode "lazy"



- ♦ 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").

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 "false" ou "true" au paramètre lazy (très important sur le plan des performances).

NB: Avec Hibernate (et sa configuration hbm.xml) , le paramètre lazy est non seulement présent au niveau d'une collection (ex <set ... lazy="...">) mais il est également possible de lui affecter une valeur au niveau d'une classe ordinaire (<class ... lazy="...">) ce qui aura pour effet de préciser si les valeurs d'une sous objet de la classe en question doivent être par défaut remontées dès de chargement d'un objet principal (ex: 1-1 ,) .

Exemple JPA:

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

```
@OneToMany(cascade=CascadeType.ALL , fetch=FetchType.LAZY ,
           mappedBy="compte" )
public List<Operation> getDernieres_operations() {
    return dernieres_operations;
}
```

Exemple hibernate (.hbm.xml):

---> **lazy="true"** or **lazy="false"** .

```
<bag name="dernieres_operations" cascade="all" inverse="true" lazy="true">
    <key column="num_compte"/>
    <one-to-many class="domain.entity.Operation" />
</bag>
```

1.6. Operations en cascade

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

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}, ...
```

Syntaxe hibernate (dans les fichiers ".hbm.xml"):

```
<... cascade="all" ou cascade="none" ou cascade="delete" ou cascade="save-update" .... />
```

NB: ***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 et générateurs)

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

Un fichier de mapping hibernate (avec pour extension .hbm.xml) peut comporter l'indication d'un algorithme à utiliser pour générer une nouvelle clef primaire . Ceci s'effectue en imbriquant une sous balise **<generator>** ou sein de la balise **<id>** .

Exemples (hibernate):

```
<id name="num" unsaved-value="0">
    <column name="numero" not-null="true" />
    <generator class="native" />
</id>
```

```
<id name="id" type="long" column="cat_id">
    <generator class="hilo">
        <param name="table">uid_table</param>
        <param name="column">next_hi_value_column</param>
    </generator>
</id>
```

Exemple (JPA):

```
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name="numero")
public long getNum() {
    return num;
}
```

Principaux algorithmes/générateurs:

Hibernate	JPA	Algorithme
increment		Simple compteur (long, int , short) qui s'incrémente (nouvelle pk = max(pk)+1) ==> à ne pas utiliser si d'autres programmes (ou d'autres threads) sont susceptibles d'effectuer des insertions au même moments (ex: autre membre d'un cluster).
hilo		Même principe que "increment" cependant la dernière valeur du compteur est stockée dans une colonne d'une table spéciale (ce qui permet de bien gérer les verrous en cas d'insertions simultanées).
uuid.hex		universal unique id (128 bits) au format hexadécimal (style base des registres de Win32) ==> identifiant dont le type est compatible avec le format String de longueur >=32 caractères
identity ou sequence		identifiant géré par des mécanismes spécifiques à une certaine base de données (ex: sequence Oracle , auto_increment MySql ,)
native	AUTO	choisit identity , sequence ou hilo selon le type de base de données
....		Voir documentation de référence

3. Clef primaire composée

Avec Hibernate:

```
<class name="..." table="..." >
<composite-id name="propertyName" class="pk_class">
    <key-property name="propertyName" type="typename" column="column_name" />
    <key-property name="propertyName" type="typename" column="column_name" />
    ou <key-many-to-one name="propertyName" class="ClassName" column="column_name" />
    ....
</composite-id>
</class>
```

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 *classe 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;
    }
    ...
}

```

4. Structure générale d'une classe d'entité persistante

Le **code java** associé à une **classe d'objet persistant hibernate** correspond à un simple **JavaBean** (classe Java respectant les conventions de nom **getXxx()** et **setXxx(...)** au niveau des propriétés).

On parle quelquefois en terme de **POJO** (*Plain Old Java Object* / *Plain Ordinary Java Object*) pour désigner le fait qu'une API n'impose quasiment rien sur la structure des objets java .

Un **constructeur sans argument** est **indispensable**.

NB: Sous *eclipse* , les méthodes publiques "**getXxx()**" et "**setXxx(...)**" peuvent être générées automatiquement à partir des attributs privés et le menu contextuel "**source/generate getter/setter**"

De façon à ce qu'un objet persistant hibernate puisse être comparé à un autre (de façon efficace), il est quelquefois conseillé de reprogrammer convenablement les méthodes **equals** et **hashCode** héritées de la classe **Object**.

Cependant, cette **optimisation** (*facultative*) est **délicate** dans le sens où le code de **equals** et **hashCode** doit prendre en charge uniquement les valeurs élémentaires de l'objet lui même (entiers, réels, chaînes) mais **ne doit surtout pas tenir compte des références (éventuellement nulles) sur les objets reliés par des associations** et d'autre part ces méthodes peuvent éventuellement ne pas tenir compte de la clef primaire car elle n'a pas toujours de valeur bien affectée (ex: objet nouvellement créé) [--> en pratique : bien décocher les propriétés à ne pas prendre en compte lors de la génération de code via l'assistant eclipse "generate equals & hashCode"].

Exemple:

```
package entites;
public class Operation {
    private String id; // PK
    private String label;
    private double mouvement;
    private Compte compte; // référence vers objet associé (extrémité inverse "n-1" d'une relation "1-n"

    public String getId() { return id;}
    public void setId(String id) { this.id = id;}
    public String getLabel() {return label;}
    public void setLabel(String label) {this.label = label;}
    public double getMouvement() {return mouvement;}
    public void setMouvement(double mouvement) {this.mouvement = mouvement; }
    public Compte getCompte() {return compte;}
    public void setCompte(Compte compte) {this.compte = compte; }

    public boolean equals(Object other) {
        if (this == other) return true;    if ( !(other instanceof Operation) ) return false;
        final Operation op = (Operation) other;
        if ( !op.getLabel().equals( getLabel() ) ) return false;
        if ( op.getMouvement() != getMouvement() ) return false;
        return true;
    }
    public int hashCode() {
        int res= getLabel().hashCode();
        res=29*res + ((int) getMouvement());
        return res;
    }
}
```

+ habituelles annotations si JPA .

5. Propriétés d'une colonne

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

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

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

<i>Annotations JPA</i>	<i>Significations</i>
@OneToOne	1-1 (fk [unique] ---> pk) [<i><many-to-one unique='true'> de hibernate</i>]
@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 côté secondaire d'une relation **bidirectionnelle** ne seront pas automatiquement sauvegardées (tant que le côté 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:

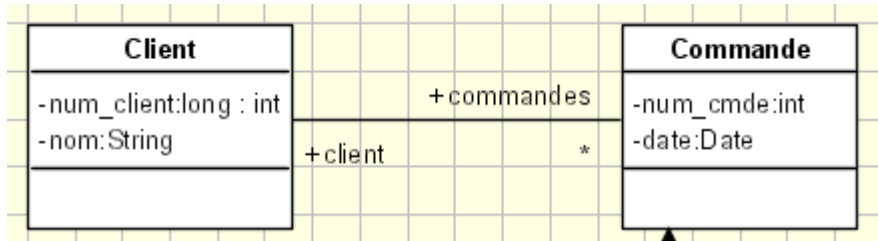
```
@OneToOne(cascade = {CascadeType.ALL})
@JoinColumn(name = "ADDRESS_ID")
public Address getAddress()
{
    return address;
}

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

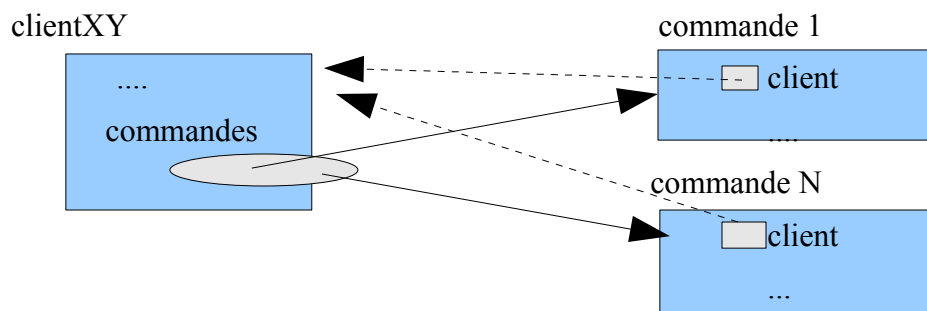
X - O.R.M. JPA – détails (1-n , 1-1 , n-n, ...)

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

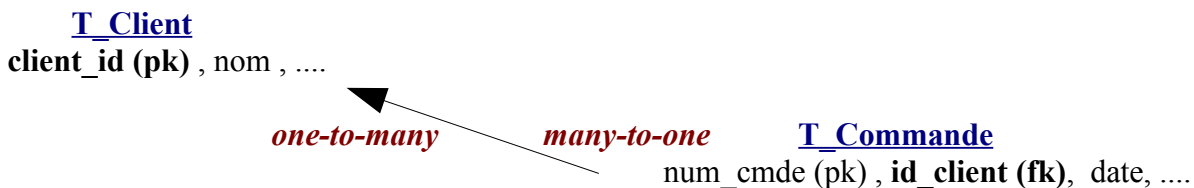
UML:



Java:



Base de données:



JPA:

```

@Entity
@Table(name = "T_Client")
public class Client {
    private Collection<Commande> commandes;
    ...
    @OneToMany(fetch=FetchType.LAZY,mappedBy="client")
    public Collection<Commande> getCommandes() { return commandes; }

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

```

@Entity
@Table(name = "T_Commande")
public class Commande {
    private Client client;
    ...
}
  
```


@ManyToOne**@JoinColumn(name = "id_client")**

```

public Order getClient() { return client; }

public void setClient(client client) { this.client = client; }
}

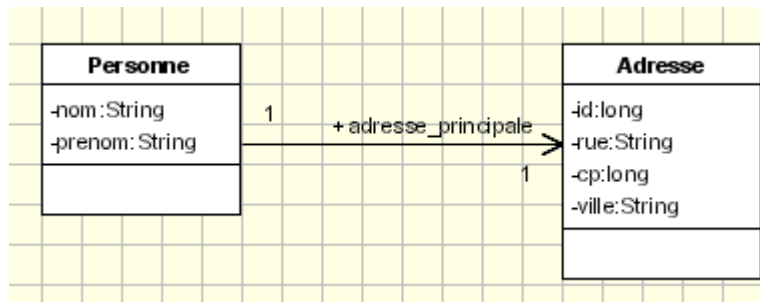
```

Rappels importants (pour JPA):

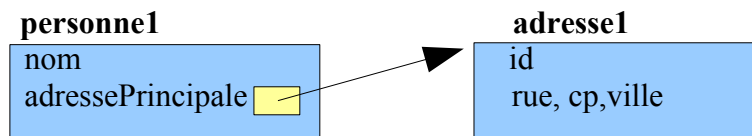
- La valeur de l'attribut "**mappedBy**" de l'annotation **@OneToMany** correspond à la **propriété qui sert à établir une correspondance inverse** au sein d'une **relation bidirectionnelle**.
- L'annotation **@JoinColumn** permet d'indiquer (via name="...") le nom de la colonne correspondant à la clef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

1.2. Relation 1-1

UML:



Java:



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

Base de données:

T_Personne

id (pk) , nom , prenom , , id_adr (fk)

@OneToOne (JPA)

ou

many-to-one unique="true"
(dans .hbm.xml d'Hibernate)

T_Adresse

id (pk) , rue , cp , ville

JPA:

```

@Entity
public class Personne
{

```

```

private Adresse adressePrincipale;
...
@OneToOne(cascade = {CascadeType.ALL})
@JoinColumn(name = "id_adr")
public Adresse getAdressePrincipale() {
    return adressePrincipale;
}

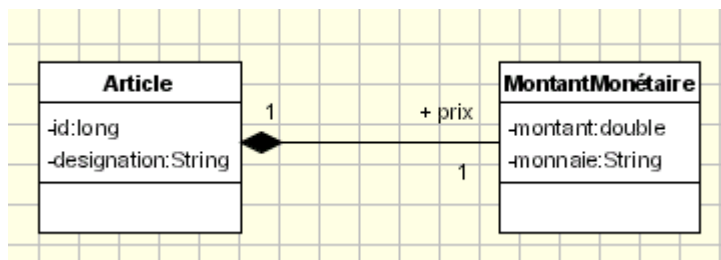
public void setAdressePrincipale(Adresse adressePrincipale) {
    this.adressePrincipale = adressePrincipale;
}
}

```

1.3. Sous objets imbriquables 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:

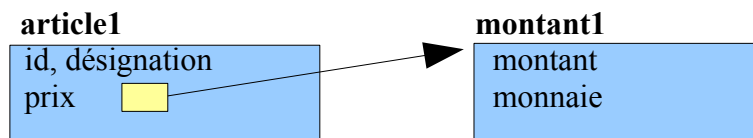
T Article

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 imbriqué 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")

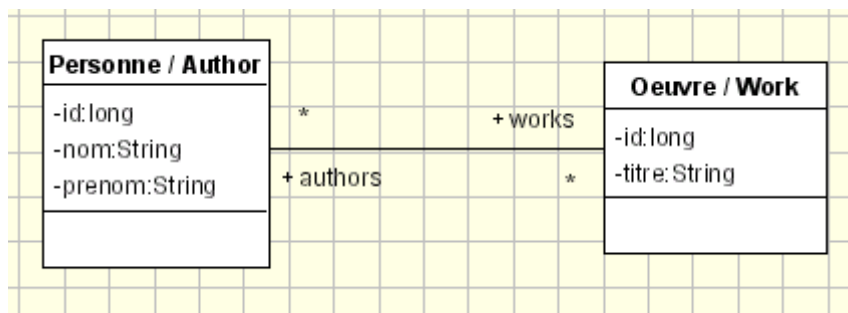
```
public class Article
{
    private long id; // avec get/set et @Id
    private MontantMonetaire prix;
    ...
}
```

@Embedded

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

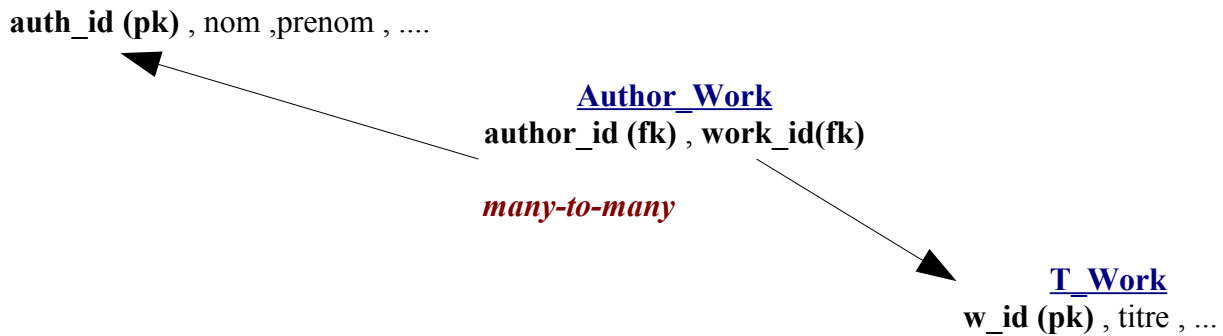
1.4. Relation n-n

UML:



Mapping relationnel ==> *Table intermédiaire*

T_Author



Java:

--> des collections dans les 2 sens

JPA:

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

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

    public void setAuthors(Collection<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 côté inverse/secondaire:

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

    public void set Works(Collection<Work> works) { this.works = works; }
}
```

Remarque 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 et les éléments de la table de jointure sont alors vus comme des entités ayant une clef primaire (éventuellement composée par les 2 clefs étrangères).

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

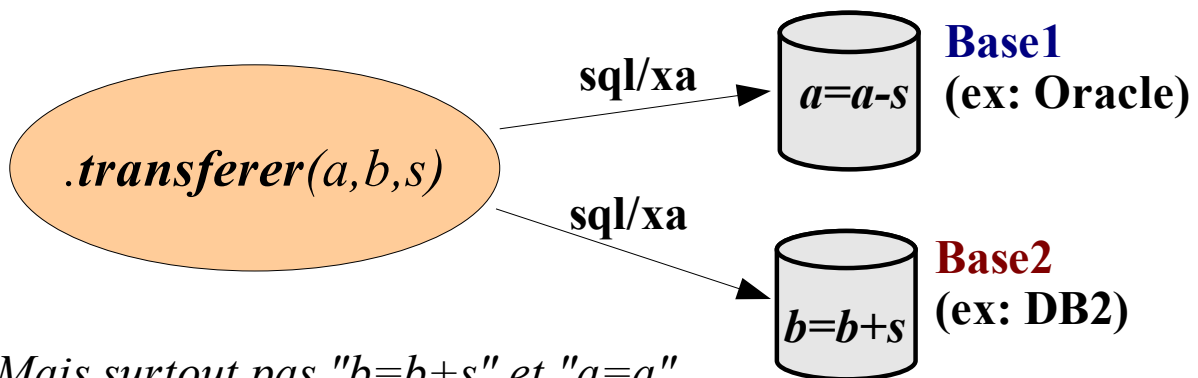
- **A.C.I.D.** ==> **A**tomicity , **C**onsistency , **I**solation , **D**urability
- L' **Atomicité** désigne le comportement "**tout ou rien**" (Le tout vu en tant qu'élément unique et atomique doit soit réussir , soit échouer). Il n'y a pas de demi-mesure.
- La **Consistance** d'une transaction désigne le fait que **les différentes opérations doivent laisser le système dans un état stable et cohérent.**
- Le concept d' **Isolation** signifie ici 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), ...

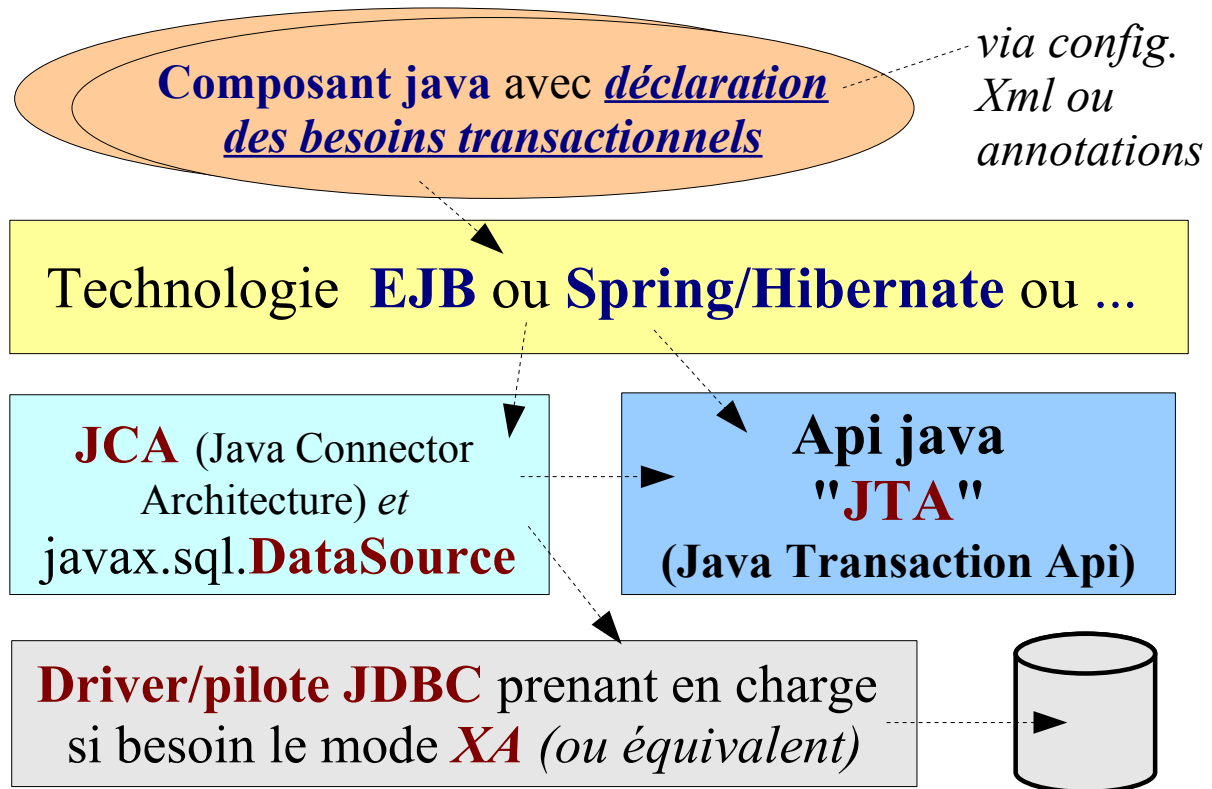


Protocole XA et commit à 2 phases (2)

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**.

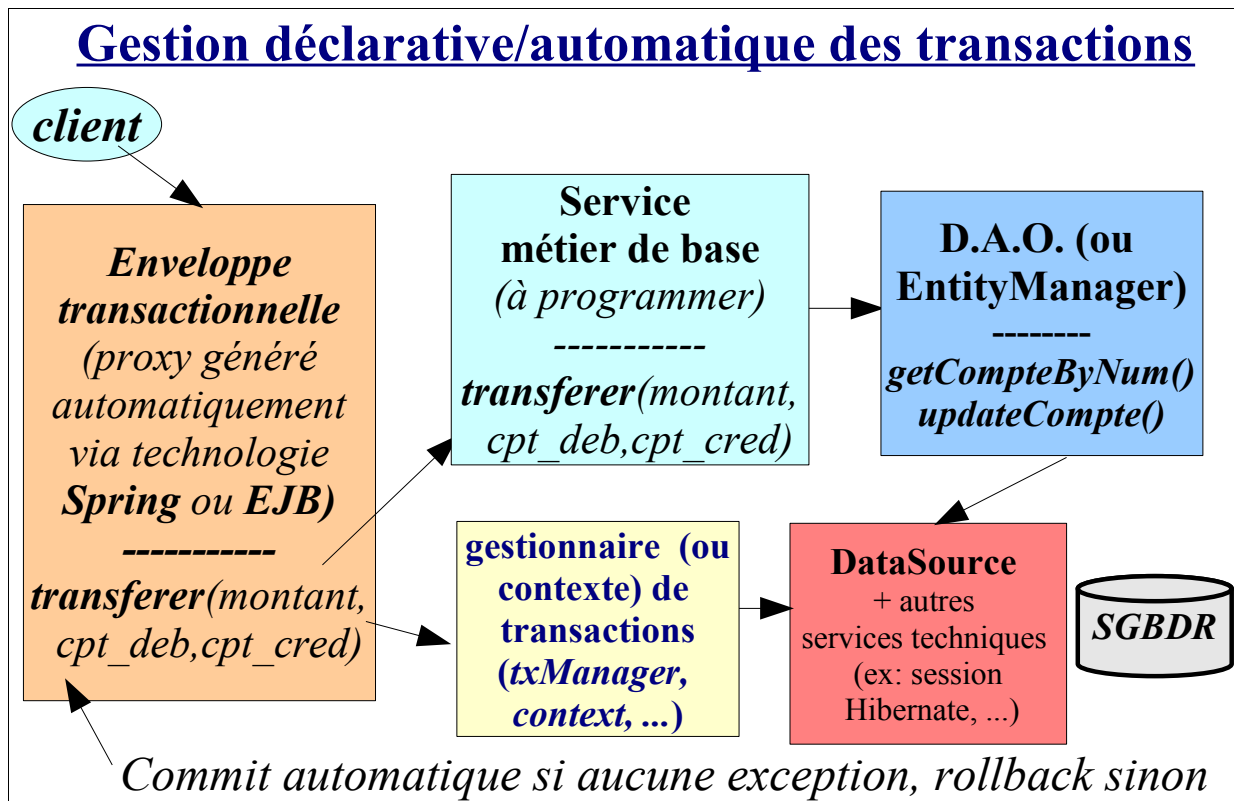
2. Infrastructure transactionnelle de JEE



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 à 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):

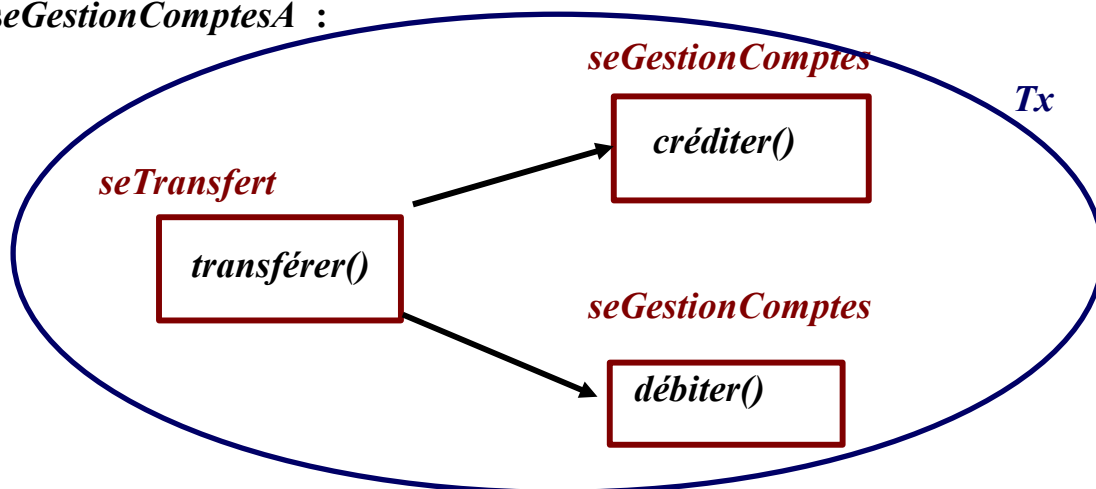
Activation d'une méthode *transférer()* sur un EJB Session *seTransfert*. Cette méthode étant associée à un attribut déclaratif valant "**Required**" et parce qu'aucune transaction existe pour l'instant une nouvelle transaction *Tx* est alors automatiquement créée et le contexte associé englobe *seTransfert.transférer()*



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.

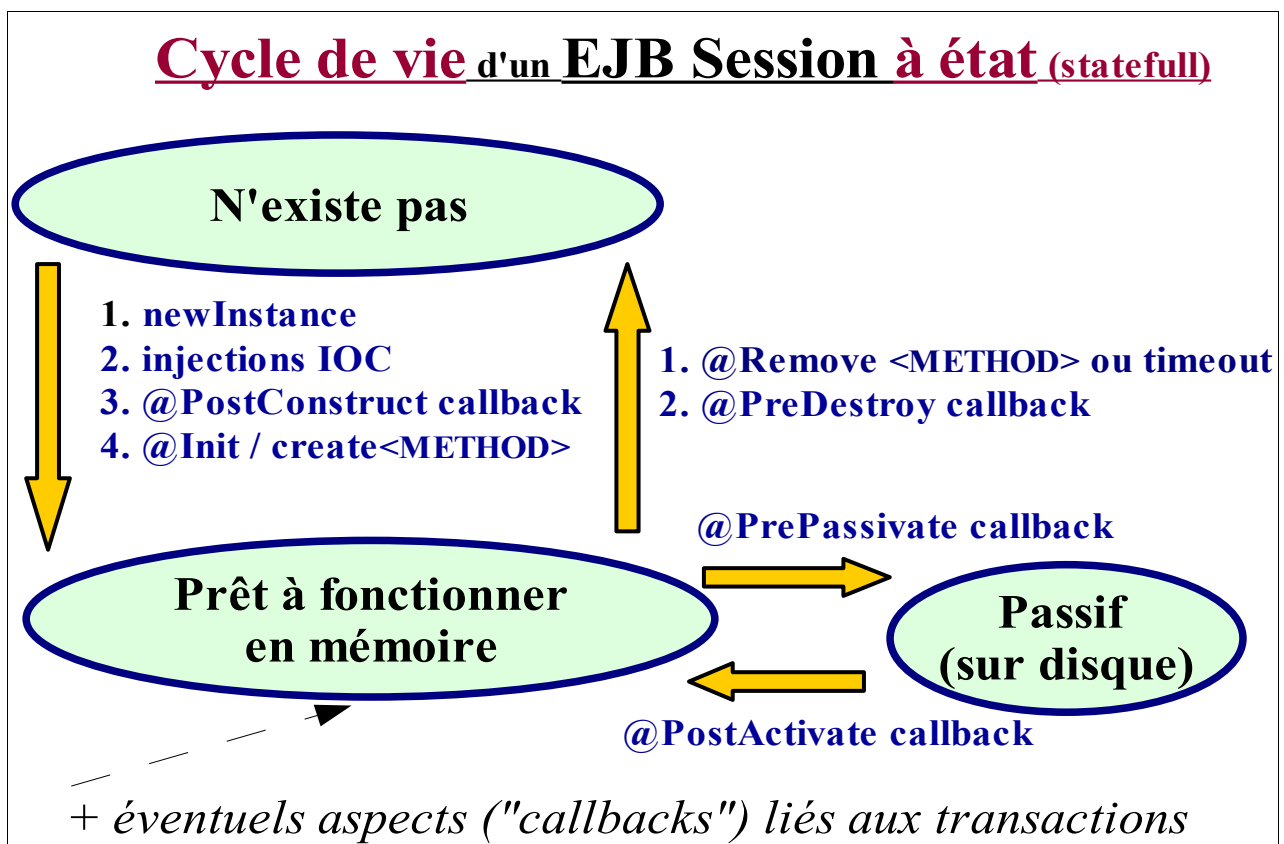
---> Les EJB3 fonctionnent donc en mode transactionnel même si rien n'est indiqué .

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 classiquement stockées au sein d'un EJB session à état :

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

NB: Lorsqu'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** (Message Oriented MiddleWare).

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.

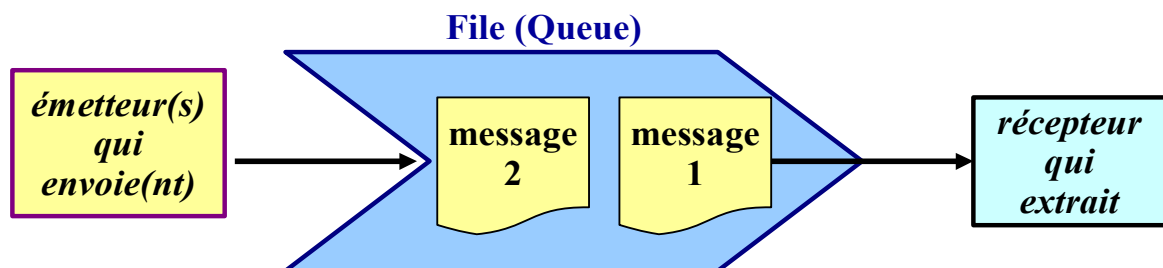
L'objet **ConnectionFactory** sera utilisé pour établir une connexion avec une file.

L'objet **Destination** (File ou Topic) sert à préciser la destination d'un message que l'on envoie ou bien la source d'un message que l'on souhaite récupérer.

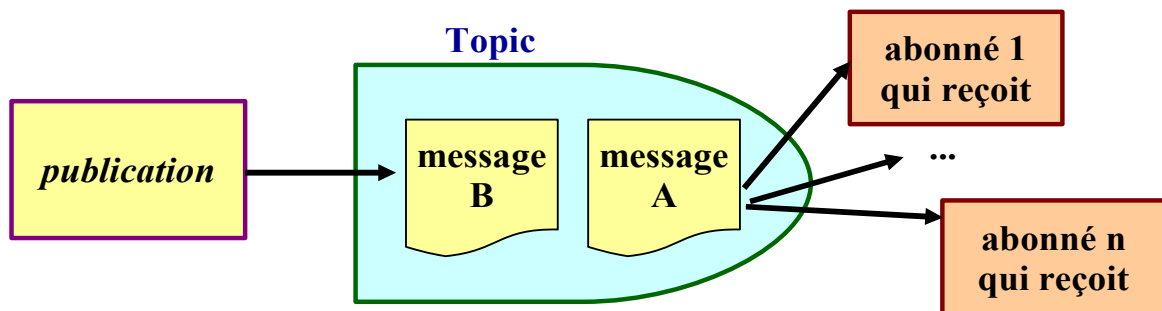
JMS permet de mettre en oeuvre les 2 modèles suivants:

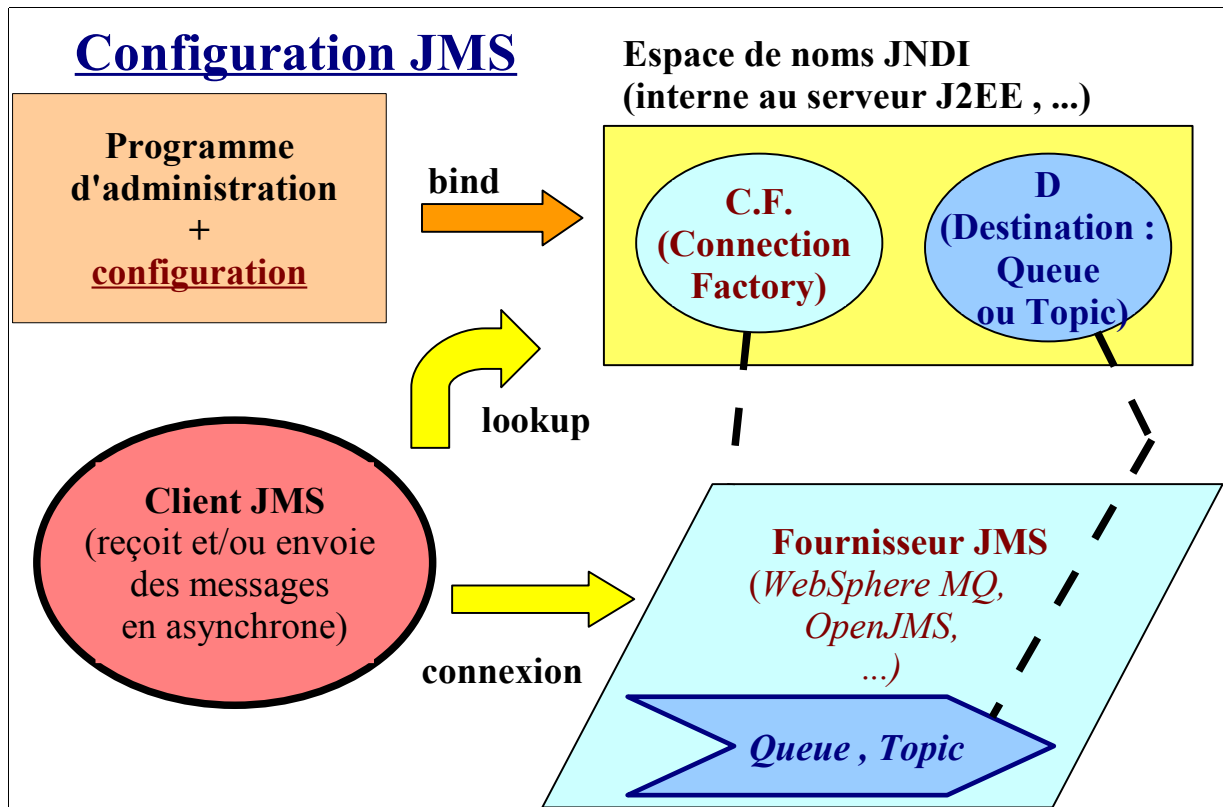
- **PTP** (Point To Point)
- **Pub/Sub** (Published & Subscribe) .../...

JMS Queue : Point To Point



JMS Topic : Publish / Subscribe





2. Configuration d'une file d'attente au sein de JBoss5

Depuis la version 5 de Jboss , le service interne gérant les files d'attentes asynchrones a évolué.

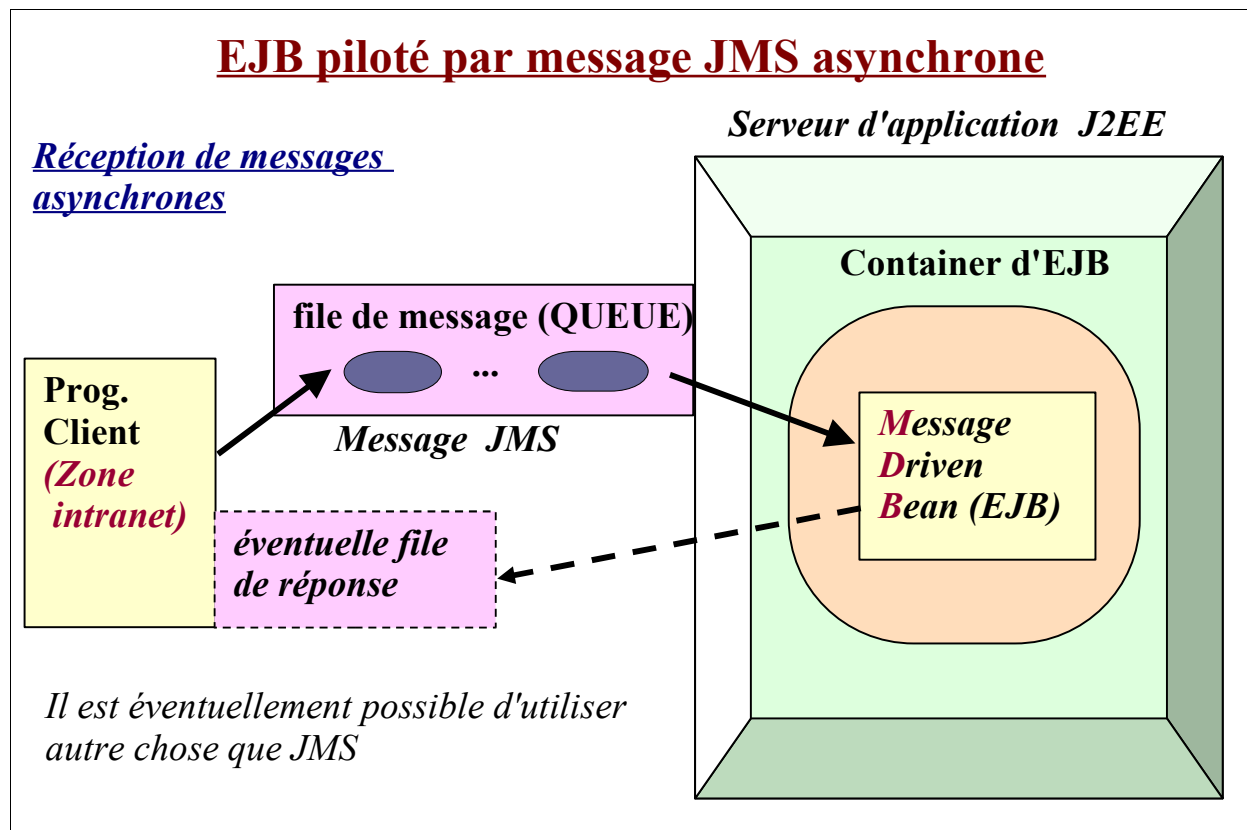
Il faut maintenant configurer le fichier "**destinations-service.xml**"

du répertoire "**jboss-5.1.0.GA\server\default\deploy\messaging**".

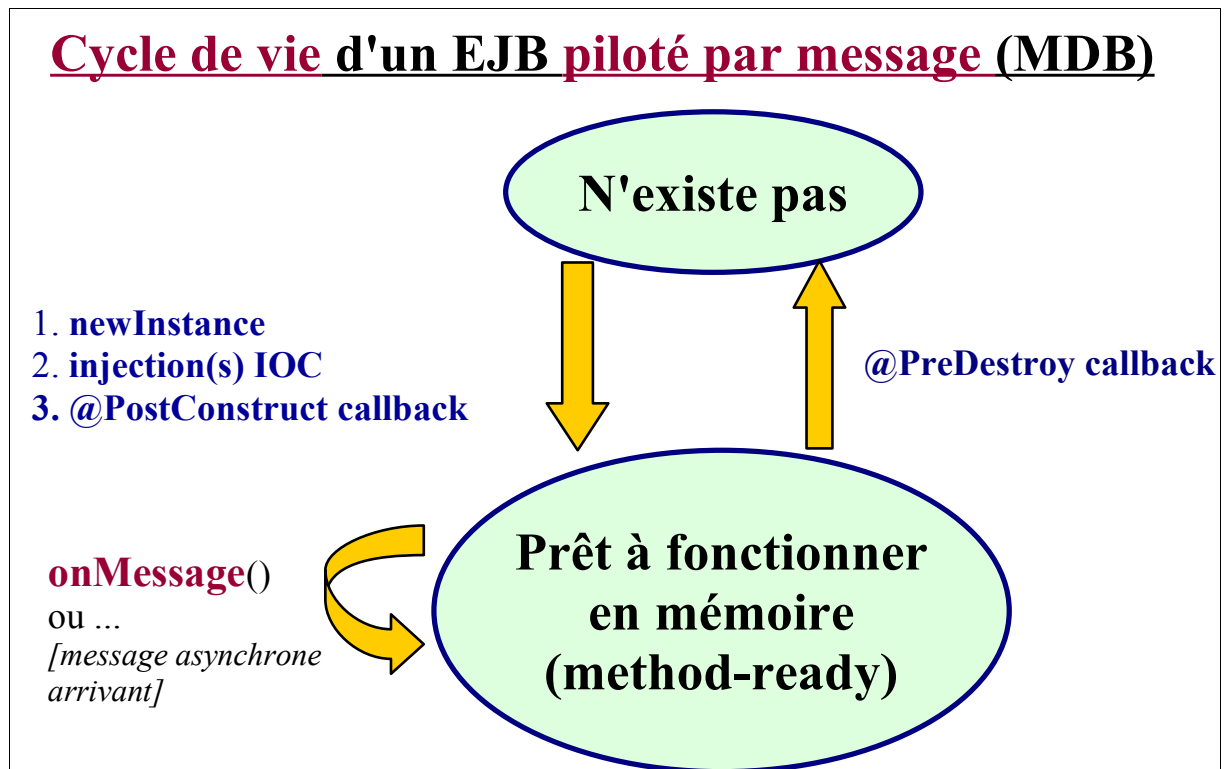
Exemple de configuration à ajouter:

```
<!-- calculator Queue a ajouter dans
      jboss-5.1.0.GA\server\default\deploy\messaging\destinations-service.xml -->
<mbean code="org.jboss.jms.server.destination.QueueService"
  name="jboss.messaging.destination:service=Queue,name=calculator"
  xmbean-dd="xmdesc/Queue-xmbean.xml">
  <depends optional-attribute-name="ServerPeer">
    jboss.messaging:service=ServerPeer</depends>
  <depends>jboss.messaging:service=PostOffice</depends>
</mbean>
```


3. EJB3 de type MDB



3.1. cycle de vie d'un EJB3 MDB



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

3.2. 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
{
    ...
}

```

3.3. 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 verifyInjection() {
        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 (JMSEException e) { e.printStackTrace(); }
        System.out.println("res="+res);
    }
}
}

```

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

clientJndi.properties

```

java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
#java.naming.provider.url=jnp://192.168.200.100:1099
java.naming.provider.url=jnp://localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming.client

```

ClientApp.java

```

...
import javax.jms.*;
...
public class ClientApp
{
    java.util.Properties props =null;

    private void init_client_env()
    {
        try {
            InputStream inStream = Thread.currentThread().getContextClassLoader()
                .getResourceAsStream("clientJndi.properties");
            this.props = new java.util.Properties();
            props.load(inStream);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) throws Exception {
        Client cli = new Client();    cli.init_client_env();
        cli.test_mdb();
    }
}

```

```
public static void test_mdb() throws Exception
{
    QueueConnection cnn = null;
    QueueSender sender = null;
    QueueSession session = null;
    InitialContext ctx = new InitialContext(this.props);
    QueueConnectionFactory factory = (QueueConnectionFactory)
        ctx.lookup("ConnectionFactory");
    Queue queue = (Queue) ctx.lookup("queue/calculator");

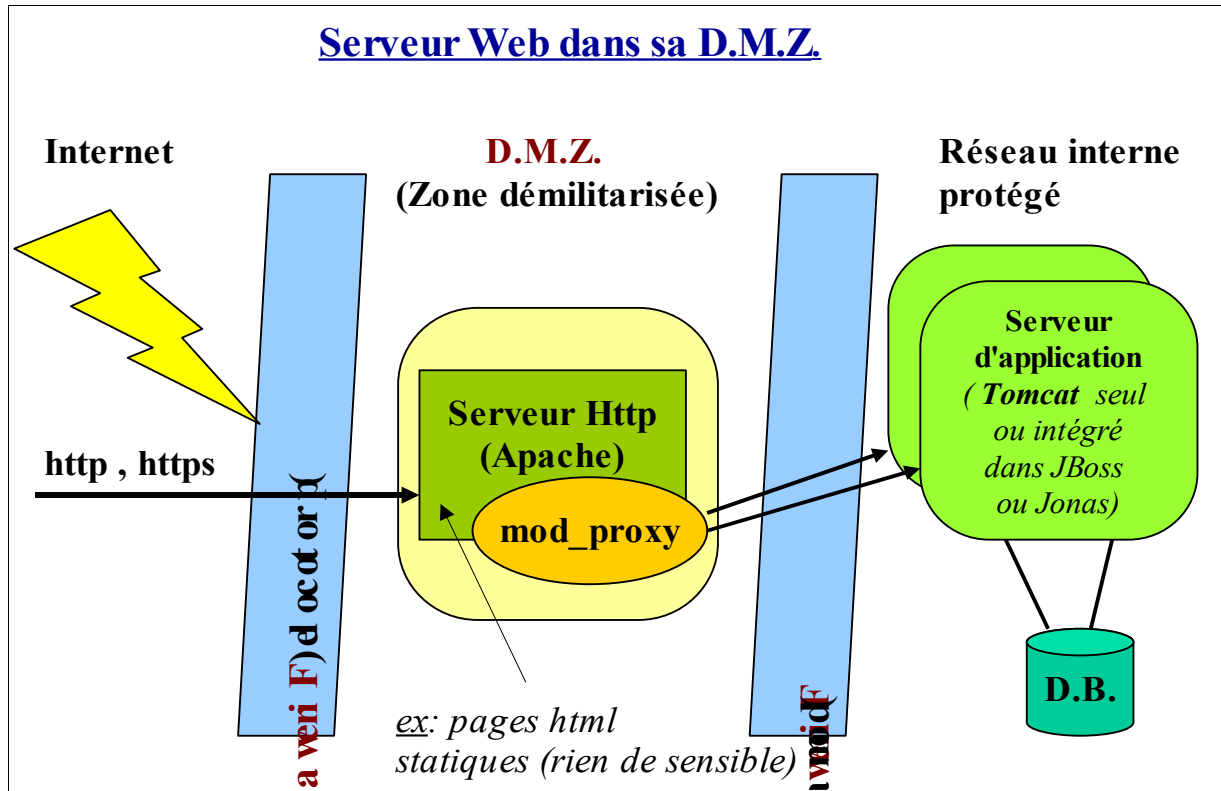
    cnn = factory.createQueueConnection();
    session = cnn.createQueueSession(false, QueueSession.AUTO_ACKNOWLEDGE);

    MapMessage msg = session.createMapMessage();
    msg.setInt("a",5);
    msg.setString("op","+");
    msg.setInt("b",6);

    sender = session.createSender(queue);
    sender.send(msg);
    System.out.println("Message sent successfully to remote queue.");
}
```

XIV - Sécurité JEE (au niveau des EJB)

1. D.M.Z. et Firewalls

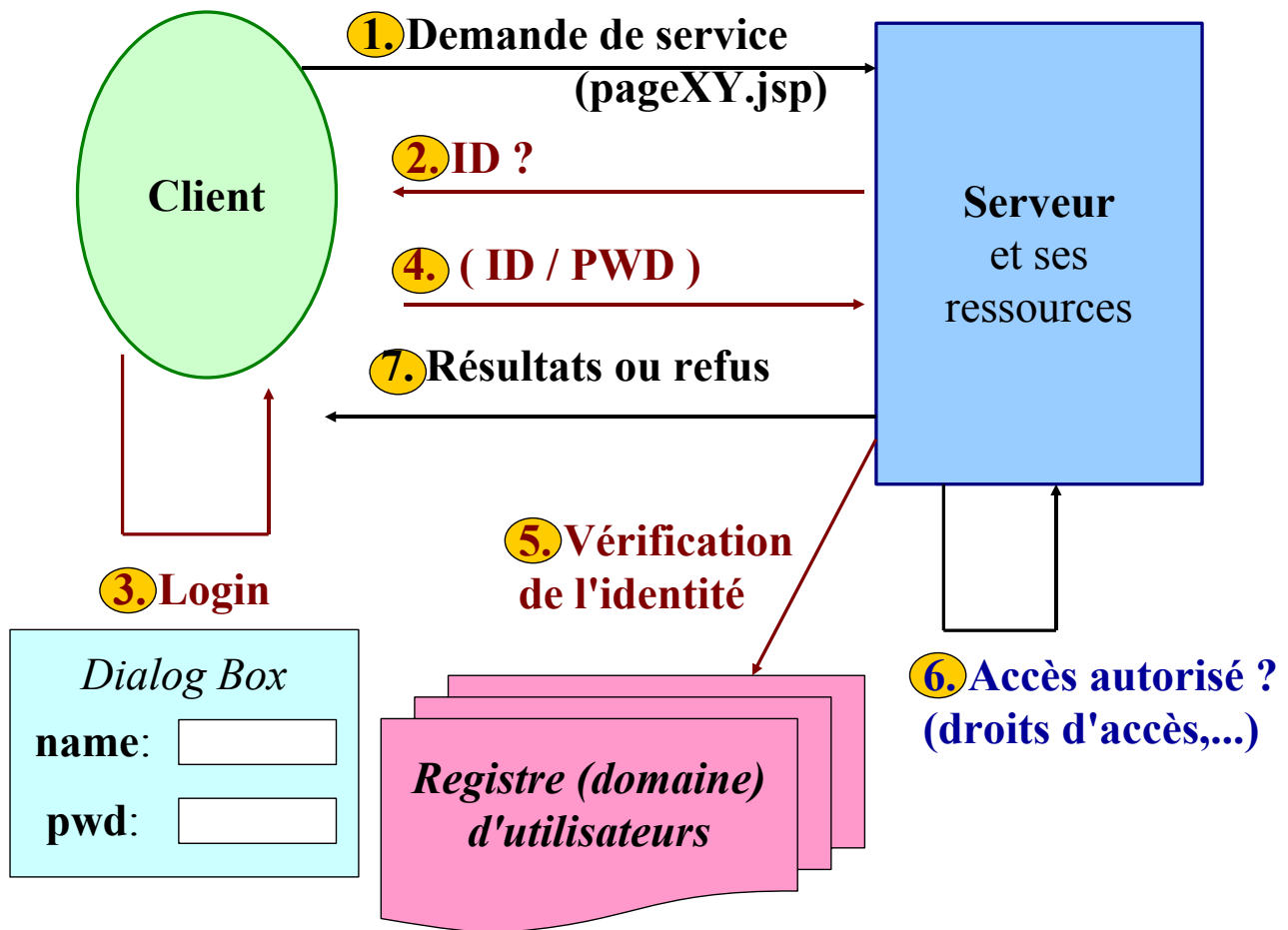


2. Sécurité J2EE/JEE5

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



3. Rôles associés aux EJB3 (via annotations)

Les annotations `@DeclareRoles` , `@RoleAllowed` 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.

<code>@DeclareRoles</code>	permet de déclarer une liste de rôles existants (et à prendre en compte)
<code>@RoleAllowed</code>	permet d'indiquer une liste de rôles dont au moins un est requis.
<code>@PermitAll</code>	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 myejb;

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
// pas de @PermitAll ici avec Jboss 5.1 !!! ??? !!!
@DeclareRoles({"student","teacher"})
public class CalculatorBean implements Calculator
{
    @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 subtract(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 J2EE .
- associer quelques utilisateurs (ou groupe d'utilisateurs) fictifs aux rôles de l'application.
- coder ou paramétrer une authentification du côté client.

Dans le cas du serveur open source JBoss , la configuration d'un domaine de sécurité s'effectue en ajoutant un bloc Xml du type suivant dans le fichier **server/default/conf/login-config.xml** :

```
<application-policy name = "my-security-domain">
  <authentication>
    <login-module
      code = "org.jboss.security.auth.spi.UsersRolesLoginModule"
      flag = "required">
      <module-option name="usersProperties">
        security/users.properties</module-option>
      <module-option name="rolesProperties">
        security/roles.properties</module-option>
    </login-module>
  </authentication>
</application-policy>
```

NB: depuis la version 5.1 de Jboss ce paramétrage peut également être effectué dans un fichier "xxx-jboss-beans.xml" à déposer/déployer dans le répertoire server/default/deploy :

myJeeApp-jboss-beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- NB: ce fichier (se terminant par -jboss-beans.xml) est prévu
      doit être recopié dans le répertoire server/default/deploy -->

<deployment xmlns="urn:jboss:bean-deployer:2.0">
<application-policy xmlns="urn:jboss:security-beans:1.0"
      name="my-security-domain">
<authentication>
<login-module code = "org.jboss.security.auth.spi.UsersRolesLoginModule"
      flag = "required">
<module-option name = "unauthenticatedIdentity">anonymous</module-option>
<module-option name="usersProperties">security/users.properties</module-option>
<module-option name="rolesProperties">security/roles.properties</module-option>
</login-module>
</authentication>
</application-policy>
</deployment>
```

Le référencement de ce domaine de sécurité Jboss s'effectue au niveau de l'application J2EE en ajoutant un bloc xml **<security-domain>** dans le fichier **META-INF/jboss.xml** du module d'EJB de l'application:

```
<?xml version="1.0"?>
<jboss xmlns="http://www.jboss.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
        http://www.jboss.org/j2ee/schema/jboss_5_0.xsd"
      version="5.0">
  <security-domain>java:jaas/my-security-domain</security-domain>
</jboss>
```


Conformément au paramétrage du domaine de sécurité "*my-security-domain*" déclaré dans *login-config.xml* les utilisateurs (avec leurs mots de passe) et les rôles associés doivent être renseignés dans les fichiers suivants (recherchés dans le *classpath*) :

security/users.properties

```
#utilisateur=mot_de_passe
s1=s1pwd
s2=s2pwd
t1=t1pwd
```

security/roles.properties

```
#utilisateur=role
s1=student
s2=student
t1=teacher
```

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 si **WEB-INF/jboss-web.xml** comporte aussi

```
<security-domain>java:jaas/my-security-domain</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 Jaas et Jndi)

Fichier de paramétrage indispensable pour l'api JAAS:

jaas-standard.cfg

```
// NB: ce fichier peut s'appeler jaas-standard.cfg ou bien auth.conf ou ....
// il correspond à la valeur de la propriété java.security.auth.login.config (clientJndi.properties)
// exemple: java.security.auth.login.config=
// file:///C:\\Prog\\java\\ServApp\\jboss-5.1.0.GA\\server\\default\\conf\\jaas-standard.cfg
// si on place ce fichier dans le répertoire jboss-5.1.0.GA\\server\\default\\conf
other {
    org.jboss.security.ClientLoginModule required;
};
```

Fichier de paramétrage indispensable pour le programme client (vis à vis de Jboss et de ses ejb) :

clientJndi.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
j2ee.clientName=myJeeAppClient
java.security.auth.login.config=file:///C:\\Prog\\java\\ServApp\\jboss-5.1.0.GA\\server\\default\\conf\\jaas-standard.cfg
```

Exemple de code client prenant en compte une authentification:

```

public class Client
{
    java.util.Properties props =null;

    private void init_client_env()    {
        try {
            InputStream inStream = Thread.currentThread().getContextClassLoader()
                                   .getResourceAsStream("clientJndi.properties");
            this.props = new java.util.Properties();
            props.load(inStream);
        } catch (IOException e) {e.printStackTrace();}
    }

    public static void main(String[] args) throws Exception
    {
        Client cli = new Client();
        cli.init_client_env();
        cli.test_calculator_with_jboss_security();
        cli.test_calculator_with_security_jaas(); // interaction (username,pwd)
    }

    public void test_calculator_with_jboss_security() throws Exception
    {
        System.out.println("***** test_calculator_with_security *****");
        Properties env = (Properties) this.props.clone();
        // NB: la propriete "java.security.auth.login.config" est pour JAAS
        // elle doit donc etre placee dans les propriétés systèmes :
        System.setProperty("java.security.auth.login.config",
                           env.getProperty("java.security.auth.login.config"));
        // pour JNDI (en version "security" ):
        env.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                        "org.jboss.security.jndi.LoginInitialContextFactory");
        // Establish the proxy with an incorrect security identity
        env.setProperty(Context.SECURITY_PRINCIPAL, "s1");
        env.setProperty(Context.SECURITY_CREDENTIALS, "invalidpassword");
        InitialContext ctx = new InitialContext(env);
        Calculator calculator = (Calculator)
                               ctx.lookup("myJeeApp/CalculatorBean/remote");
        System.out.println("s1 is a student.");
        System.out.println("s1 types in the wrong password");
        System.out.flush();
        try { System.out.println("1 + 1 = " + calculator.add(1, 1));
        } catch (Exception ex){
            System.err.println("Saw expected SecurityException (add): "
                               + ex.getMessage()); System.err.flush();
        }
        System.out.flush();
        System.out.println("s1 types in correct password.");
        System.out.println("s1 does unchecked addition.");
        // Re-establish the proxy with the correct security identity
        env.setProperty(Context.SECURITY_CREDENTIALS, "s1pwd");
        ctx = new InitialContext(env);
        calculator = (Calculator) ctx.lookup("myJeeApp/CalculatorBean/remote");
        System.out.println("1 + 1 = " + calculator.add(1, 1));
        System.out.flush();
        System.out.println("s1 is not a teacher so he cannot do division");
        try { System.out.println("16 / 4 = " +calculator.divide(16, 4));
            System.out.flush(); }
        catch (Exception ex) {
            System.err.println("pour division non autorisee:" + ex.getMessage());
            System.err.flush();
        }
        System.out.flush();
    }
}

```

```

        System.out.println("Students are allowed to do subtraction");
        System.out.println("1 - 1 = " + calculator.subtract(1, 1));
        System.out.flush();
    }

    public void test_calculator_with_security_jaas() throws Exception
    {
        System.out.println("***** test_calculator_with_jaas_security *****");
        // Establish the proxy with an incorrect security identity
        Properties env = (Properties) this.props.clone(); //
        System.setProperty("java.security.auth.login.config",
            env.getProperty("java.security.auth.login.config"));
        //DialogCallbackHandler handler = new DialogCallbackHandler();
        TextCallbackHandler handler = new TextCallbackHandler();
        LoginContext lc = new LoginContext("other", handler); // JAAS
        System.out.println("(s1,invalidpwd) is a invalid user");
        System.out.println("(s1,s1pwd) and (s2,s2pwd) are students");
        System.out.println("(t1,t1pwd) is a teatcher");
        System.out.flush();
        lc.login(); // demandera (username,password)
        InitialContext ctx = new InitialContext(env);
        Calculator calculator = (Calculator)
            ctx.lookup("myJeeApp/CalculatorBean/remote");

        try {
            System.out.println("1 + 1 = " + calculator.add(1, 1));
        } catch (Exception ex) {
            System.err.println("Saw expected SecurityException (add): "
                + ex.getMessage());
        }
        System.out.flush();
        try {
            System.out.println("16 / 4 = " + calculator.divide(16, 4));
        }
        catch (Exception ex) {
            System.err.println("Saw expected SecurityException (divide): "
                + ex.getMessage());
        }
        System.out.flush();
        try {
            System.out.println("1 - 1 = " + calculator.subtract(1, 1));
        }
        catch (Exception ex) {
            System.err.println("Saw expected SecurityException (subtract): "
                + ex.getMessage());
        }
        lc.logout();
    }
}

```

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 J2EE].

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

1. Timer sur EJB3 (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);
}
```

2. 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 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 ,

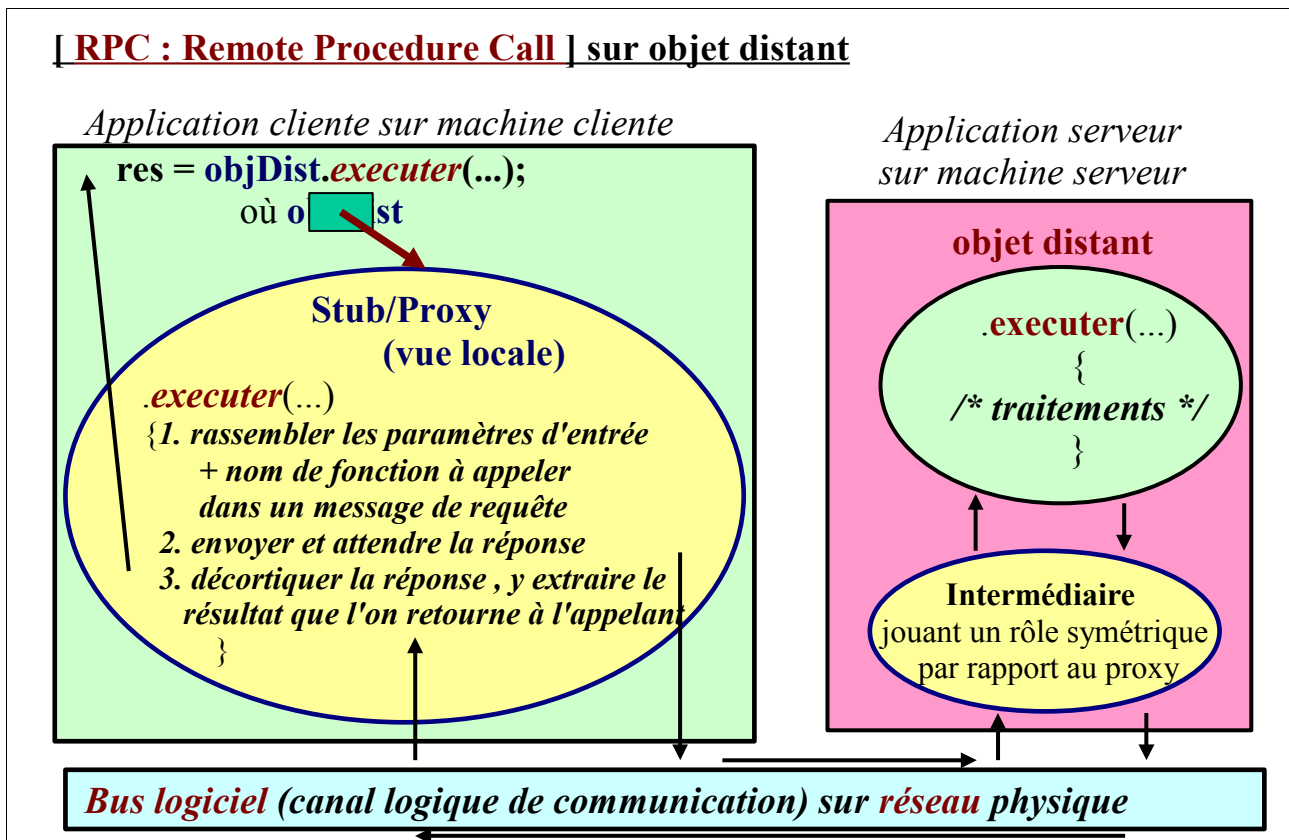
.... passer par dessus ?

... passer par dessous ?

Impossible , il faut le traverser !

2. Principe général des RPC

[**RPC : Remote Procedure Call**] sur objet distant



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 côté serveur.

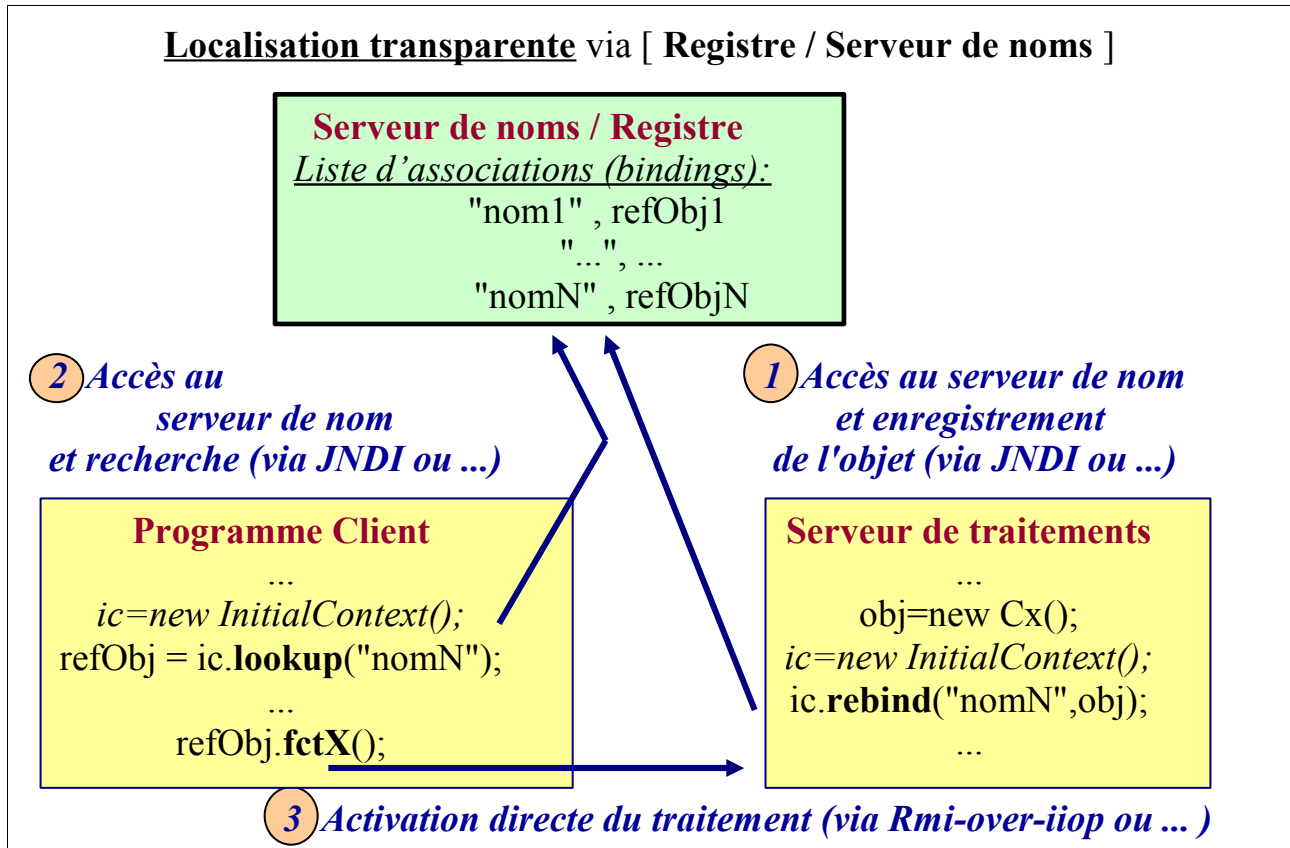
Quelque soit la technologie utilisée (CORBA, RMI, COM+ , Service WEB , ...), le code (très technique) du **stub** est toujours **généré automatiquement** (via un générateur de code ou via un mécanisme complètement dynamique).

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'à 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
-

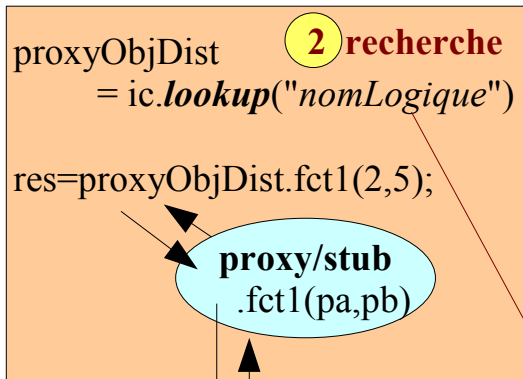
==> Le code client est dépendant du type de serveur de noms qu'il faut utiliser.

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

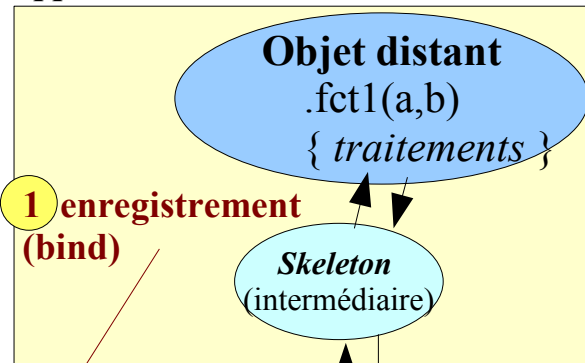
Bus logiciel (*canal de communication pour objets distribués*)

Application cliente (sur machine cliente)



3 appel
distant

Application serveur (sur machine serveur)



Service de noms

Bus logiciel (canal de communication)

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:

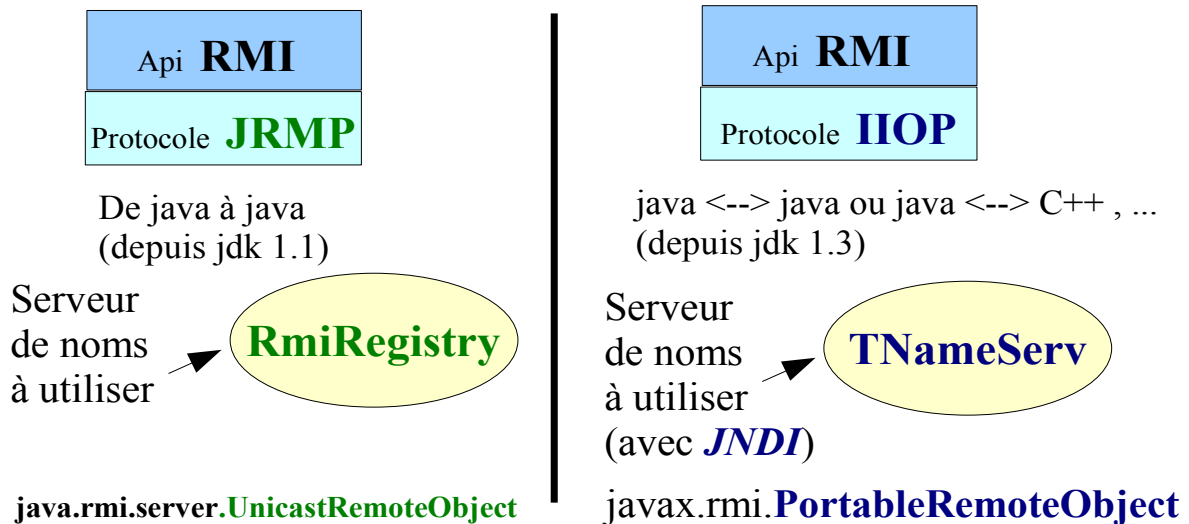
Simple Object Acces Protocol (dialogue Xml sur HTTP)
Interopérabilité totale / Protocole des "service web" .

DCOM/COM+ : Protocole propriétaire de Microsoft .

6. Présentation api RMI (Remote Method Invocation)

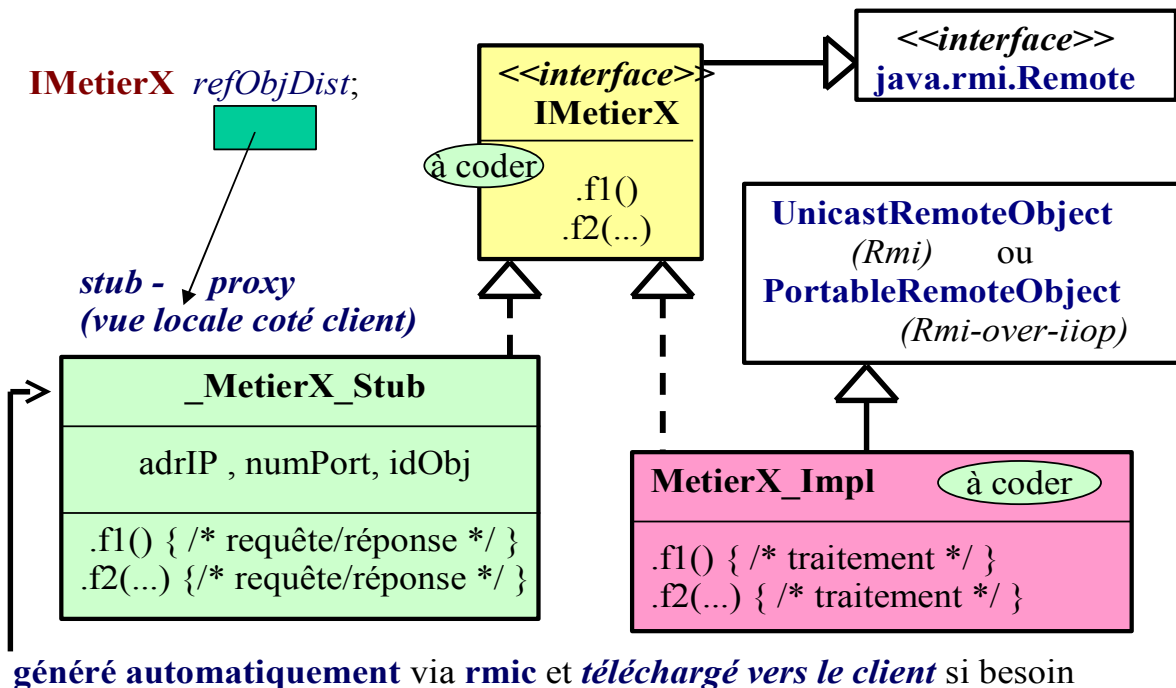
RMI (Remote Method Invocation)

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



6.1. Structure du code java (RMI)

R.M.I. – Structure du code java



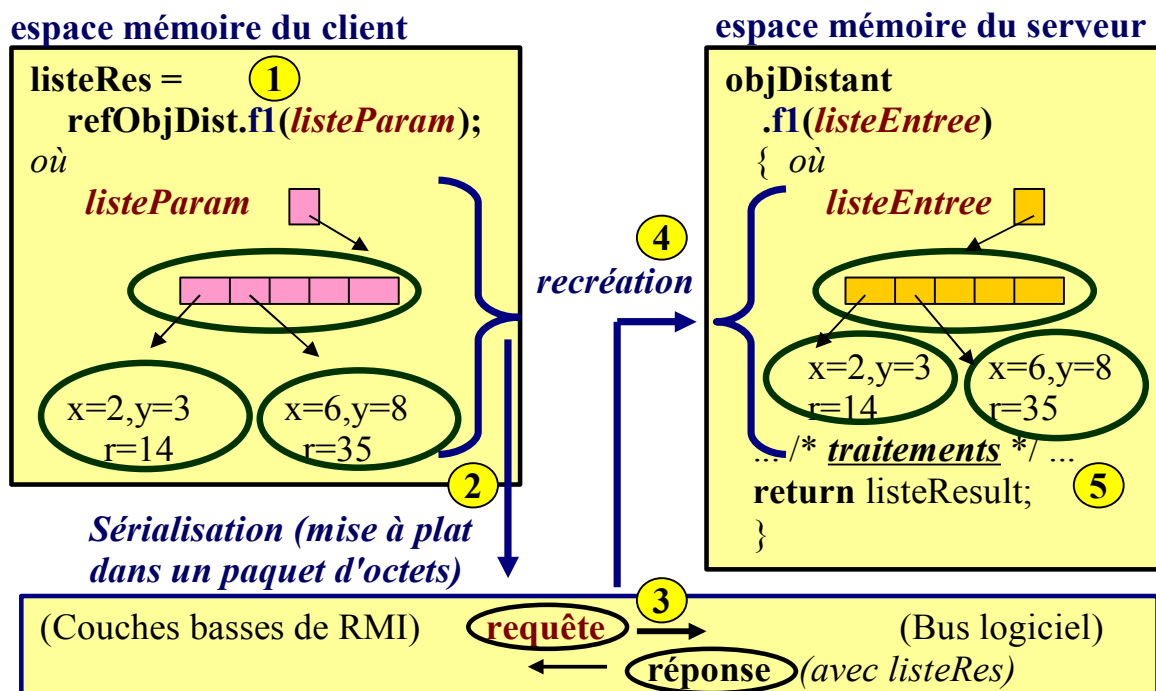
6.2. Code type de l'interface de l'objet distant (RMI):

```
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:

Spécificités de R.M.I. - sérialisation



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 ici 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 où 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\jdk1.4.2_06\bin
REM   chemins menant aux 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 propriété 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_SRV_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

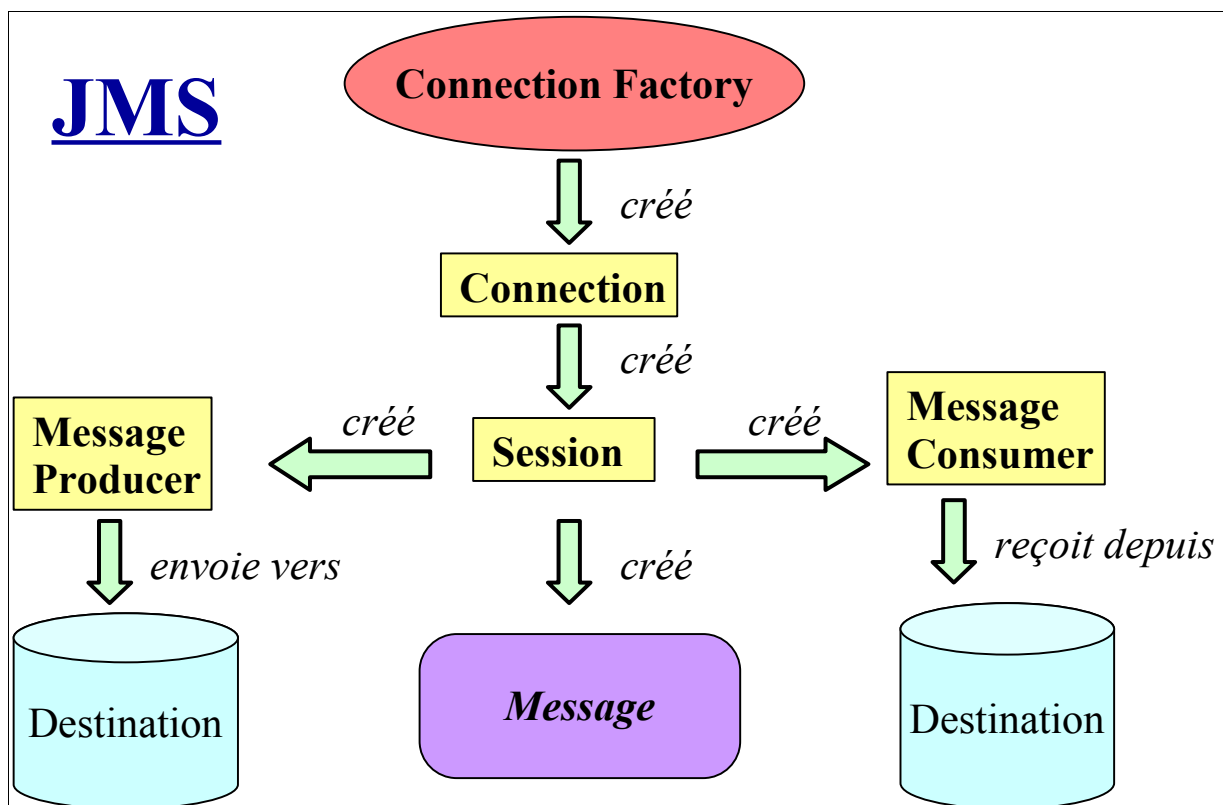
2 modèles de destination :

- **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 Connexion:

```
QueueSession session;
session = queueConnection.createQueueSession(false /*transact. */,
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.createBytesMessage();  
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	PERSISTENT ou NON_PERSISTENT	méthode send()
JMSExpiration	0 : pas d'expiration. sinon <i>n ms</i> 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** .

<i>mode d'acquittement</i>	<i>caractéristiques</i>
Session.CLIENT_ACKNOWLEDGE	il faut appeler explicitement <code>message.acknowledge()</code> .
Session.AUTO_ACKNOWLEDGE	automatique en fin de <i>OnMessage</i> ou en fin de <i>receive()</i>
Session.DUPS_OK_ACKNOWLEDGE	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).

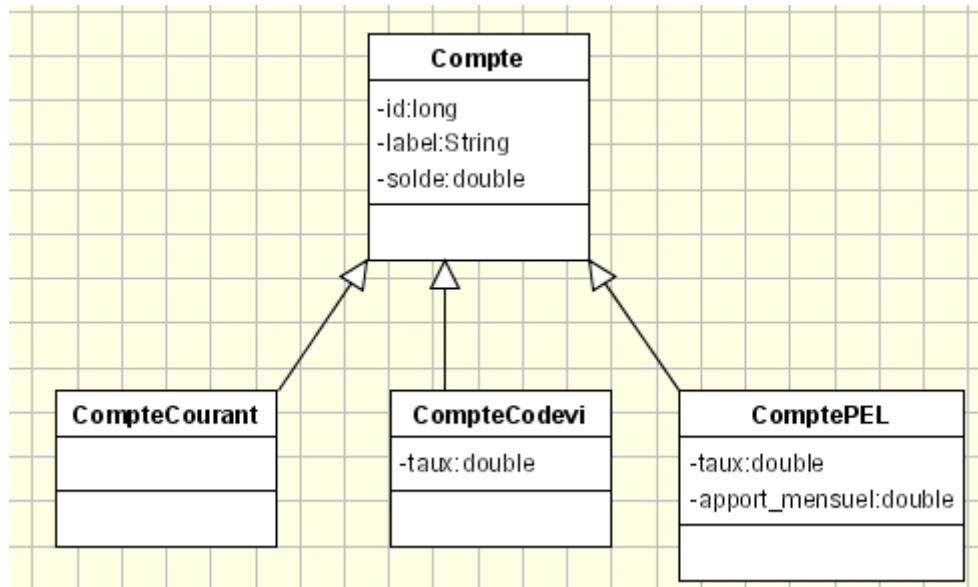
Produits (FreeWare - OpenSource):

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

XVII - Annexe – Détails sur JPA

1. Relations d'héritage & polymorphisme

UML:



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

Stratégie "une table par hiérarchie de classes"
(avec colonne discriminante)

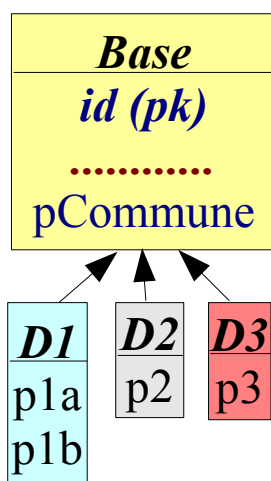


Table "Base_et_dérivés"

<i>id</i>	<i>typeDérivé</i> (discriminateur)	<i>pC.</i>	<i>p1a</i>	<i>p1b</i>	<i>p2</i>	<i>p3</i>
1	D1	xx	aa	bb	Null	Null
2	D2	yy	Null	Null	22	Null
3	D3	zz	Null	Null	Null	33

Colonne discriminante (*typeDérivé*)

Null (N/A)

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 entre 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)
public class Compte
{
    private long id;
    private String label;
    private double solde;

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    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; }

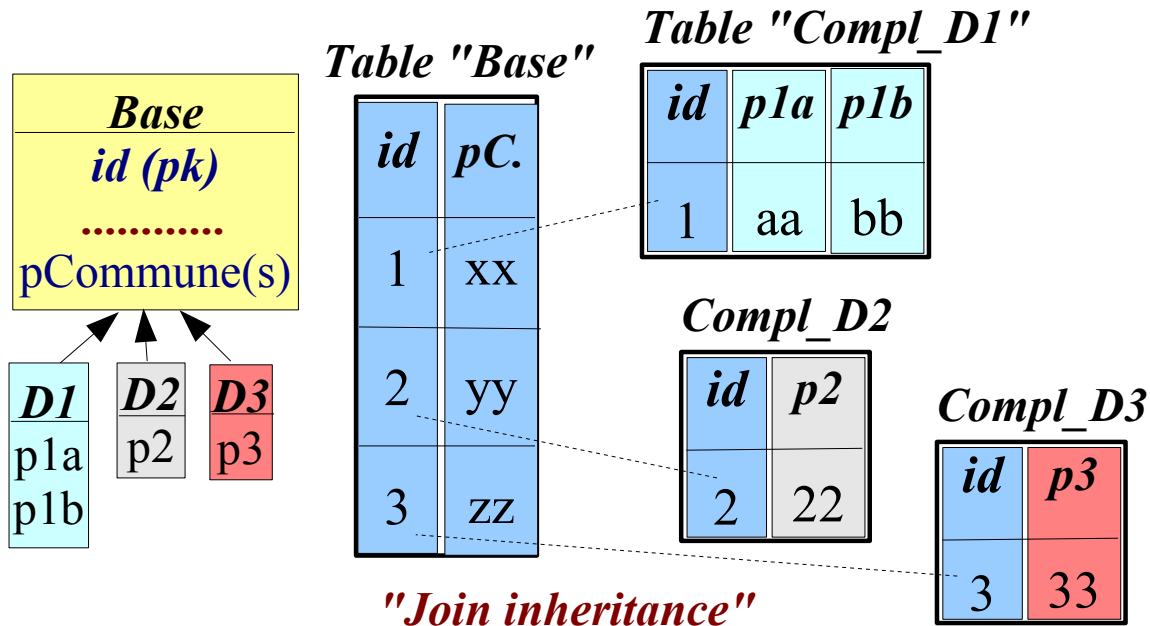
    public double getSolde() { return solde;}
    public void setSolde(double solde) {this.solde = solde; }
}
```

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(discriminatorType = DiscriminatorType.STRING)
@DiscriminatorValue("PEL")
public class ComptePEL extends Compte
{
    private double tauxInteret; //+get/set
    private double apportMensuel; //+get/set
    ...
}
```

1.2. Stratégie "Join_inheritance"

Stratégie "table principale

+ une table (de compléments) par classe fille"



```
@Entity
@Table(name="Compte")
@Inheritance(strategy=InheritanceType.JOINED )
public class Compte
{
    private long id;
    private String label; // +get/set
    private double solde; // +get/set

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    ...
}
```

```
@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")
    public String getStreet() { return street; }
    public void setStreet(String street) { this.street = street; }

    @Column(name = "CITY", table = "EMBEDDED_ADDRESS")
    public String getCity() { return city; }
    public void setCity(String city) { this.city = city; }
}
```

==> dans ce cas (clef primaire table_secondeire = 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"

<i>Annotations (callback)</i>	<i>déclenchement (étape du cycle de persistance)</i>
@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 ...

*Ex: Liste de
composants (EJB, ...)
et de ressources
techniques (pool, ...)*

*Ex: annuaire LDAP
(employés, ...)*

- **Des services de noms** : liste d'associations (bindings) entre des ressources et des noms logiques.
- **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.

Recherche (via JNDI) de ressources enregistrées avec des noms logiques

Serveur de noms (souvent intégré ...)

Liste d'associations (bindings):

"nom1" , refObj1
"nomN" , refObjN

② Accès au
serveur de nom et
recherche via JNDI

Client

```
...
ic=new InitialContext();
ressource = ic.lookup("nomN");
...
ressource.fctX();
```

③ utilisation
de la ressource

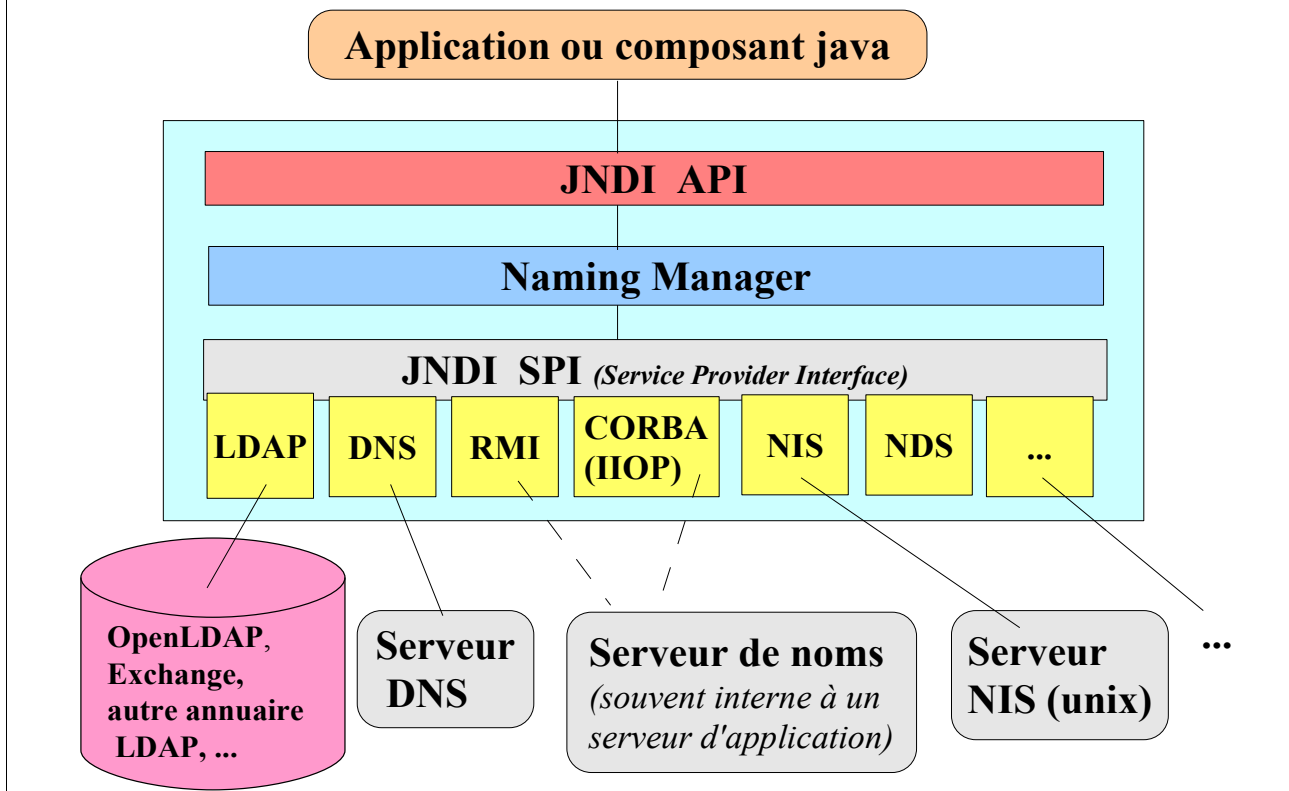
① Accès au serveur de nom
et enregistrement
de la ressource via JNDI

Serveur de ressources

(ex: Pool de connexions, objet RMI, ...)

```
...
objRessource=new CRessource();
ic=new InitialContext();
ic.rebind("nomN", objRessource);
...
```

Structure de JNDI



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
```

XVIII - 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>
  </dependencies>
</project>
```

```

....
...
<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}
  </persistence-unit>
</persistence>

```

```

    <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);
    }
}

```

XIX - 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 eclipse (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/invoction 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 , ...)
-