



Enterprise Java Beans

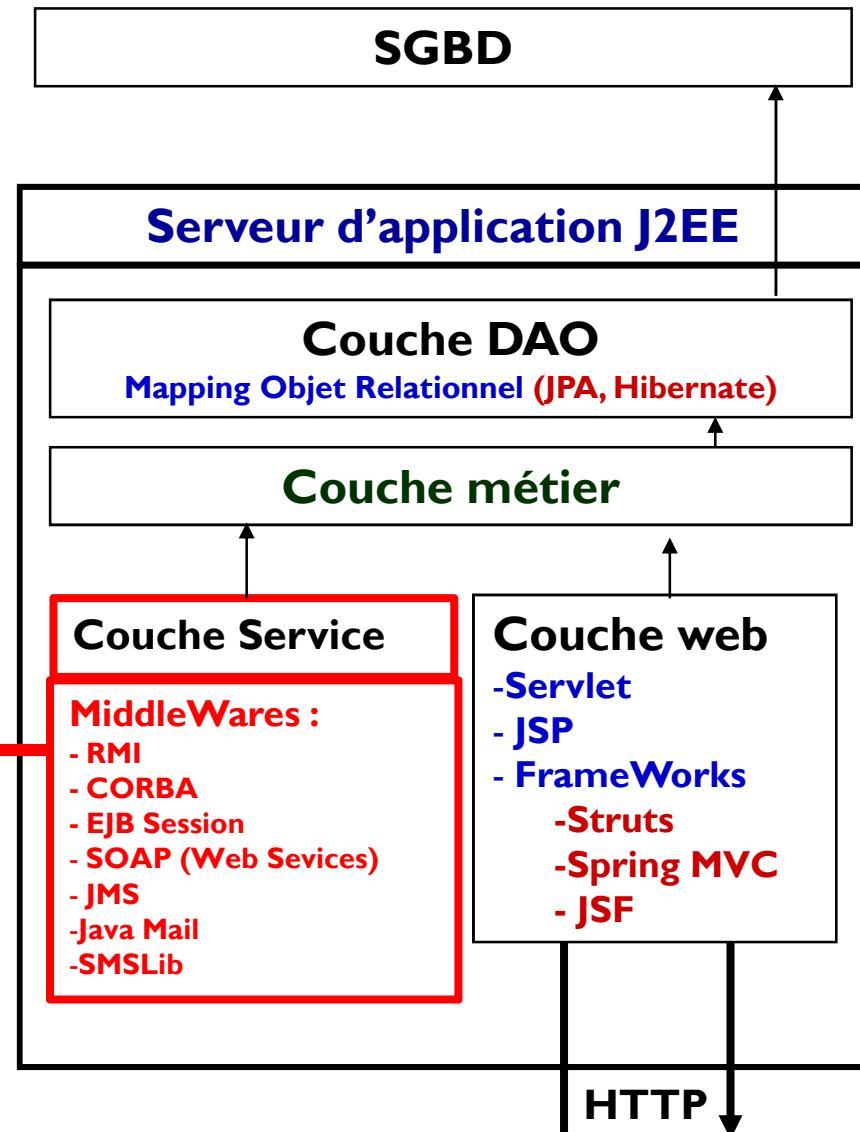
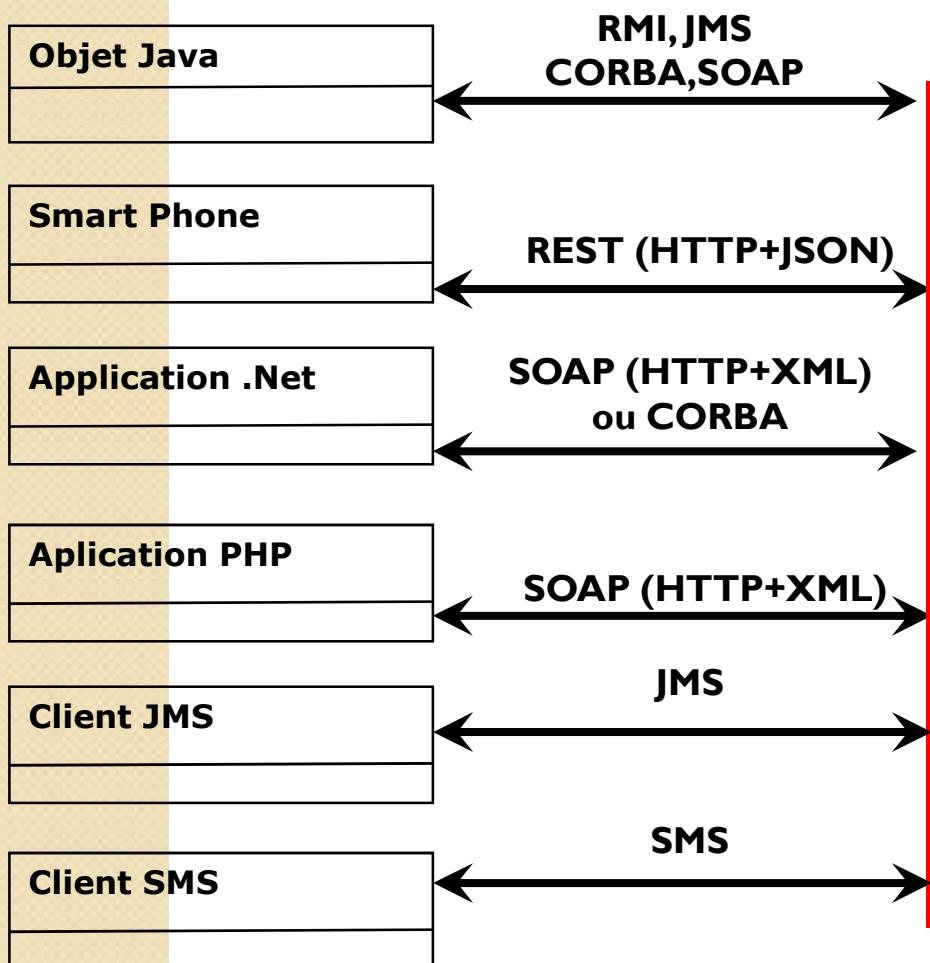
Mohamed Youssfi : med@youssfi.net
ENSET, Université Hassan II Mohammedia

Exigences de qualité d'un système logiciel

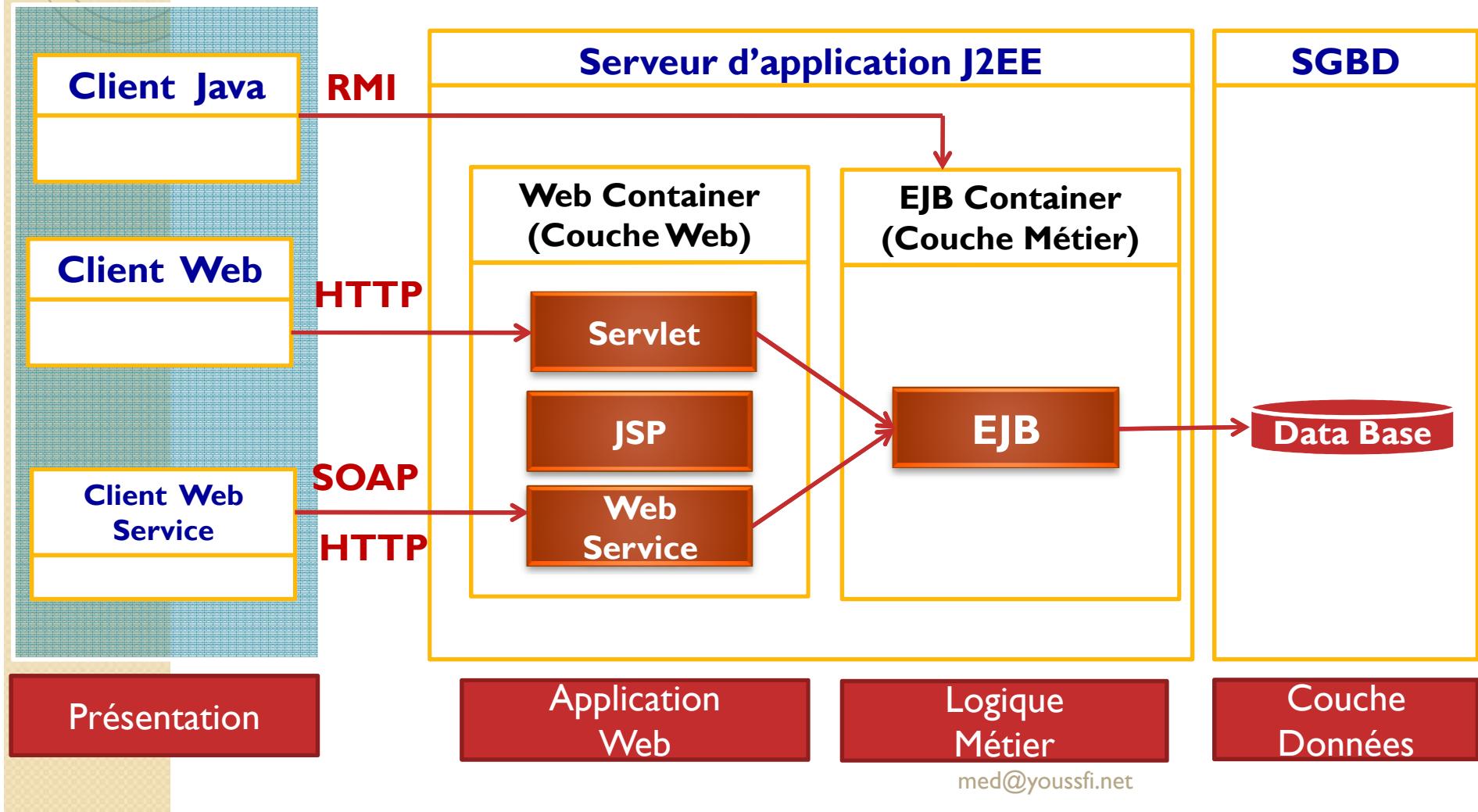
La qualité d'un logiciel se mesure par rapport à plusieurs critères :

- **Exigences fonctionnelles:**
 - Une application est créée pour répondre , tout d'abord, aux besoins fonctionnels des entreprises.
- **Exigences Techniques :**
 - **Les performances:**
 - La rapidité d'exécution et Le temps de réponse
 - Eviter le problème de montée en charge
 - **La maintenance:**
 - Une application doit évoluer dans le temps.
 - Doit être fermée à la modification et ouverte à l'extension
 - Sécurité
 - Portabilité
 - Capacité de communiquer avec d'autres applications distantes.
 - Disponibilité et tolérance aux pannes
 - Capacité de fournir le service à différents type de clients
 - Design des ses interfaces graphiques
 - Coût du logiciel

Architectures Distribués



Couches d'une application



Couche Présentation

- Elle implémente la logique présentation de l'application
- La couche présentation est liée au type de client utilisé :
 - Client Lourd java Desktop:
 - Interfaces graphiques java SWING, AWT, SWT.
 - Ce genre de client peut communiquer directement avec les composants métiers déployés dans le conteneur EJB en utilisant le middleware RMI (Remote Method Invocation)
 - Client Leger Web
 - HTML, Java Script, CSS.
 - Un client web communique avec les composants web Servlet déployés dans le conteneur web du serveur d'application en utilisant le protocole HTTP.
 - Un client .Net, PHP, C++, ...
 - Ce genre de clients développés avec un autre langage de programmation autre que java, communiquent généralement avec les composants Web Services déployés dans le conteneur Web du serveur d'application en utilisant le protocole SOAP (HTTP+XML)
 - Client Mobile
 - Androide, iPhone, Tablette etc..
 - Généralement ce genre de clients communique avec les composants Web Services en utilisant le protocole HTTP ou SOAP



Couche Application

- Appelée également couche web.
- La couche application sert de médiateur entre la couche présentation et la couche métier.
- Elle contrôle l'enchaînement des tâches offertes par l'application
 - Elle reçoit les requêtes http clientes
 - Assure le suivi des sessions
 - Vérifier les autorisations d'accès de chaque session
 - Assure la validation des données envoyées par le client
 - Fait appel au composants métier pour assurer les traitements nécessaires
 - Génère une vue qui sera envoyée à la couche présentation.
- Elle utilise les composants web Servlet et JSP
- Elle respecte le modèle MVC (Modèle Vue Contrôleur)
- Des framework comme JSF, SpringMVC ou Struts sont généralement utilisés dans cette couche.



Couche Métier

- La couche métier est la couche principale de toute application
 - Elle implémente la logique métier d'une entreprise
 - Elle se charge de récupérer, à partir des différentes sources de données, les données nécessaires pour assurer les traitements métiers déclenchés par la couche application.
 - Elle assure la gestion du WorkFlow (Processus de traitement métier en plusieurs étapes)
- Il est cependant important de séparer la partie accès aux données (Couche DAO) de la partie traitement de la logique métier (Couche Métier) pour les raisons suivantes :
 - Ne pas se perdre entre le code métier, qui est parfois complexe, et le code d'accès aux données qui est élémentaire mais conséquent.
 - Ajouter un niveau d'abstraction sur l'accès aux données pour être plus modulable et par conséquent indépendant de la nature des unités de stockage de données.
 - La couche métier est souvent stable. Il est rare qu'on change les processus métier. Alors que la couche DAO n'est pas stable. Il arrive souvent qu'on est contraint de changer de SGBD ou de répartir et distribuer les bases de données.
 - Faciliter la répartition des tâches entre les équipes de développement.
 - Déléguer la couche DAO à frameworks spécialisés dans l'accès aux données (Hibernate, Toplink, etc...)



Entreprise Java Beans (EJB)

- **Enterprise JavaBeans (EJB)** est une architecture de composants logiciels côté serveur pour la plateforme de développement Java EE.
- Cette architecture propose un cadre (Framework) pour créer des composants **distribués** (c'est-à-dire déployés sur des serveurs distants) écrit en langage de programmation Java hébergés au sein d'un serveur applicatif.
- Les EJB permettant de :
 - Représenter des données manipulées par l'application (EJB dit **Entity**),
 - Proposer des services avec ou sans conservation d'état entre les appels (EJB dit **Session**),
 - Accomplir des tâches de manière asynchrone (EJB dit **Message Driven Bean**).



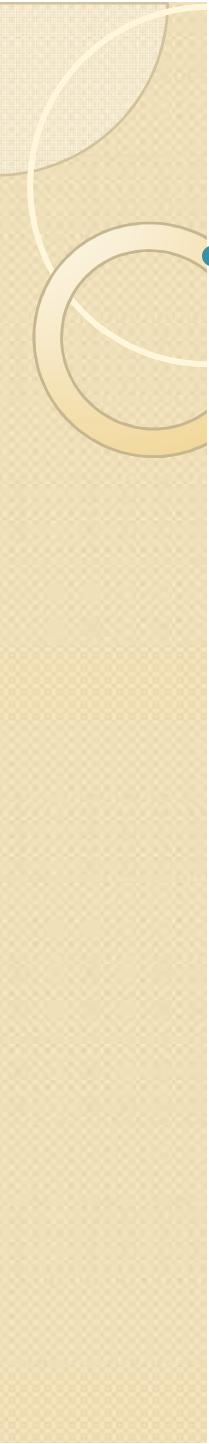
Entreprise Java Beans (EJB)

- Tous les EJB peuvent évoluer dans un contexte transactionnel.
- Les Entreprise Java Bean ou EJB sont des composants qui permettent de d'implémenter la logique **métier**.
- Ce sont des composant **distribués** qui s'exécutent au sein d'un conteneur EJB qui fait partie du serveur d'application J2EE
- Le but des EJB est de faciliter la création d'applications distribuées pour les entreprises.



Entreprise Java Beans (EJB)

- Une des principales caractéristiques des EJB est de permettre aux développeurs de se concentrer sur les traitements orientés métiers car les EJB et l'environnement dans lequel ils s'exécutent prennent en charge un certain nombre de traitements techniques en utilisant les services de l'infrastructure offert par le serveur d'application tel que :
 - La distribution
 - La gestion des transactions,
 - La persistance des données,
 - Le cycle de vie des objets
 - La montée en charge
 - La concurrence
 - La sécurité
 - La sérialisation
 - Journalisation
 - Etc....
- Autrement dit, EJB permet de **séparer** le code **métier** qui est propre aux spécifications fonctionnelles du code **technique** (spécification non fonctionnel)

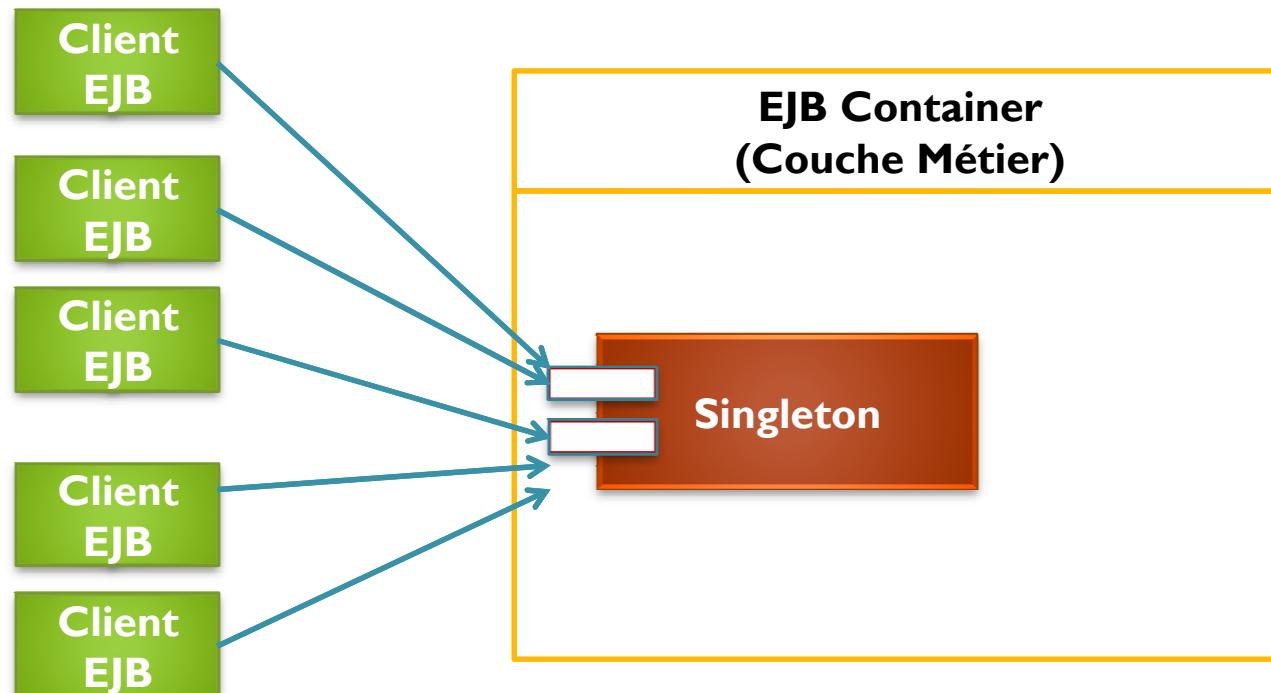


Différents types d'EJB

- Il existe trois types d'EJB :
 - **Entity Beans** : Les EJB entités
 - Représentent les données manipulées par l'application (Composants persistants)
 - Chaque EJB Entity est associé à une table au niveau de la base de données
 - **Session Beans** : Les EJB session
 - Composants distribués qui implémentent les traitement de la logique métier.
 - Ces composants sont accessibles à distance via les protocole RMI et IIOP.
 - Il existe deux types d'EJB Session.
 - **Stateless** : sans état
 - Une instance est créée par le serveur pour plusieurs connexions clientes.
 - Ce type de bean ne conserve aucune donnée dans son état.
 - **Statefull** : avec état
 - Création d'une instance pour chaque connexion cliente.
 - Ce type de bean peut conserver des données entre les échanges avec le client.
 - **Singleton**: Instance Unique
 - Création d'une instance unique quelque soit le nombre de connexion.
 - **Message Driven Beans** : Beans de messages
 - Un listener qui permet de déclencher des traitements au niveau de l'application suite à la réception d'un message asynchrone JMS

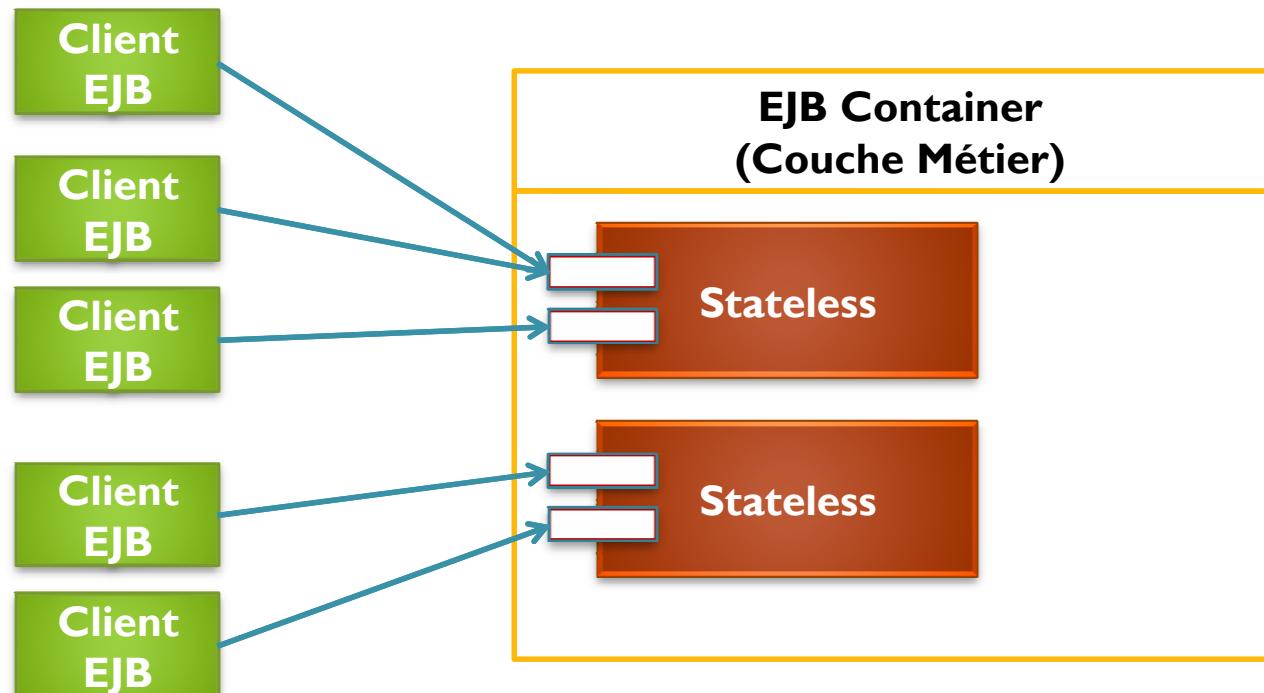
EJB Session Singleton

- Crédation d'une seule instance



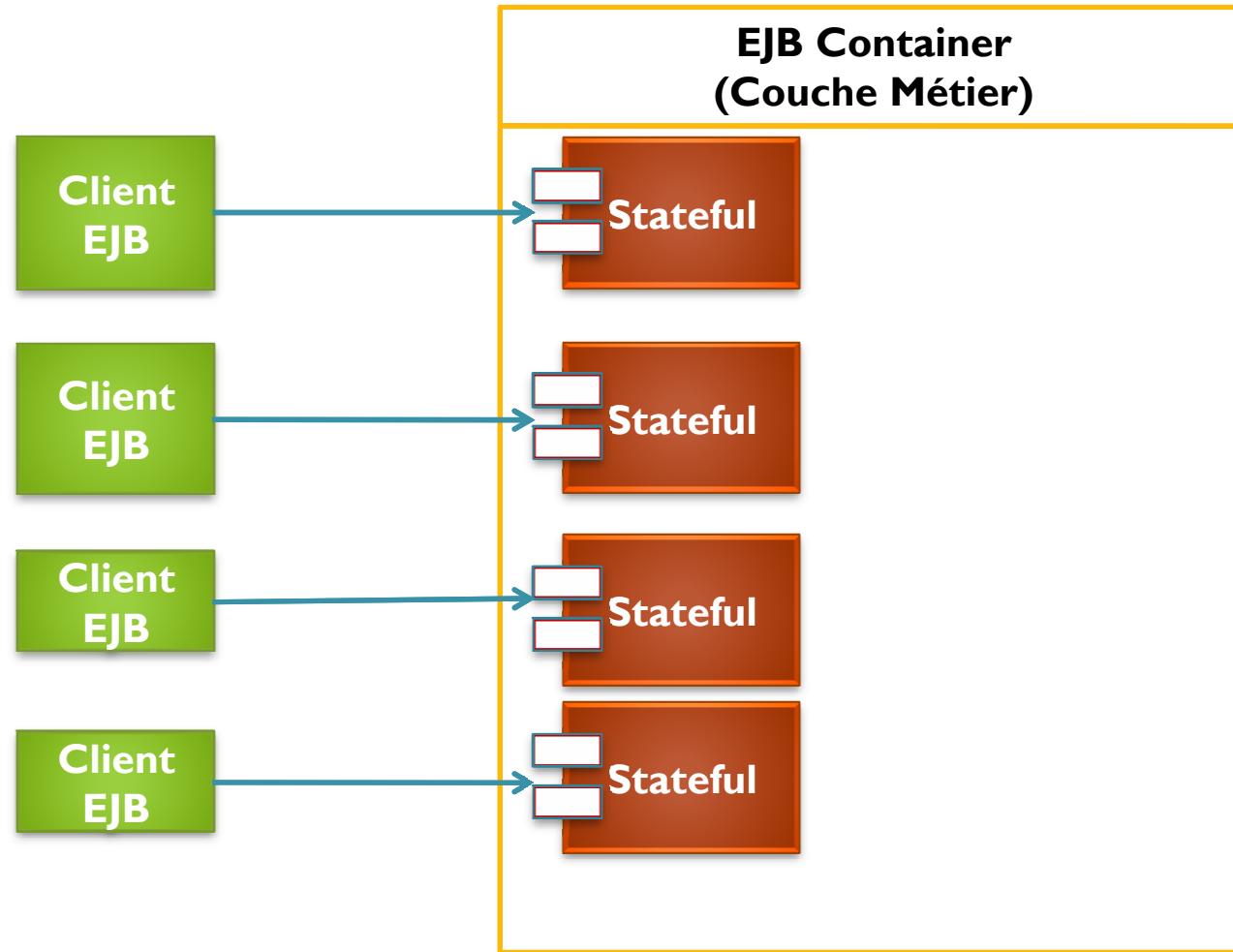
EJB Session Stateless

- Crédation d'un pool d'instances

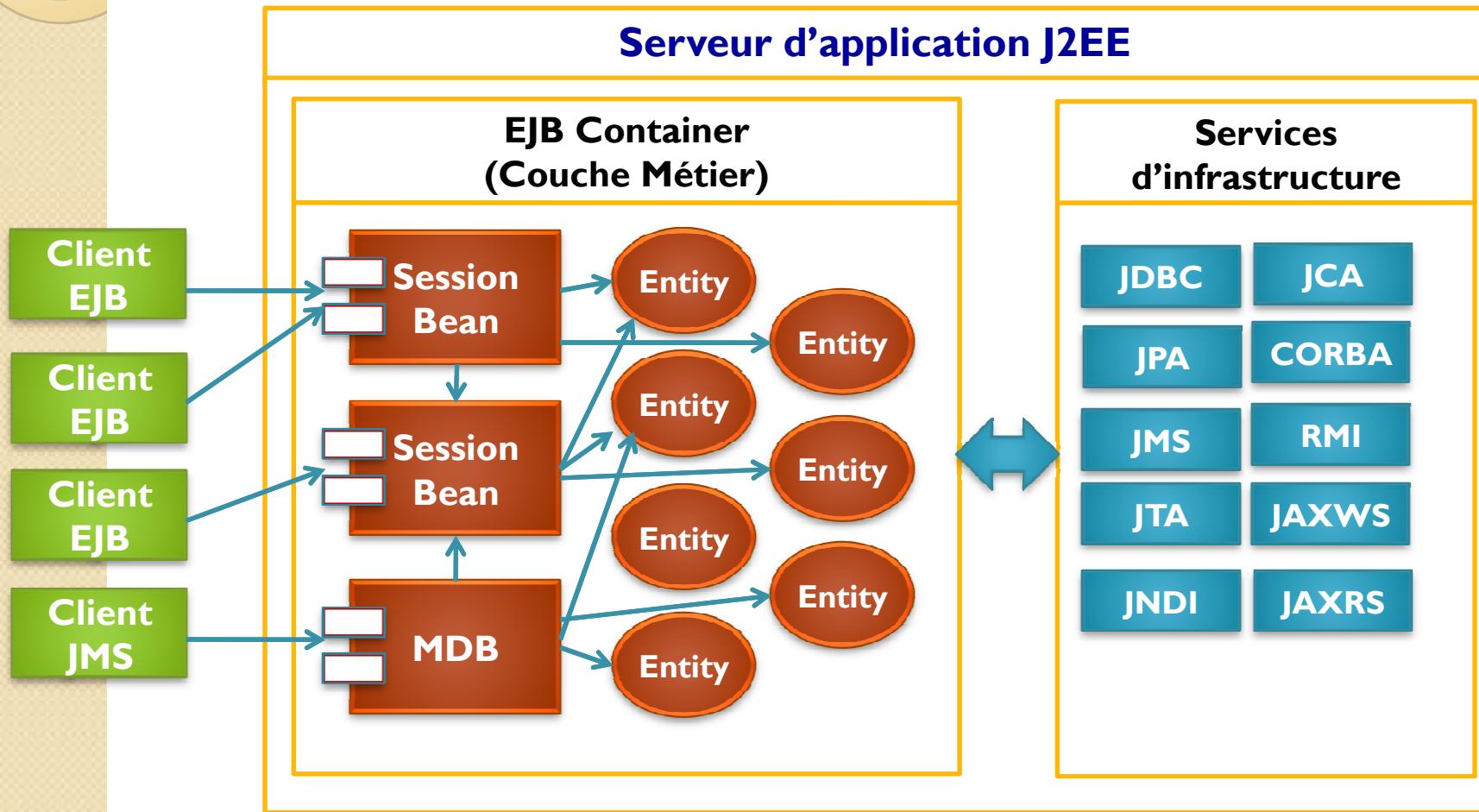


EJB Session Stateful

- Création d'une instance pour chaque connexion



Conteneur EJB et Services d'infrastructures





Conteneur EJB

- Le rôle du conteneur EJB et de :
 - Gérer le cycle de vie des composants EJB de votre application
 - Instanciation
 - Accès au bean
 - Sérialisation
 - Désérialisation
 - Destruction du bean
 - Permettre aux EJB d'accéder aux services d'infrastructures offerts par le serveur d'application JEE.

Services d'infrastructure d'un serveur d'application J2EE

- **JDBC** (Java Data Base Connectivity) : Pilotes java pour permettre l'accès aux bases de données
- **JPA** (Java Persistence API). Interface qui permet de gérer la persistance des EJB Entity. Le serveur d'application fournit un Framework implémentant la spécification JPA pour assurer le mapping objet relationnel (Pour JBOSS c'est Hibernate qui est utilisé)
- **JNDI (Java Naming Directory Interface)** : Service permettant de connecter les composants de l'application vers les services de noms ou d'annuaires comme les annuaires RMI, CORBA, LDAP, DNS etc... chaque serveur d'application possède son propre annuaire qui lui permet de publier les références des composants déployés dans le serveur en les associant à un nom. Comme ça ils seront accessibles en local ou à distance par d'autres composants.
- **JMS** : Service de messagerie asynchrone. Ce service permet aux composants comme les MDB de se connecter à une file d'attente du service de messagerie inter-application JMS pour envoyer et recevoir d'une manière asynchrone les messages JMS à destination et en provenance de d'autres composants distants. Chaque serveur d'application possède son propre provider JMS qui permet de déployer de nouvelles files d'attente JMS.
- **JMX** : service qui permet de modifier les paramètres de configuration du serveur d'application à chaud



Services d'infrastructure d'un serveur d'application J2EE

- **JCA (Java Connector Architecture)**: pour connecter les composants d'une application J2EE vers d'autres systèmes d'informations d'entreprises comme les ERP.
- **JAXWS** : Spécification pour l'implémentation des web services SOAP. Chaque serveur d'application possède sa propre implémentation JaxWS (CXF, AXIS,...)
- **JAXRS** : Spécification pour l'implémentation des web services REST FULL. Chaque serveur d'application possède sa propre implémentation JaxRS (Jersey, RestEasy,...)
- **JSTL** (Java Server Pages Standard Tag Library) : Bibliothèques de tag JSP qui permettent de faciliter l'écriture des pages web dynamique JSP.
- **JSF** (Java Server faces) : Framework MVC pour développer les application web J2EE au même titre que Struts et Spring MVC.
-



EJB Session

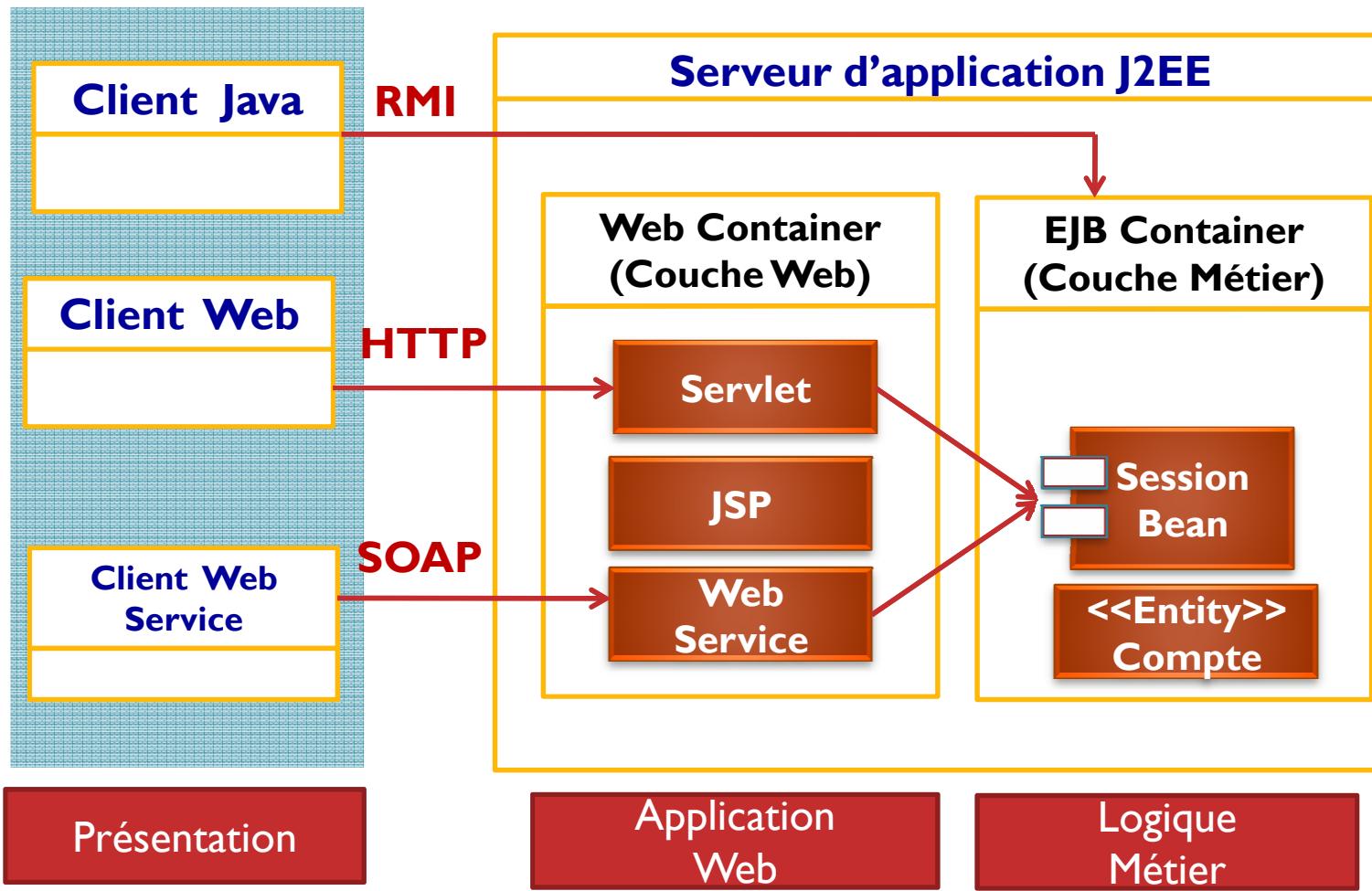
- Un EJB Session est un composant qui possède
 - Une interface **remote** : qui permet de déclarer les méthodes qui sont accessibles à distance. C'est-à-dire accessible aux composants qui sont déployés dans d'autres machines.
 - Une interface **Local** : qui permet de déclarer les méthodes qui sont accessible en local. C'est-à-dire les méthodes accessible par les composants déployés dans le même serveur d'application.
 - La **classe du bean** qui implémente les deux interfaces remote et local. l'implémentation des méthodes de cette classe représentent les traitements métier de l'application



Exemple de problème

- On souhaite créer une application qui permet de gérer des comptes.
- Chaque compte est défini par son code, son solde et sa date de création
- L'application doit permettre de réaliser les opérations suivantes:
 - Ajouter un compte
 - Consulter tous les comptes
 - Consulter un compte
 - Effectuer le versement d'un montant dans un compte
 - Effectuer le retrait d'un montant d'un compte
- L'application doit être accessible par :
 - Un client lourd java distant
 - Une application web basée sur Servlet et JSP
 - Un client SOAP
- On supposera, dans un premier temps, que les comptes sont stockés dans une liste non persistante.
- Par la suite nous aborderons la persistance des comptes dans une base de données en utilisant JPA.

Architecture

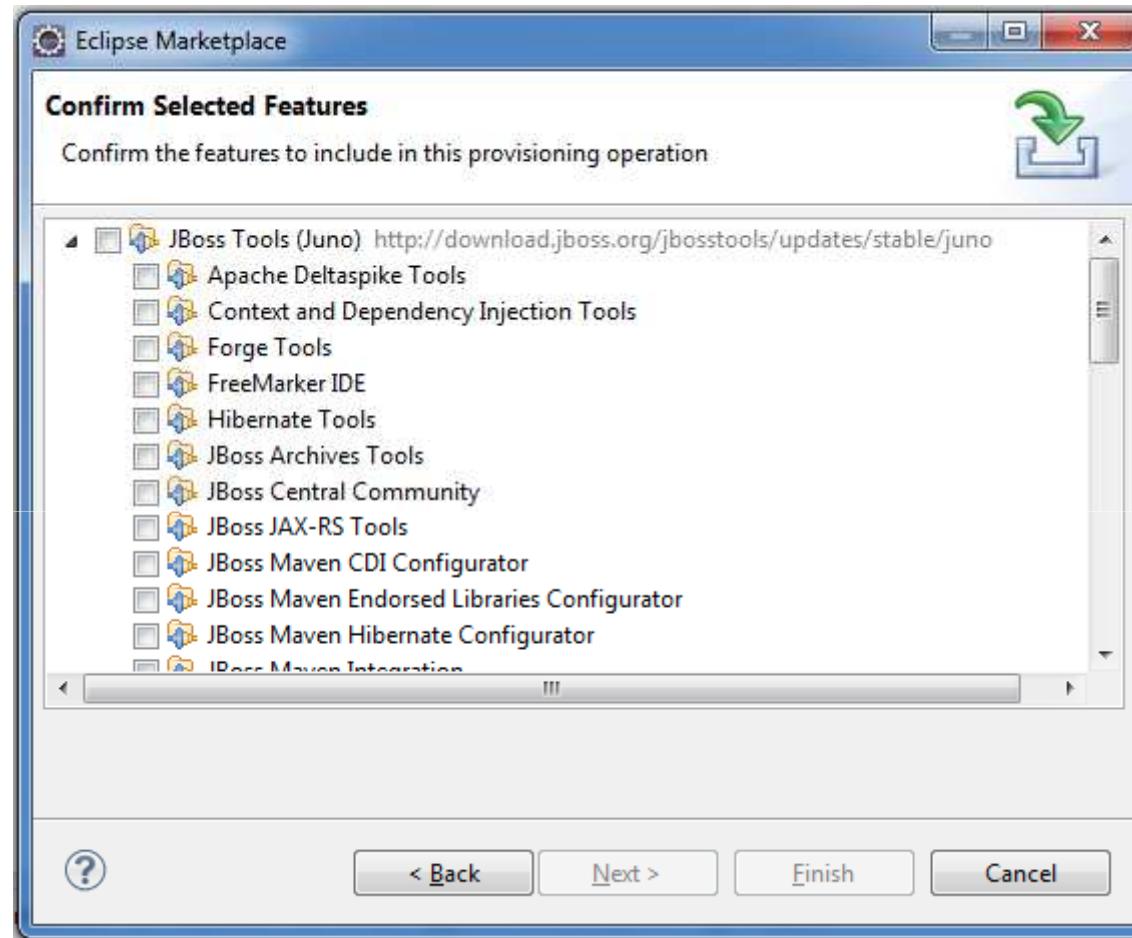


Installation du plugin Jboss Tools pour Eclipse



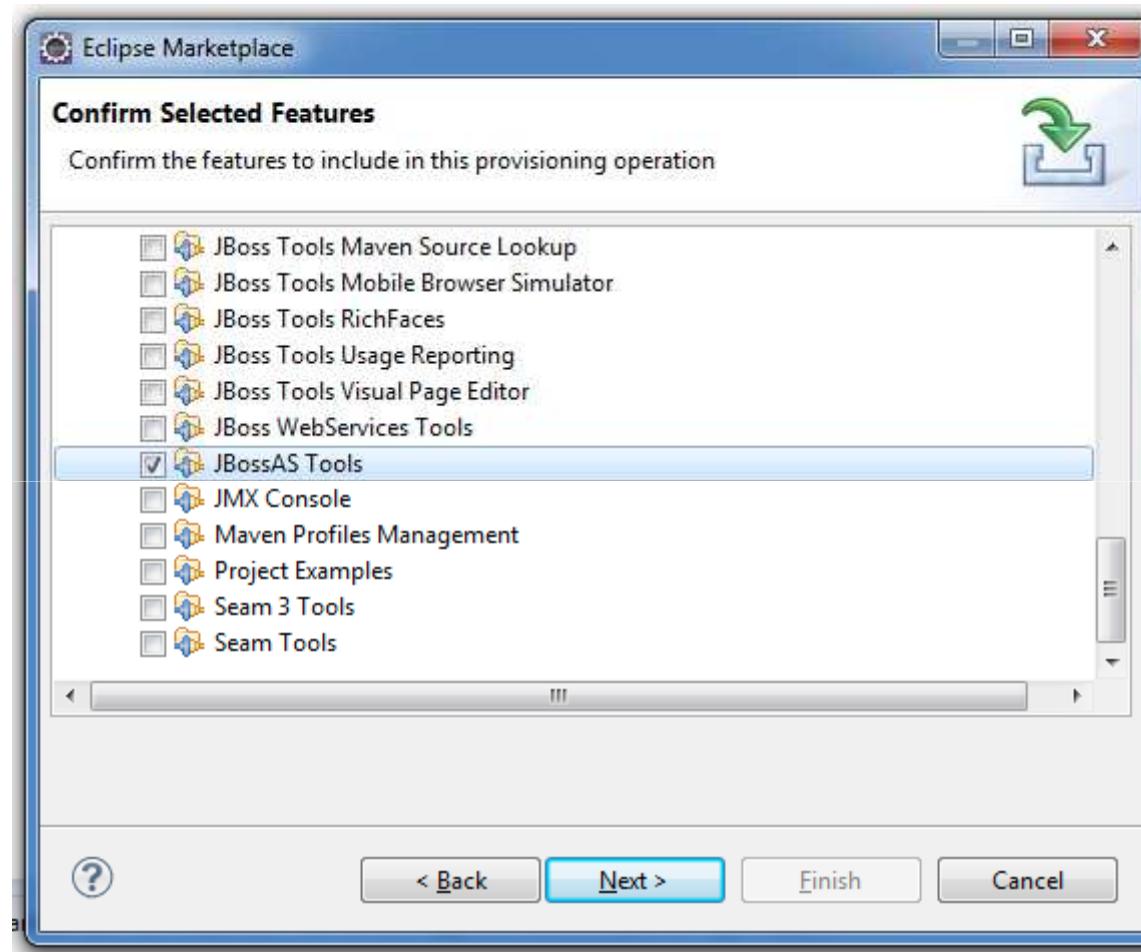
- Help>Eclipse Marketplace

Installation du plugin Jboss Tools pour Eclipse



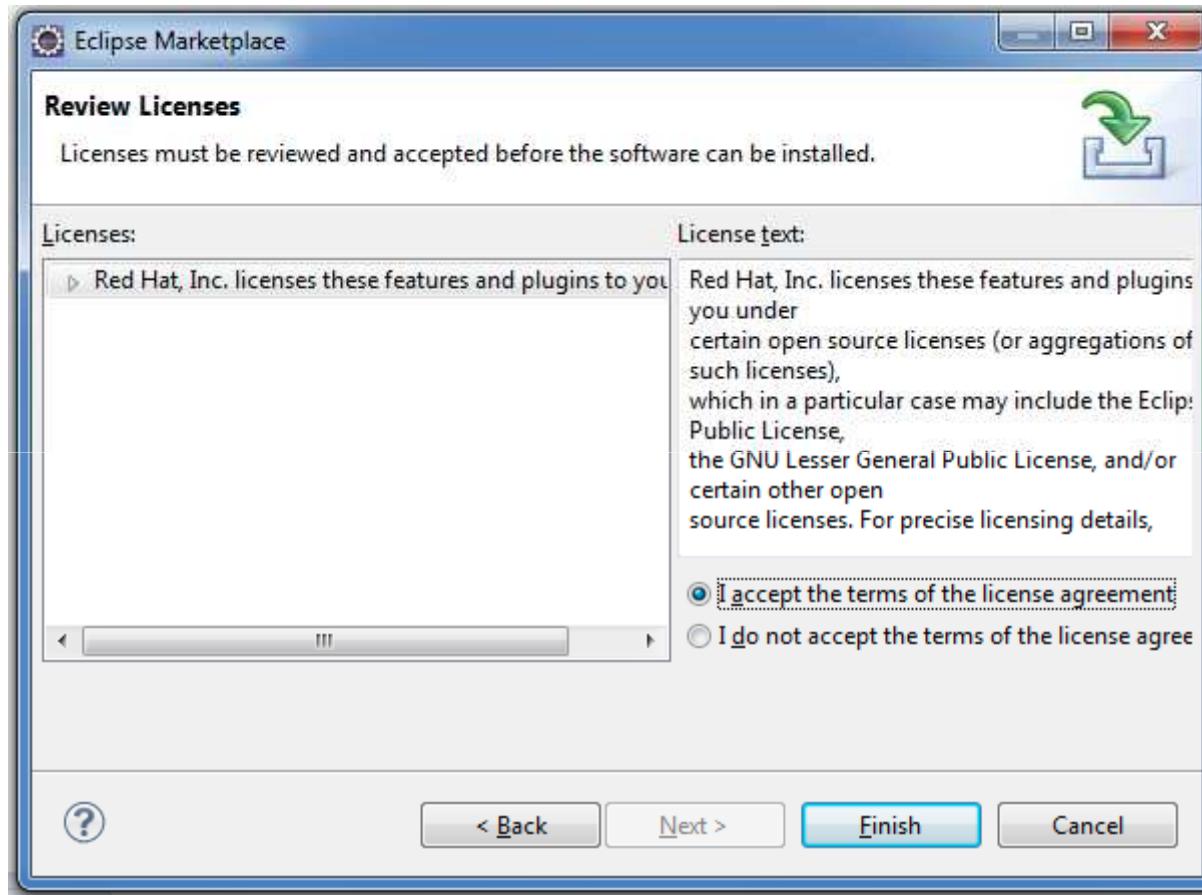
- Il n'est pas nécessaire d'installer tous les outils fournis par Jboss Tools
- Désactiver le plugin pour ne sélectionner que les plugins dont on a besoin

Installation du plugin Jboss Tools pour Eclipse



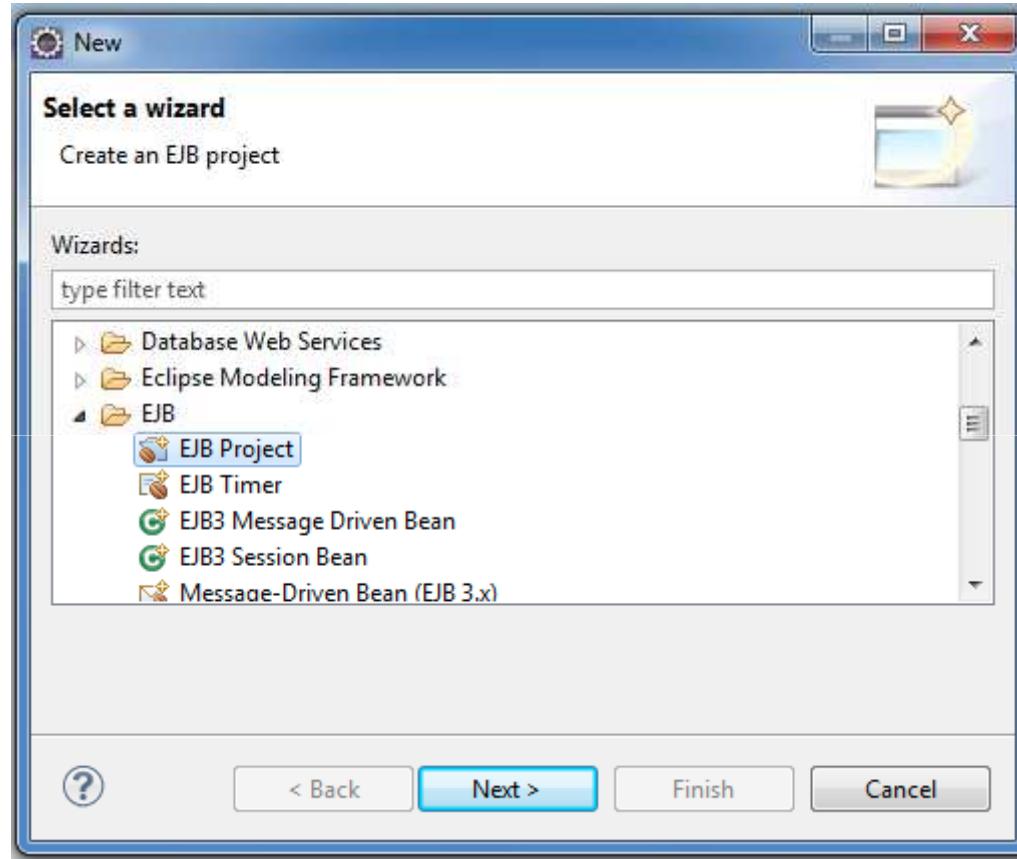
- Activer Uniquement JBossAS Tools
- Pour installer les outils du serveur d'application Jboss

Installation du plugin Jboss Tools pour Eclipse



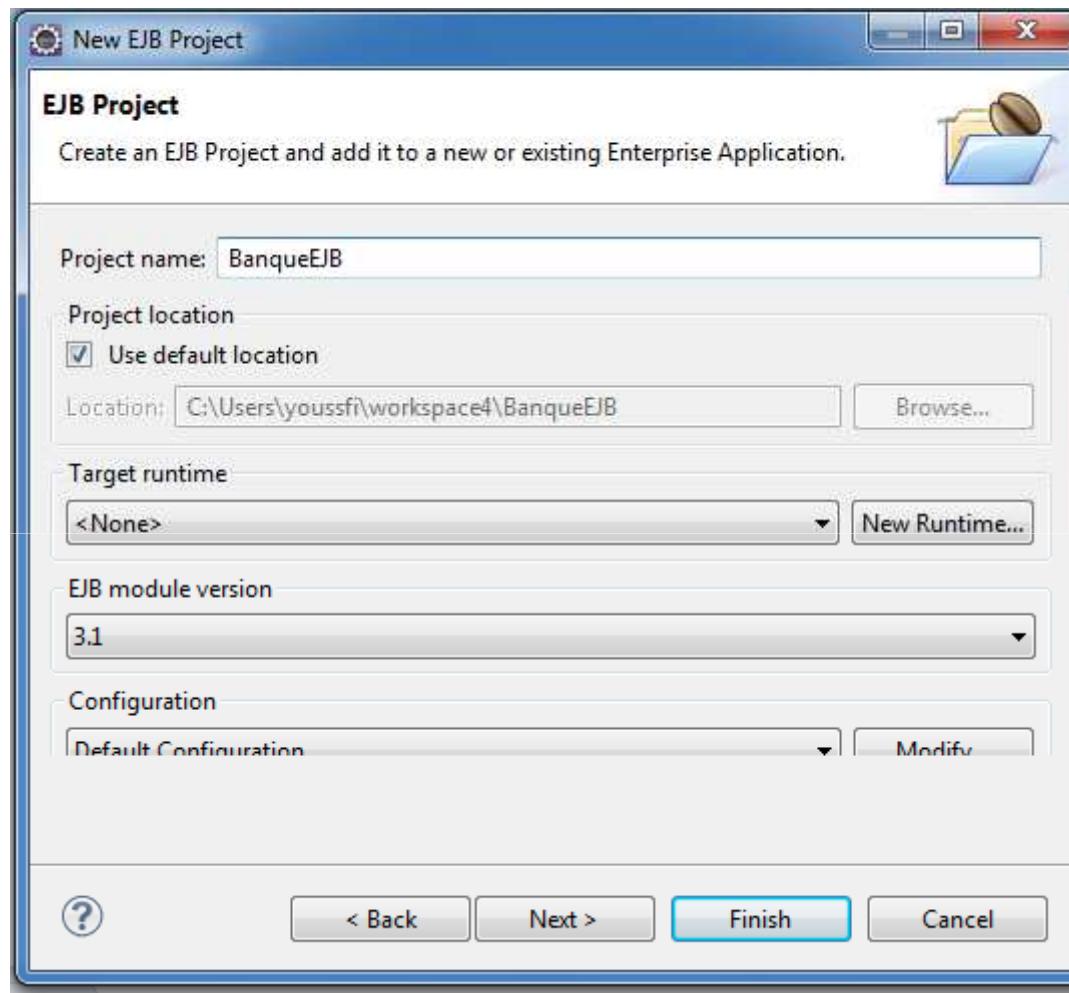
- Acceptez la licence
- Une fois l'installation terminée, L'assistanat vous demandera de redémarrer Eclipse

Création d'un projet EJB



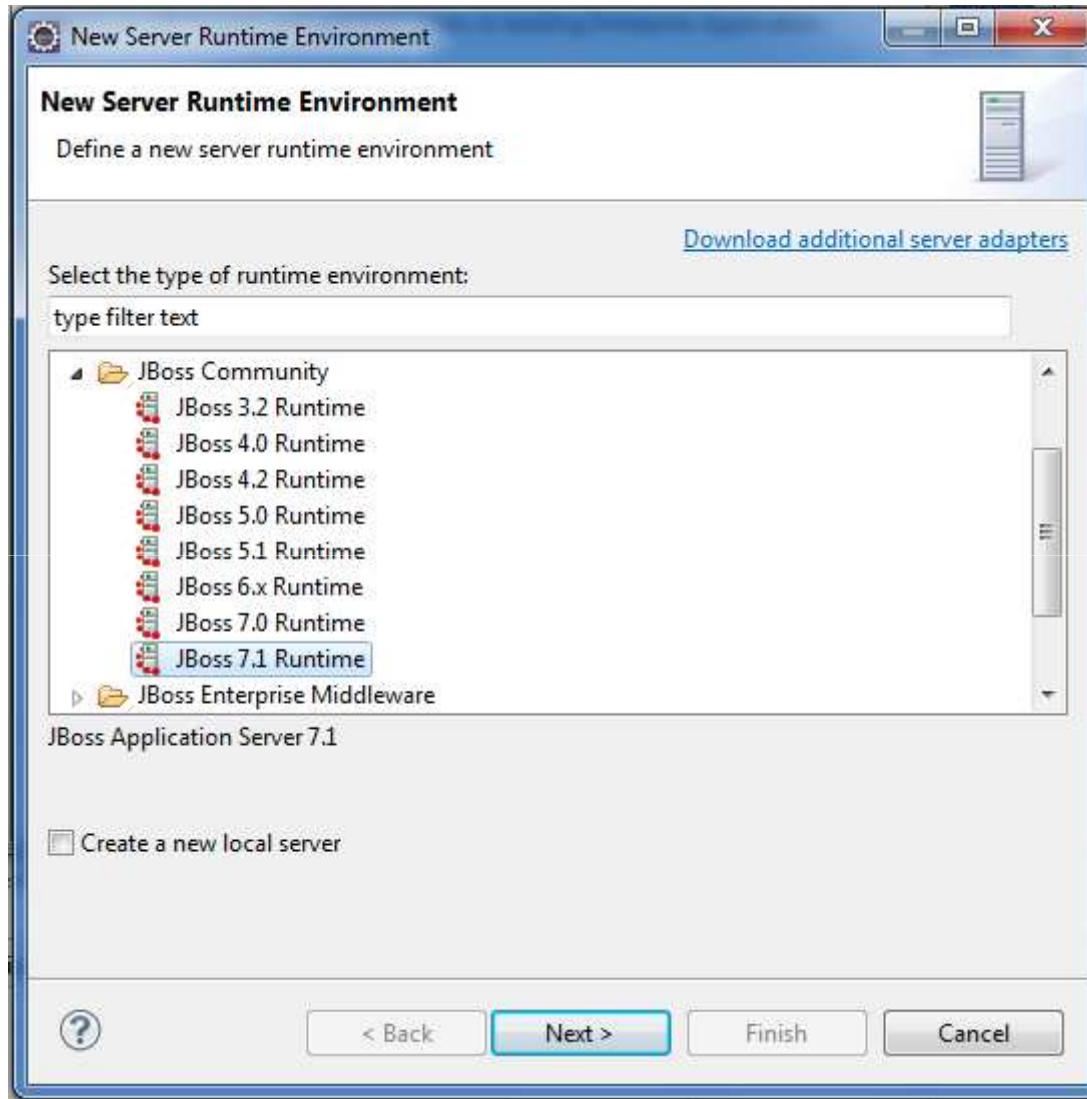
- New > EJB Project

Création d'un projet EJB



- Après avoir spécifié le nom du projet,
- Cliquez sur le bouton New Runtime, pour associer un serveur à ce projet

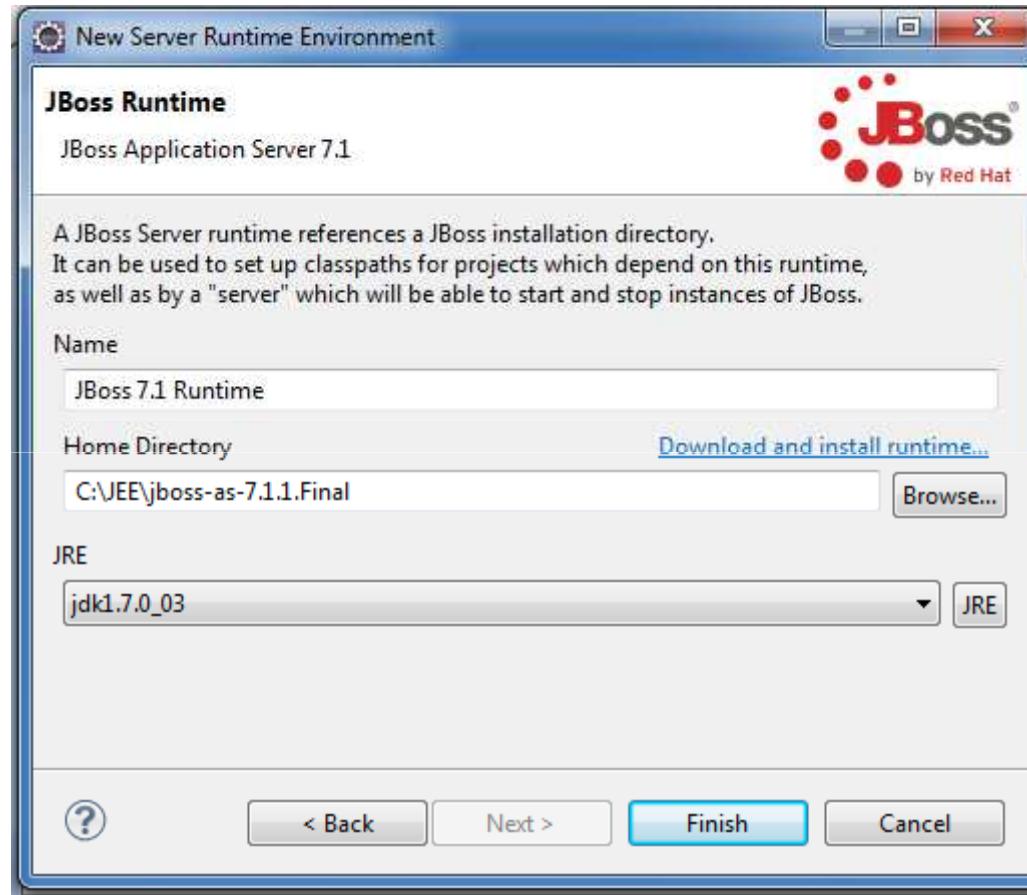
Associer un serveur au projet



- Dans notre cas, nous utilisons Jboss 7.1
- A télécharger sur le site de Jboss : <http://www.jboss.org/jbosas/downloads/>

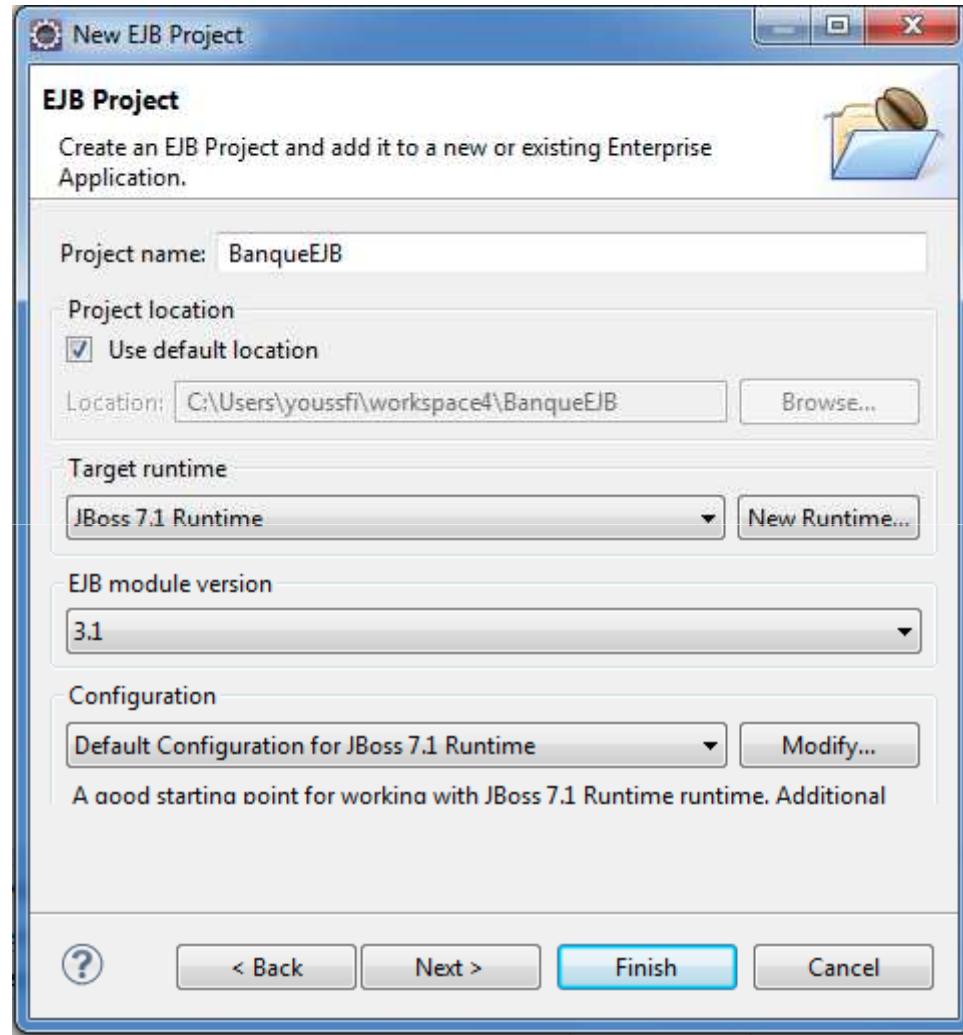
med@youssfi.net

Associer un serveur au projet



- Spécifier le chemin du serveur Jboss
- Sélectionner un JRE installé (Pour Jboss 7, il préférable d'utiliser JDK1.7)

Associer un serveur au projet

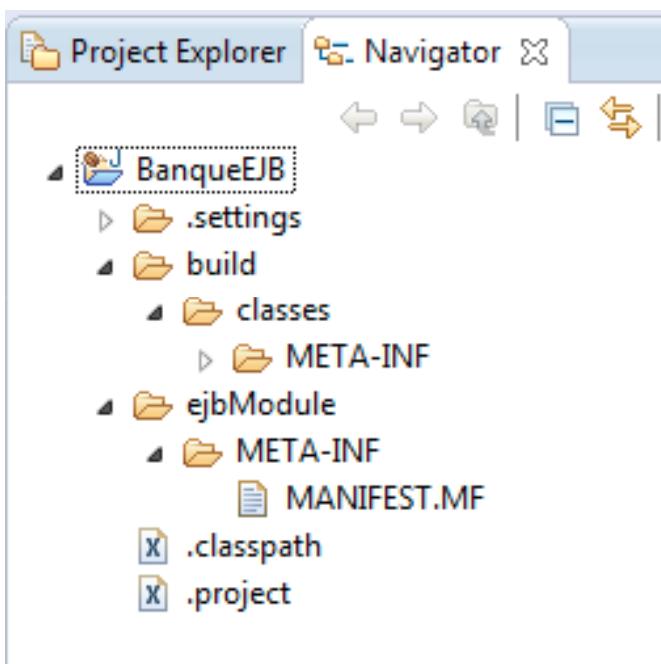


- Finir la création du projet EJB

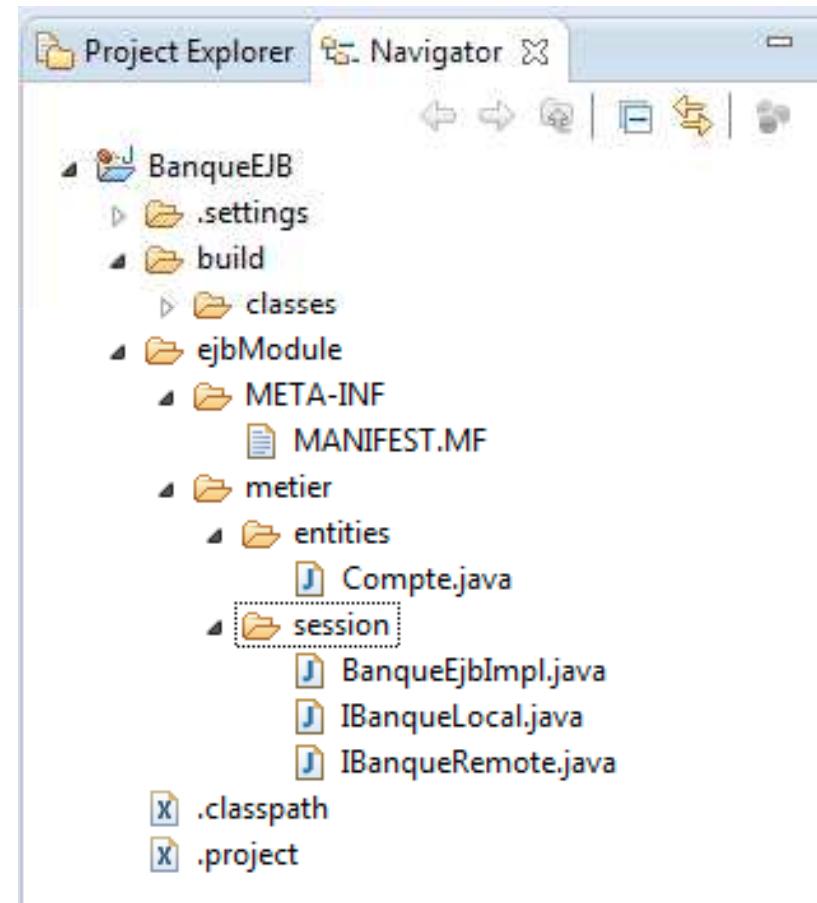
med@youssfi.net

Structure du Projet EJB

- Projet Vide



- Projet à créer



Les données manipulées (Entités) : Compte.java

- On commence par créer les structures de données manipulées par l'application.
- Il s'agit des entités (Entities) de l'application. Dans notre cas, notre application traite des comptes. Pour le moment nous n'allons pas traiter la persistance des entités
- Nous considérons que Compte est un simple Java Bean et non pas un EJB Entity

```
package metier.entities;
import java.io.Serializable;
import java.util.Date;
public class Compte implements Serializable {
    private Long code;
    private double solde;
    private Date dateCreation;

    public Compte(Long code, double solde, Date dateCreation) {
        this.code = code;  this.solde = solde;  this.dateCreation = dateCreation;
    }
    public Compte() { }
    @Override
    public String toString() {
        return "Compte [code=" + code + ", solde=" + solde + ", dateCreation=" +
dateCreation + "]";
    }
    // Getters et Setters
}
```

Interface Remote

- Une interface Remote d'un EJB Session doit être annotée **@Remote**

```
package metier.session;
import java.util.List;
import javax.ejb.Remote;
import metier.entities.Compte;
@Remote
public interface IBanqueRemote {
    public void addCompte(Compte c);
    public List<Compte> consulterComptes();
    public Compte consulterCompte(Long code);
    public void verser(Long code,double montant);
    public void retirer(Long code,double montant);
}
```

L'interface Local

- Une interface Locale d'un EJB Session doit être annotée **@Local**
- Dans notre cas, nous supposons que toutes les méthodes de l'EJB Session sont accessible à distance et en local

```
package metier.session;
import java.util.List;
import javax.ejb.Local;
import metier.entities.Compte;
@Local
public interface IBanqueLocal {
    public void addCompte(Compte c);
    public List<Compte> consulterComptes();
    public Compte consulterCompte(Long code);
    public void verser(Long code,double montant);
    public void retirer(Long code,double montant);
}
```

Implémentation d'un EJB Session Stateless

- Un EJB Session est une classe qui implémente ses deux interfaces Local et Remote.
- Cette classe doit être annoté par l'une des annotations suivantes:
 - **@Statless** : Un pool d'instances de cet EJB sera créé par le serveur
 - **@Statfull** : Pour chaque connexion, le serveur crée une instance
 - **@Singleton** : Un instance unique sera créée quelque soit le nombre de connexions
- Après instantiation d'un EJB Session, ses références Remote (IP, Port, Adresse Mémoire) et Locale (Adresse Mémoire) seront publiée dans l'annuaire JNDI.
- L'attribut **name** de ces trois annotations, permet de spécifier le nom JNDI qui sera associé aux références de l'EJB dans l'annuaire JNDI
- Par défaut, c'est le nom de la classe qui sera utilisé.
- Ce nom sera combiné à d'autres informations techniques pour garantir l'unicité de ce nom.
- Avec Jboss 7, le nom complet JNDI d'un EJB Session est de la forme suivante :
 - Pour un statless et singleton
 - `Nom_Projet_EAR/Nom_Projet_EJB/Name!Package.NomInterface`
 - Pour un statful
 - `Nom_Projet_EAR/Nom_Projet_EJB/Name!Package.NomInterface?statful`

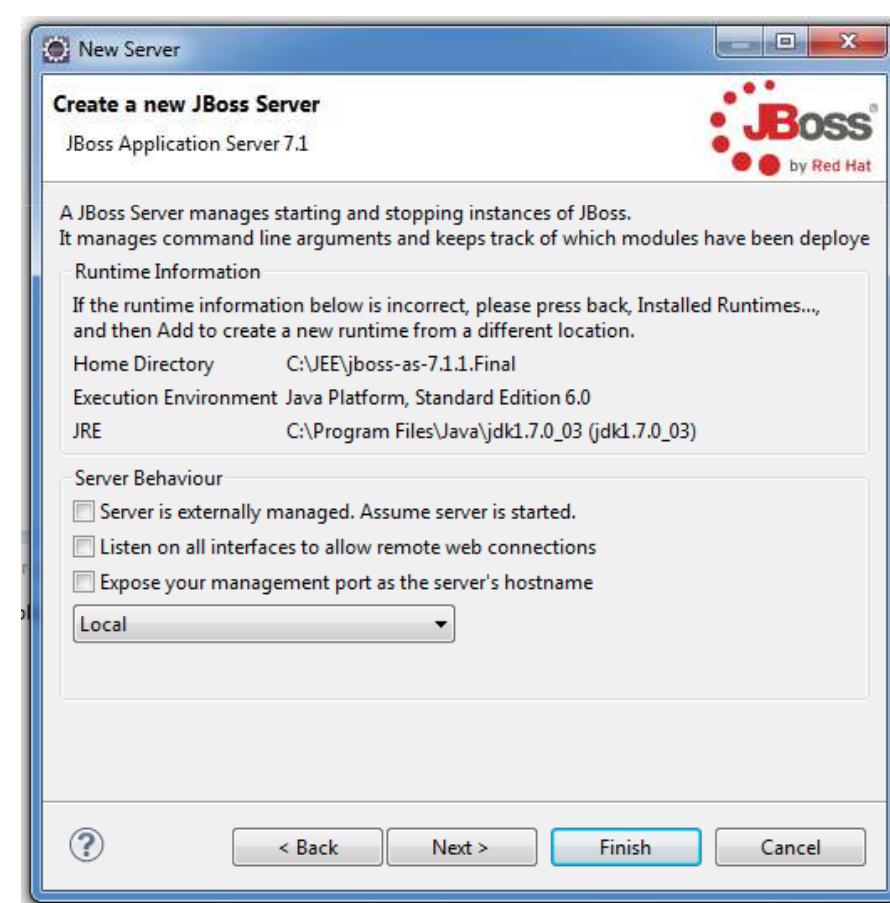
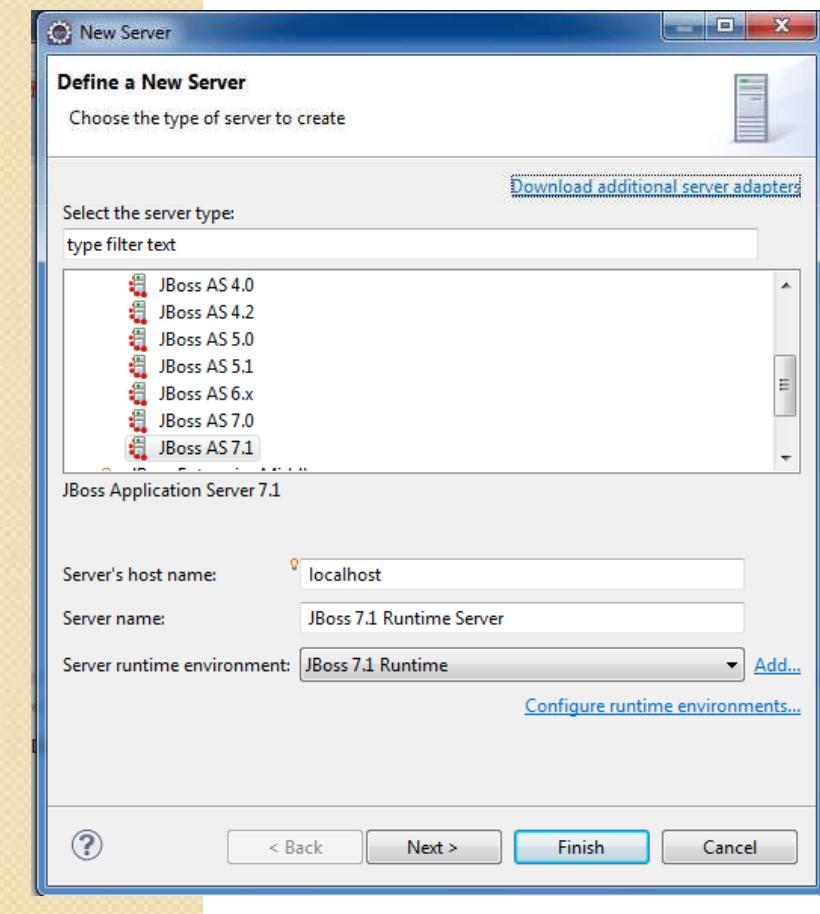
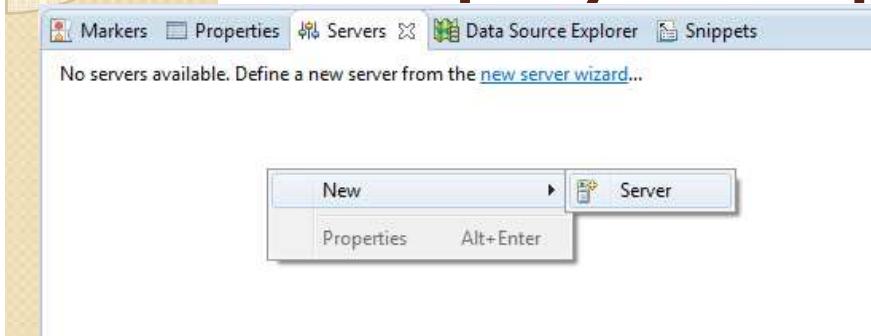
Implémentation de L'EJB Session

```
package metier.session;
import java.util.*; import javax.annotation.PostConstruct;
import javax.ejb.Singleton;import metier.entities.Compte;
@Singleton(name="BP2")
public class BanqueEjbImpl implements IBanqueLocal,IBanqueRemote {
    private Map<Long, Compte> comptes=new Hashtable<Long, Compte>();
    @Override
    public void addCompte(Compte c) {
        comptes.put(c.getCode(), c);
    }
    @Override
    public List<Compte> consulterComptes() {
        List<Compte> cptes=new ArrayList<Compte>(comptes.values());
        return cptes;
    }
    @Override
    public Compte consulterCompte(Long code) {
        Compte cp=comptes.get(code);
        if(cp==null) throw new RuntimeException("Compte introuvable");
        return cp;
    }
}
```

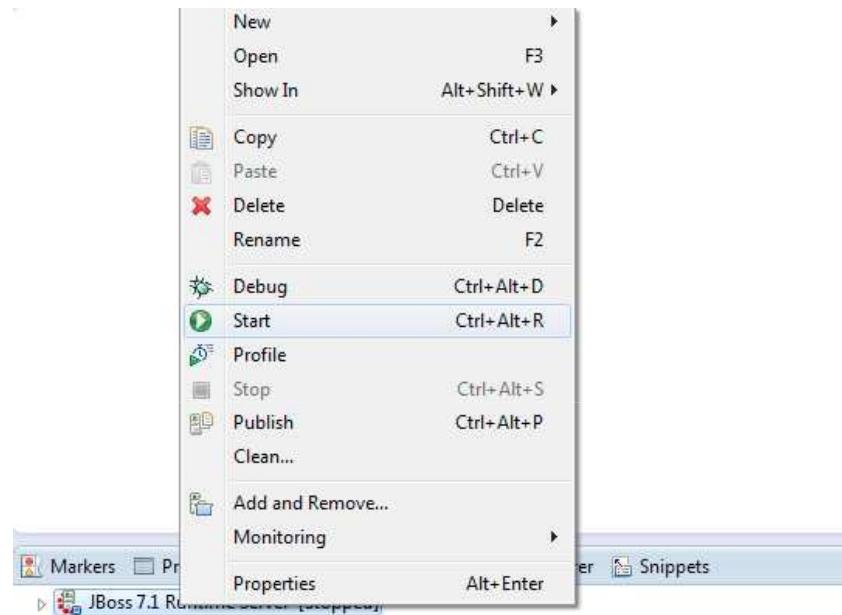
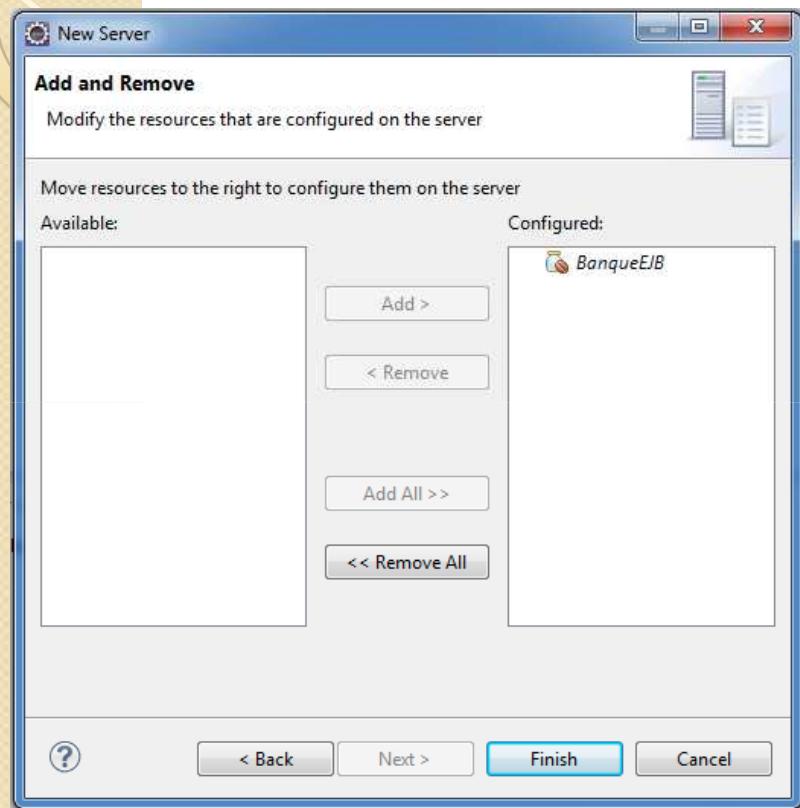
Implémentation d'un EJB Session Stateless (Suite)

```
@Override  
public void verser(Long code, double montant) {  
    Compte cp=comptes.get(code);  
    cp.setSolde(cp.getSolde()+montant);  
}  
  
@Override  
public void retirer(Long code, double montant) {  
    Compte cp=comptes.get(code);  
    if(cp.getSolde()<montant) throw new RuntimeException("Solde insuffisant");  
    cp.setSolde(cp.getSolde()-montant);  
}  
  
@PostConstruct  
public void initialisation(){  
    addCompte(new Compte(1L, 7000, new Date()));  
    addCompte(new Compte(2L, 4000, new Date()));  
    addCompte(new Compte(3L, 2000, new Date()));  
}  
}
```

Déployer le projet EJB



Déployer le projet EJB



Aperçu de la console du serveur Jboss après démarrage

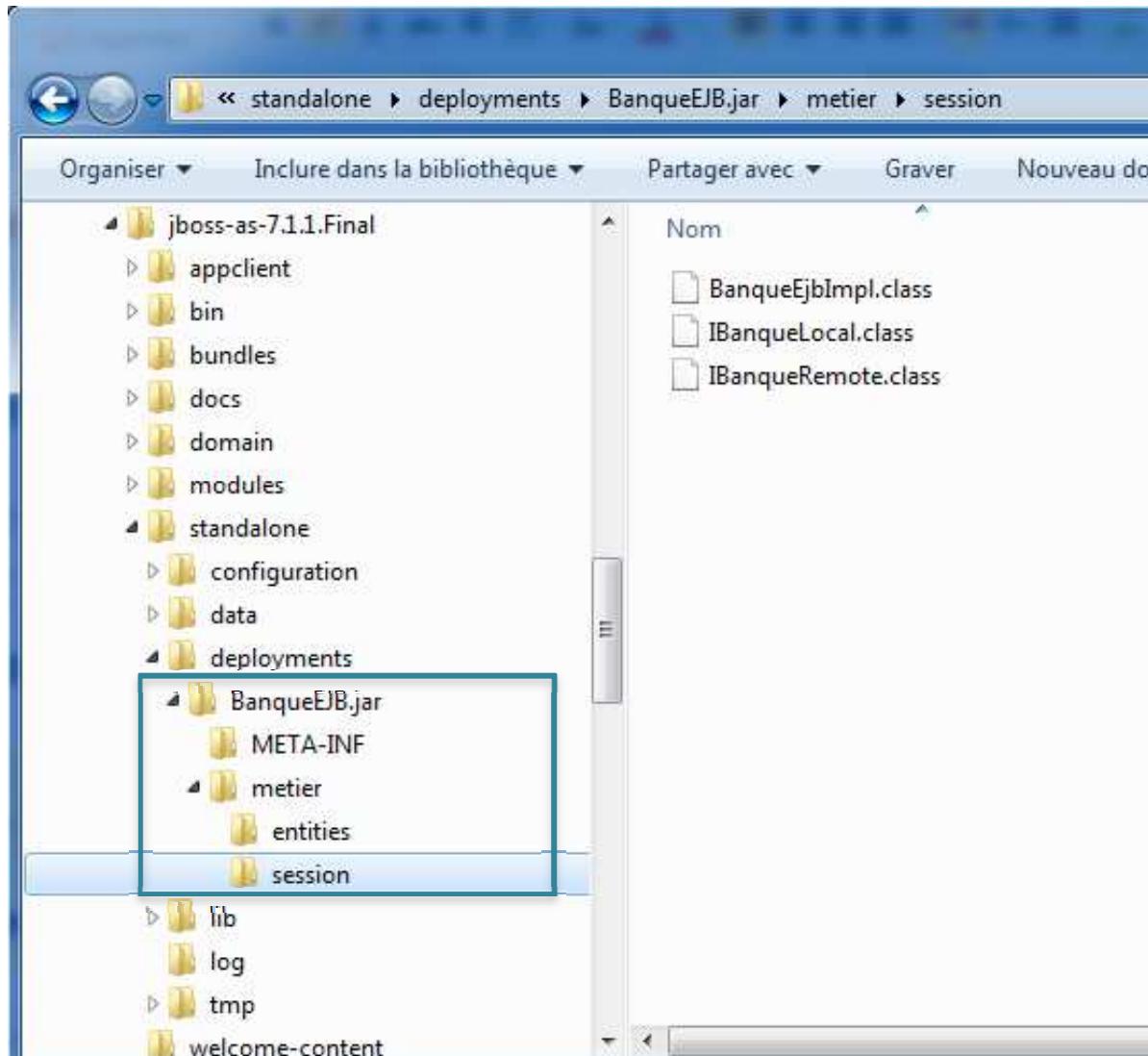
```
.....  
08:26:10,612 INFO [org.jboss.as.server.deployment] (MSC service thread 1-6) JBAS015876: Starting deployment of  
"BanqueEJB.jar"  
08:26:10,858 INFO [org.jboss.as.ejb3.deployment.processors.EjbJndiBindingsDeploymentUnitProcessor] (MSC service  
thread 1-6) JNDI bindings for session bean named BP2 in deployment unit deployment "BanqueEJB.jar" are as  
follows:
```

```
java:global/BanqueEJB/BP2!metier.session.IBanqueRemote  
java:app/BanqueEJB/BP2!metier.session.IBanqueRemote  
java:module/BP2!metier.session.IBanqueRemote  
java:jboss/exported/BanqueEJB/BP2!metier.session.IBanqueRemote  
java:global/BanqueEJB/BP2!metier.session.IBanqueLocal  
java:app/BanqueEJB/BP2!metier.session.IBanqueLocal  
java:module/BP2!metier.session.IBanqueLocal
```

Les références
Remote et Local
De l'EJB Session
Publiées dans
L'annuaire JNDI
Avec ces noms

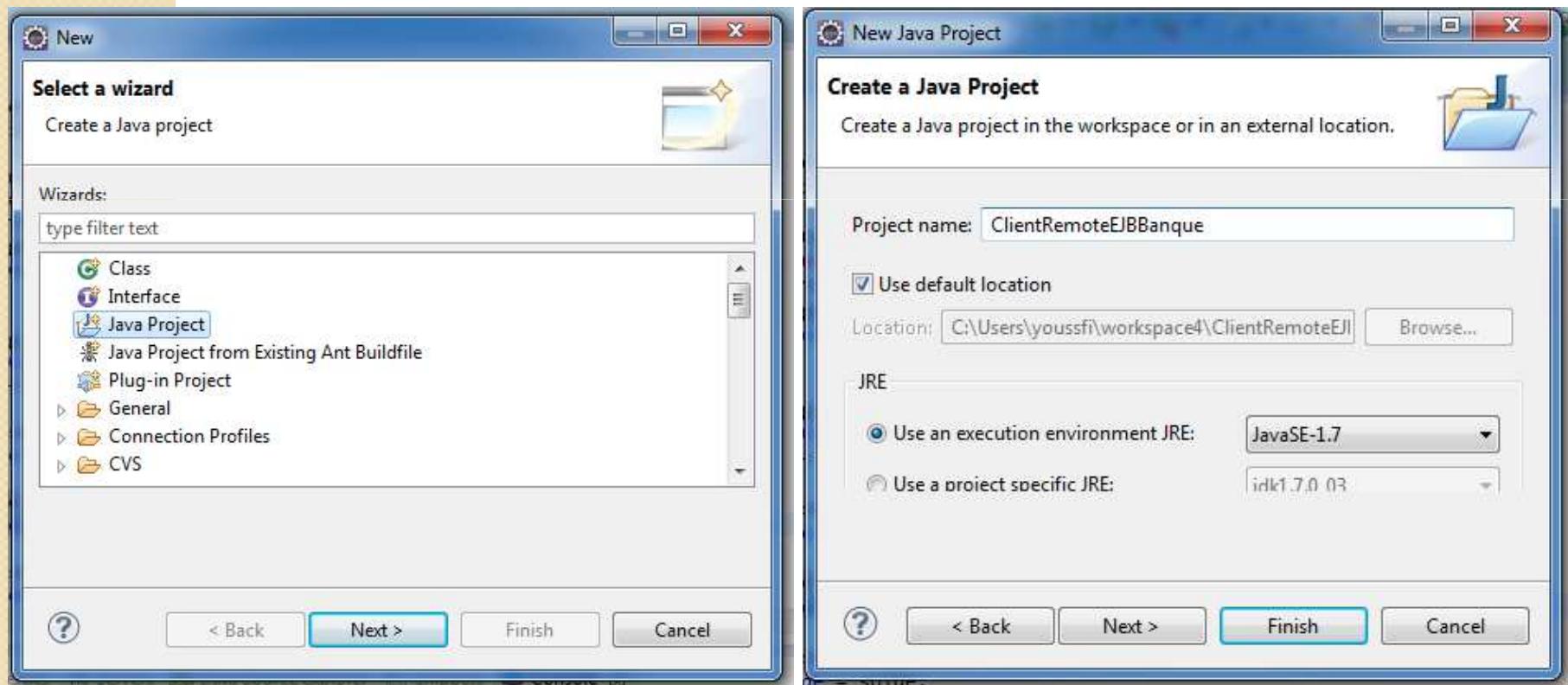
```
08:26:11,268 INFO [org.jboss.as] (MSC service thread 1-4) JBAS015951: Admin console listening on  
http://127.0.0.1:9990  
08:26:11,270 INFO [org.jboss.as] (MSC service thread 1-4) JBAS015874: JBoss AS 7.1.1.Final "Brontes" started in  
5261ms - Started 173 of 250 services (76 services are passive or on-demand)  
08:26:11,411 INFO [org.jboss.as.server] (DeploymentScanner-threads - 2) JBAS018559: Deployed "BanqueEJB.jar"
```

Projet déployé : BanqueEJB.jar



Client Java Remote

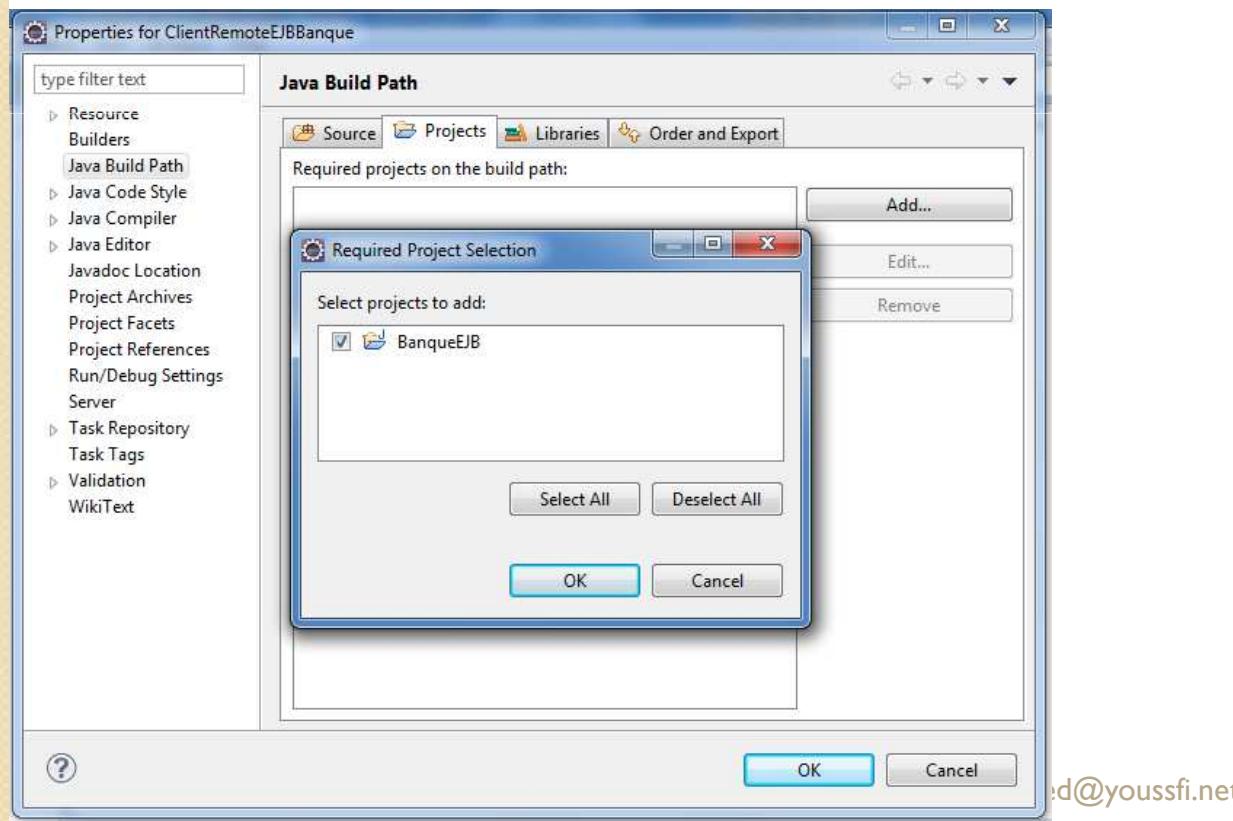
- Créer un nouveau projet Java pour le client
- New > Java Project



Le client a besoin de l'interface Remote de l'EJB Session et de l'entité Compte

- Première solution (Phase de Test):

- Lier le classpath du projet client au classpath du projet EJB. De cette façon, tous les composants du projet EJB seront accessible au projet Java
- Afficher la fenêtre propriétés du projet client java
- Java Build Path > Onglet Projects > Add



Ajouter jboss-client.jar au classpath du projet client

- Pour accéder à distance au à l'EJB session via le protocole RMI, le client a besoin d'un proxy dynamique (Stub) dont l'implémentation et ses dépendances sont définies dans un fichier jar unique « jboss-client.jar » fournie par jboss.
- Ce fichier se trouve dans le dossier bin/client de jboss
- Il faut l'ajouter au classpath du projet client
- Java Build Path > Librairies > Add External JARs...

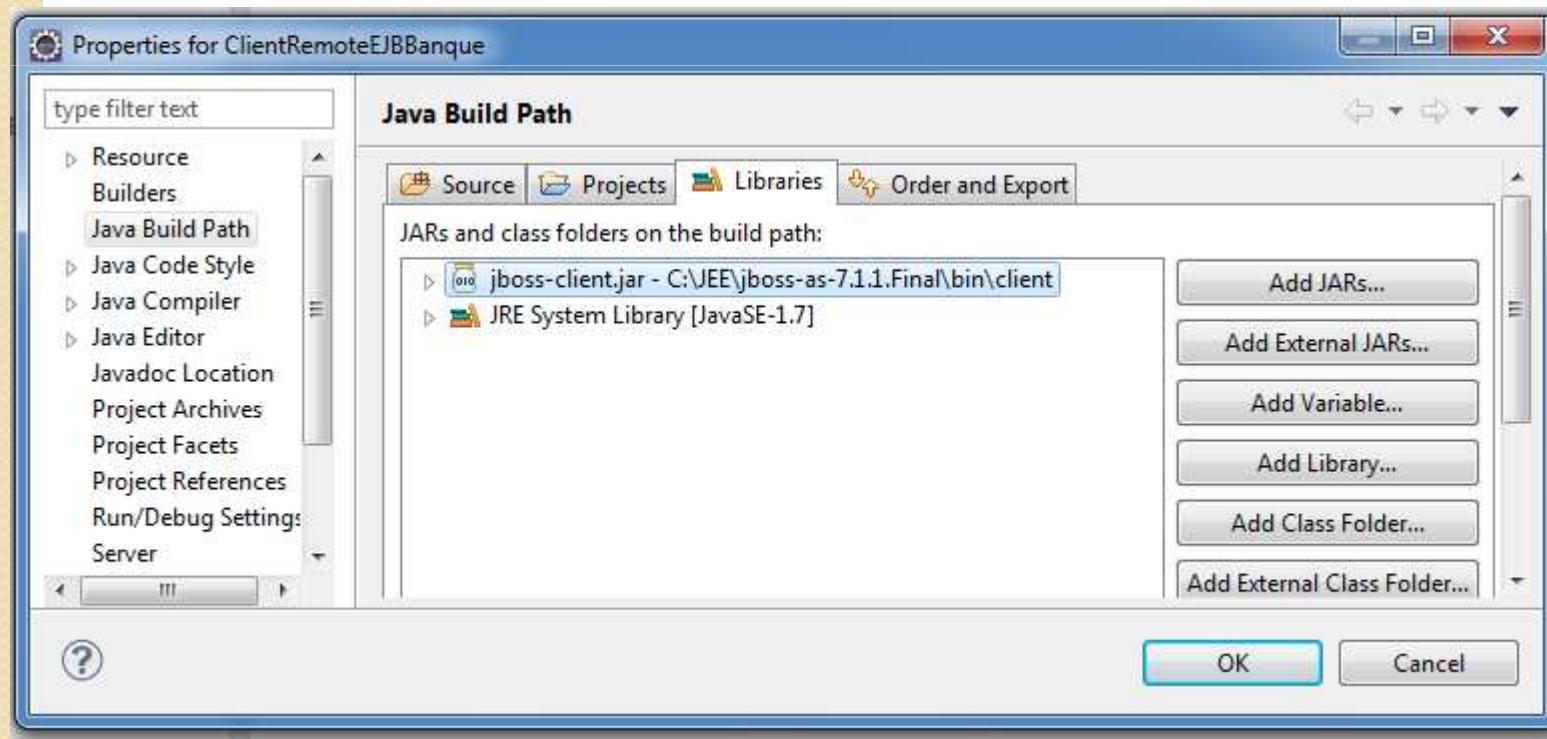
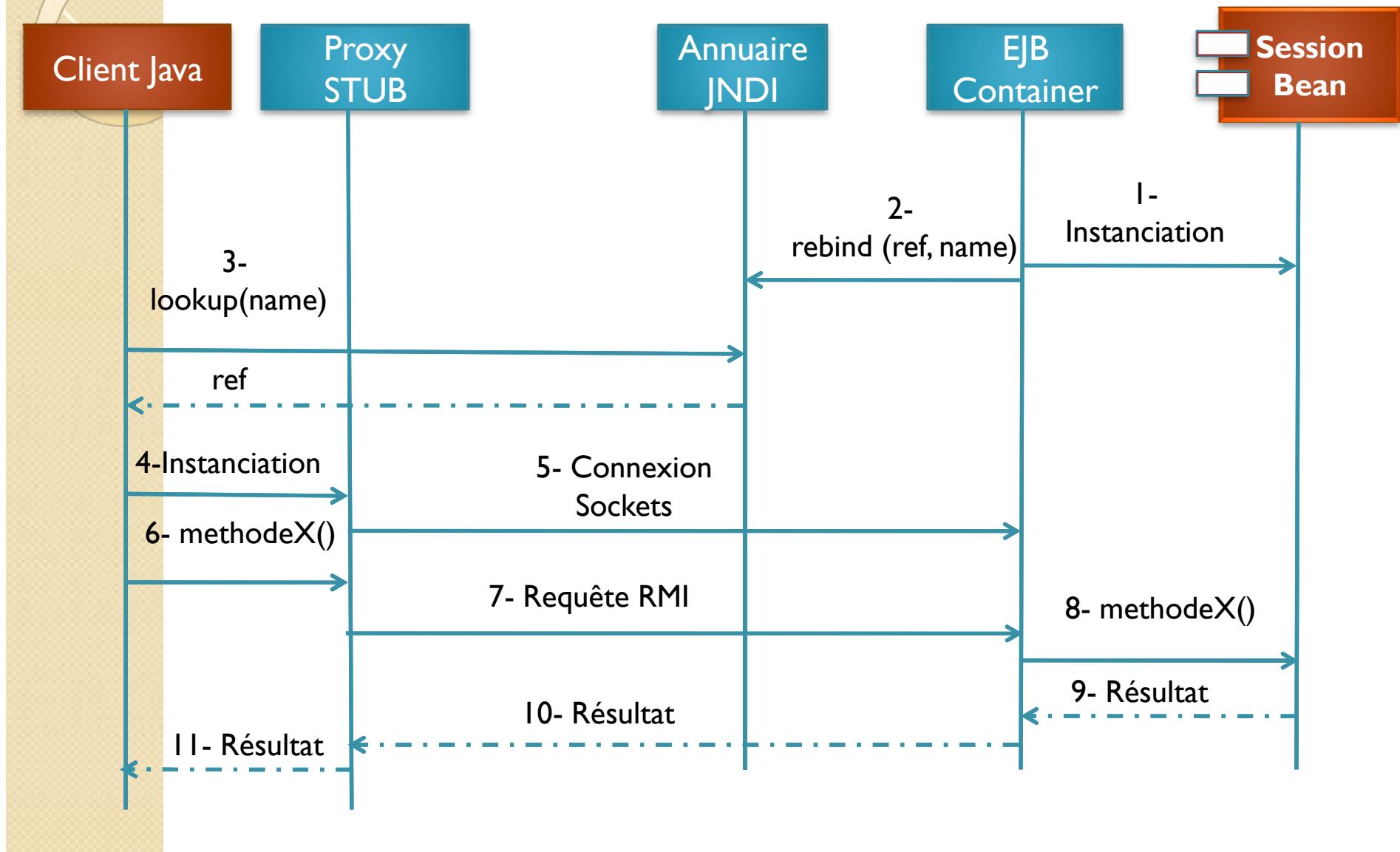


Diagramme de séquence de communication Remote entre le client java et l'EJB Session





Propriétés JNDI

- Pour que le client java puisse se connecter à l'annuaire JNDI pour chercher la référence remote de l'EJB, il a besoin de connaître un cernain nombre de propriétés JNDI.
- Ces propriétés sont généralement définies dans le fichier **jndi.properties**.
- Pour le cas de Jboss7, l'essentiel des propriétés peuvent être définie dans le fichier **jboss-ejb-client.properties**

Propriétés JNDI

jndi.properties

```
java.naming.factory.url.pkgs=org.jboss.ejb.client.naming
```

jboss-ejb-client.properties

```
endpoint.name=client-endpoint
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false
remote.connections=default
remote.connection.default.host=127.0.0.1
remote.connection.default.port = 4447
remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS=false
remote.connection.default.username=admin
remote.connection.default.password=
```

Client EJB

```
import java.util.*;import javax.naming.*;
import metier.entities.Compte; import metier.session.IBanqueRemote;
public class ClientEJB {
public static void main(String[] args) {
try {
String appName=""; String moduleName="BanqueEJB";
String distinctName="BP2"; String viewClassName=IBanqueRemote.class.getName();
Context ctx=new InitialContext();
String ejbRemoteJNDIName="ejb:"+appName+"/"+moduleName+"/"+distinctName+"!"+viewClassName;
System.out.println(ejbRemoteJNDIName);
IBanqueRemote stub=(IBanqueRemote) ctx.lookup(ejbRemoteJNDIName);
System.out.println("Affichage de tous les comptes");
List<Compte> cptes=stub.consulterComptes();
for(Compte cp:cptes) System.out.println(cp);
System.out.println("Versement d'un montant"); stub.verser(2L, 22000);
System.out.println("Consultation du compte 2"); Compte cp=stub.consulterCompte(2L);
System.out.println(cp);
System.out.println("Retrait d'un montant"); stub.retirer(1L, 3000);
System.out.println("Consultation du compte 1"); cp=stub.consulterCompte(1L);
System.out.println(cp);
} catch (Exception e) { e.printStackTrace();}
}
```

Résultat de l'exécution

```
nov. 10, 2013 11:05:34 AM org.jboss.ejb.client.EJBClient <clinit>
INFO: JBoss EJB Client version 1.0.5.Final
nov. 10, 2013 11:05:35 AM org.xnio.Xnio <clinit>
INFO: XNIO Version 3.0.3.GA
nov. 10, 2013 11:05:35 AM org.xnio.nio.NioXnio <clinit>
INFO: XNIO NIO Implementation Version 3.0.3.GA
nov. 10, 2013 11:05:35 AM org.jboss.remoting3.EndpointImpl <clinit>
INFO: JBoss Remoting version 3.2.3.GA
nov. 10, 2013 11:05:35 AM org.jboss.ejb.client.remoting.VersionReceiver handleMessage
INFO: Received server version 1 and marshalling strategies [river]
nov. 10, 2013 11:05:35 AM org.jboss.ejb.client.remoting.RemotingConnectionEJBReceiver associate
INFO: Successful version handshake completed for receiver context EJBReceiverContext{clientContext=org.jboss.ejb.client.EJBClientContext@10ca25a, receiver=Remoting connection EJB receiver [connection=Remoting connection <9f26c7>,channel=jboss.ejb,nodename=win-7pa448puor]} on channel Channel ID a258e3f6 (outbound) of Remoting connection 0006812d to /127.0.0.1:4447
nov. 10, 2013 11:05:35 AM org.jboss.ejb.client.remoting.ChannelAssociation$ResponseReceiver handleMessage
WARN: Unsupported message received with header 0xffffffff
```

Affichage de tous les comptes

ejb:/BanqueEJB/BP2!metier.session.IBanqueRemote

Compte [code=3, solde=2000.0, dateCreation=Sun Nov 10 11:05:35 WET 2013]

Compte [code=2, solde=4000.0, dateCreation=Sun Nov 10 11:05:35 WET 2013]

Compte [code=1, solde=7000.0, dateCreation=Sun Nov 10 11:05:35 WET 2013]

Versement d'un montant

Consultation du compte 2

Compte [code=2, solde=26000.0, dateCreation=Sun Nov 10 11:05:35 WET 2013]

Retrait d'un montant

Consultation du compte 1

Compte [code=1, solde=4000.0, dateCreation=Sun Nov 10 11:05:35 WET 2013]

med@youssfi.net

Méthodes du cycle de vie d'un session bean

Afin de gérer le cycle de vie d'un session bean, le conteneur EJB doit posséder aux étapes suivantes :

1. Instanciation du session Bean. Cette étape s'effectue en faisant appel à la méthode `newInstance()` de l'objet Class lié à la classe du session bean:

```
Class c=Class.forName("metier.session.BanqueImpl");  
Object o=c.newInstance();
```

Cela suppose donc que la classe du bean dispose d'un constructeur par défaut.

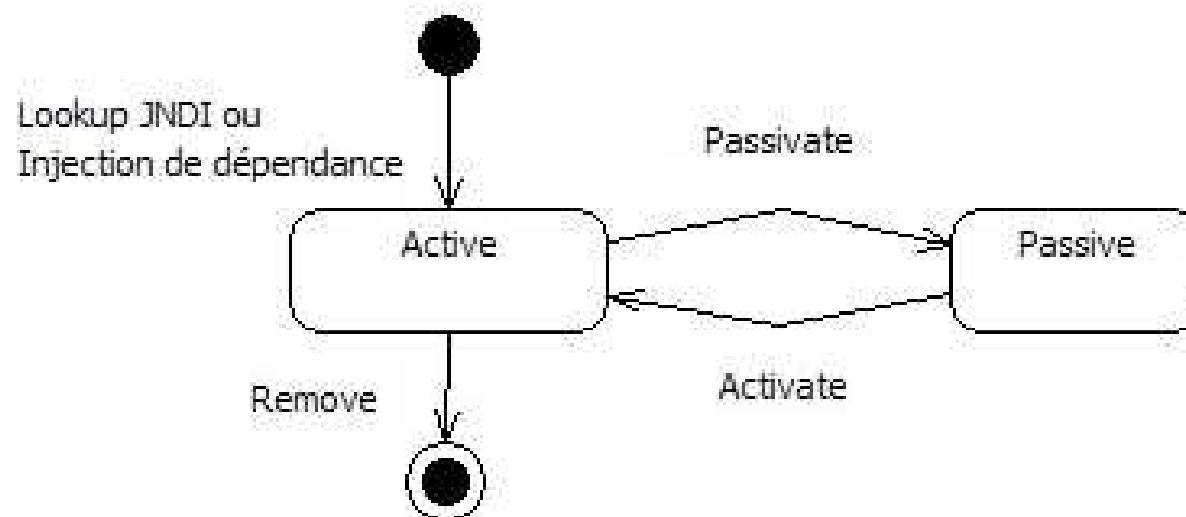
2. Le conteneur va analyser ensuite cette instance afin de vérifier la présence d'éventuelles injection de dépendances à effectuer.
 - Cette opération permet d'associer l'instance du session Bean à d'autres objet déployés dans le conteneur et qui seront sollicités par les méthodes de l'ejb session.
 - Le développeur peut spécifier ces dépendances grâce à quelques annotation comme `@Ressource`, `@EJB`, `@PersistenceContext`, ...



Méthodes du cycle de vie d'un session bean

4. Après l'injection des dépendances, le conteneur va exécuter les **callback interceptors** annotés avec **@PostConstruct**
5. Avant la destruction du session Bean, le conteneur peut exécuter les méthodes annotées **@PreDestroy**
6. Quand il s'agit d'un EJB Session de type Statful, si le conteneur a besoin d'espace mémoire, il procède à la sérialisation du session bean. Ce qui permet de le faire passer à l'état « passivation ». Dans ce cas si une méthode du bean est annotée **@PrePassivate**, elle sera exécutée avant la sérialisation
7. Quand il s'agit d'un EJB Session de type Statful au moment de la désérialisation du session bean, la méthode annotée **@PostPassivate**, sera exécutée.
8. L'instance du session bean (Statfull) peut être supprimée lorsque le client fait appel à une méthode annotée **@Remove** .

Cycle de vie d'un statful





Intercepteurs : Gestion avancée du cycle de vie

- L'utilisation des méthodes de type **callback interceptors**, est une pratique facile à mettre en place.
- Toutefois l'inconvénient principal de cette approche est le mélange qu'elle génère entre le code métier et le code du cycle de vie du session bean.
- Il est cependant possible de définir le code de gestion du cycle de vie dans une classe séparée appelée classe « *Interceptor* »
- *L'association d'une classe d'intercepteur à un session bean peut être effectuée à l'aide de l'annotation `@Interceptors` qui prend en argument la ou les classes d'intercepteurs.*
- Un intercepteur peut servir à intercepter les changements du cycle de vie du session bean, mais peut également servir à intercepter les appels aux méthodes métier.



Utilisation des intercepteurs

- Supposant que nous souhaitons loger au niveau du serveur, pour chaque méthode de l'EJB invoqué,
 - Le nom de la méthode
 - La date et le temps d'invocation
 - La durée d'exécution de la méthode.
- Dans les solutions classiques, il faudrait écrire ce même code dans toutes les méthodes de l'EJB session.
- Dans ce cas là, nous allons mélanger le code métier avec ce code technique.
- La programmation orientée Aspect est venue pur résoudre ce problème.
- L'AOP permet de séparer le code métier du code technique.
- Un intercepteur EJB est un aspect dans lequel nous pourrons ajouter ce code technique sans avoir besoin de modifier notre code métier.

Déclarer les intercepteurs d'un EJB Session.

```
package metier.session;
import interceptors.BanqueInterceptor; import java.util.*;
import javax.ejb.Singleton;
import javax.interceptor.Interceptors; import metier.entities.Compte;
@Singleton(name="BP2")
@Interceptors({BanqueInterceptor.class})
public class BanqueEjbImpl implements IBanqueLocal,IBanqueRemote {
private Map<Long, Compte> comptes=new Hashtable<Long, Compte>();
@Override
public void addCompte(Compte c) {
comptes.put(c.getCode(), c);
}
// Suite des méthodes métiers déclarées dans les interfaces
}
```

Implémentation de l'intercepteur

```
package interceptors;

import java.util.Date; import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy; import javax.ejb.PostActivate;
import javax.ejb.PrePassivate;import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext; import metier.entities.Compte;
import metier.session.IBanqueLocal; import org.jboss.logging.Logger;
public class BanqueInterceptor {

    private Logger log=Logger.getLogger(BanqueInterceptor.class);

    @PostConstruct
    public void initialisation(InvocationContext ctx){
        IBanqueLocal metier=(IBanqueLocal) ctx.getTarget();
        metier.addCompte(new Compte(1L, 7000, new Date()));
        metier.addCompte(new Compte(2L, 4000, new Date()));
        metier.addCompte(new Compte(3L, 2000, new Date()));
        log.info("Initialisation Ajout de 3 Comptes");
    }
}
```

Implémentation de l'intercepteur

```
@PreDestroy  
public void destruction(){  
    System.out.println("Destruction de l'instance");  
}  
  
@PrePassivate  
public void serialisation(){  
    System.out.println("Sérialisation");  
}  
  
@PostActivate  
public void deserialisation(){  
    System.out.println("Désérialisation");  
}
```

Implémentation de l'intercepteur

@AroundInvoke

```
public Object test(InvocationContext ctx) throws Exception{  
    long t1=System.currentTimeMillis();  
    long t2;  
    try {  
        Object o=ctx.proceed();  
        return o;  
    } finally {  
        t2=System.currentTimeMillis();  
        String methodName=ctx.getMethod().getName();  
        Object[] parameters=ctx.getParameters();  
        String params="";  
        for(Object p:parameters) params+=p;  
        log.info(new Date()+" Appel de la méthode "+methodName+" Params "+params+"  
        Durée :"+(t2-t1));  
    }  
}
```

Console du serveur après Exécution du client

```
11:40:11,480 INFO [interceptors.BanqueInterceptor] (EJB default - 1)
Initialisation Ajout de 3 Comptes

11:40:11,482 INFO [interceptors.BanqueInterceptor] (EJB default - 1) Sun Nov 10
  11:40:11 WET 2013 Appel de la méthode consulterComptes Params Durée :0

11:40:11,570 INFO [interceptors.BanqueInterceptor] (EJB default - 2) Sun Nov 10
  11:40:11 WET 2013 Appel de la méthode verser Params 222000.0 Durée :0

11:40:11,579 INFO [interceptors.BanqueInterceptor] (EJB default - 3) Sun Nov 10
  11:40:11 WET 2013 Appel de la méthode consulterCompte Params 2 Durée :0

11:40:11,586 INFO [interceptors.BanqueInterceptor] (EJB default - 4) Sun Nov 10
  11:40:11 WET 2013 Appel de la méthode retirer Params 13000.0 Durée :0

11:40:11,593 INFO [interceptors.BanqueInterceptor] (EJB default - 5) Sun Nov 10
  11:40:11 WET 2013 Appel de la méthode consulterCompte Params 1 Durée :0

11:46:02,629 INFO [interceptors.BanqueInterceptor] (EJB default - 6) Sun Nov 10
  11:46:02 WET 2013 Appel de la méthode consulterComptes Params Durée :0

11:46:02,702 INFO [interceptors.BanqueInterceptor] (EJB default - 7) Sun Nov 10
  11:46:02 WET 2013 Appel de la méthode verser Params 222000.0 Durée :0

11:46:02,708 INFO [interceptors.BanqueInterceptor] (EJB default - 8) Sun Nov 10
  11:46:02 WET 2013 Appel de la méthode consulterCompte Params 2 Durée :0

11:46:02,714 INFO [interceptors.BanqueInterceptor] (EJB default - 9) Sun Nov 10
  11:46:02 WET 2013 Appel de la méthode retirer Params 13000.0 Durée :0

11:46:02,720 INFO [interceptors.BanqueInterceptor] (EJB default - 10) Sun Nov 10
  11:46:02 WET 2013 Appel de la méthode consulterCompte Params 1 Durée :0
```

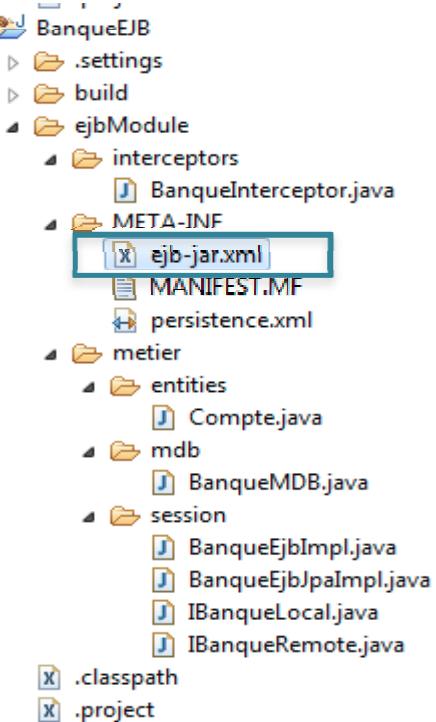


Intercepteurs par défaut

- Il est possible de définir des intercepteurs par défaut qui seront appliqués à tous les EJB d'un même jar.
- La définition d'un intercepteur par défaut ne peut se faire que dans le descripteur de déploiement **ejb-jar.xml**. Ils ne peuvent pas être définis par des annotations.
- Pour déclarer un intercepteur par défaut, il faut modifier le descripteur de déploiement ejb-jar.xml en utilisant un tag **<interceptor-binding>** fils du tag **<assembly-descriptor>**.
- Le tag **<interceptor-binding>** peut avoir deux tags fils :
 - **<ejb-name>** dont la valeur précise un filtre sur les ejb-name qui indique les EJB concernés. La valeur ***** permet d'indiquer que tous les EJB sont concernés.
 - **<interceptor-class>** permet de préciser la classe pleinement qualifiée de l'intercepteur
- Dans ce cas, nous n'avons plus besoin de préciser le nom de l'intercepteur dans la classe de l'EJB.
- Ce qui permet d'ajouter d'autres intercepteurs sans toucher le code source de l'application.
- Avec cette technique, nous pourrons séparer le code métier du code technique.

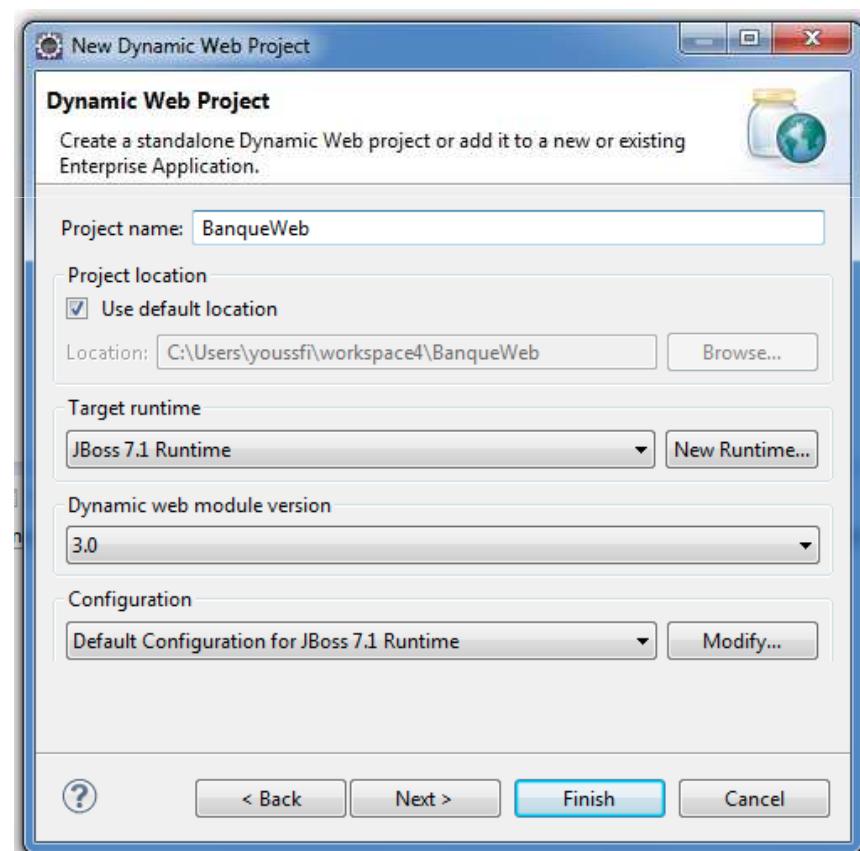
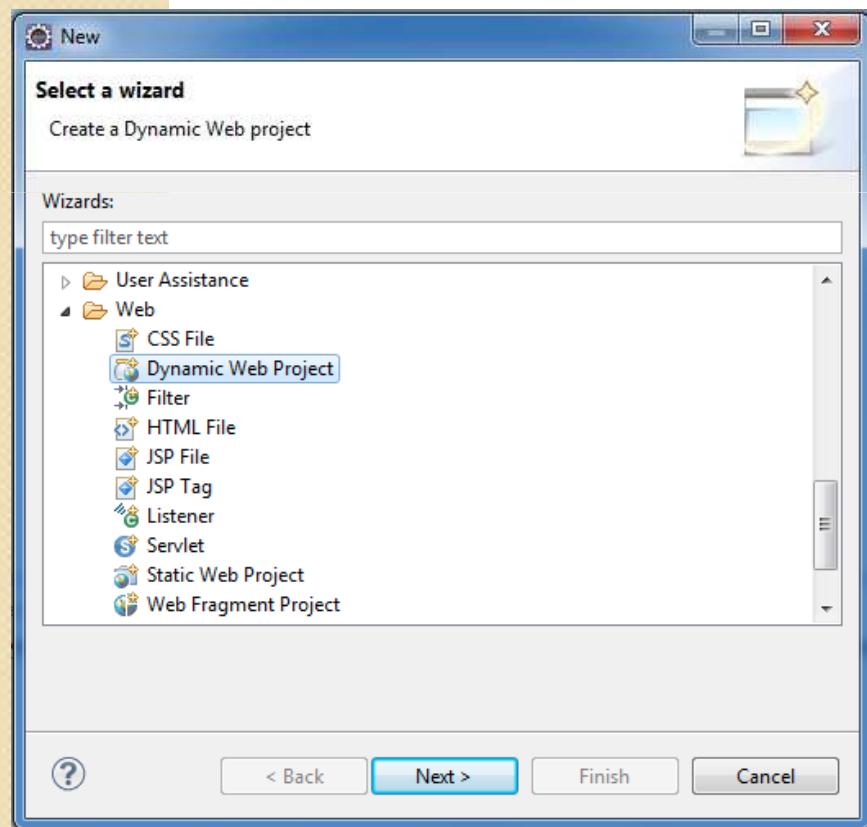
Déclaration des intercepteurs dans le fichier ejb-jar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.1" 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_1.xsd">
  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>BP2</ejb-name>
      <interceptor-class>interceptors.BanqueInterceptor</interceptor-class>
    </interceptor-binding>
  </assembly-descriptor>
</ejb-jar>
```

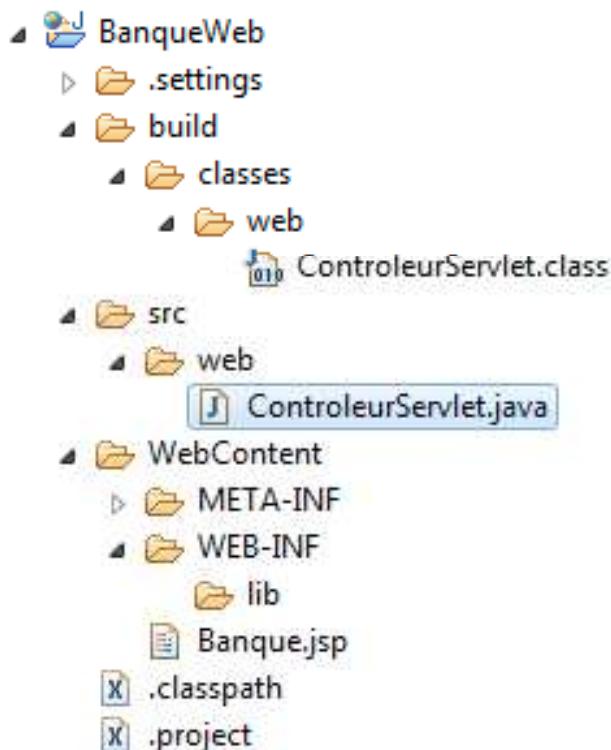


Client Web

- Créer un nouveau projet web dynamique qui sera déployé dans Jboss

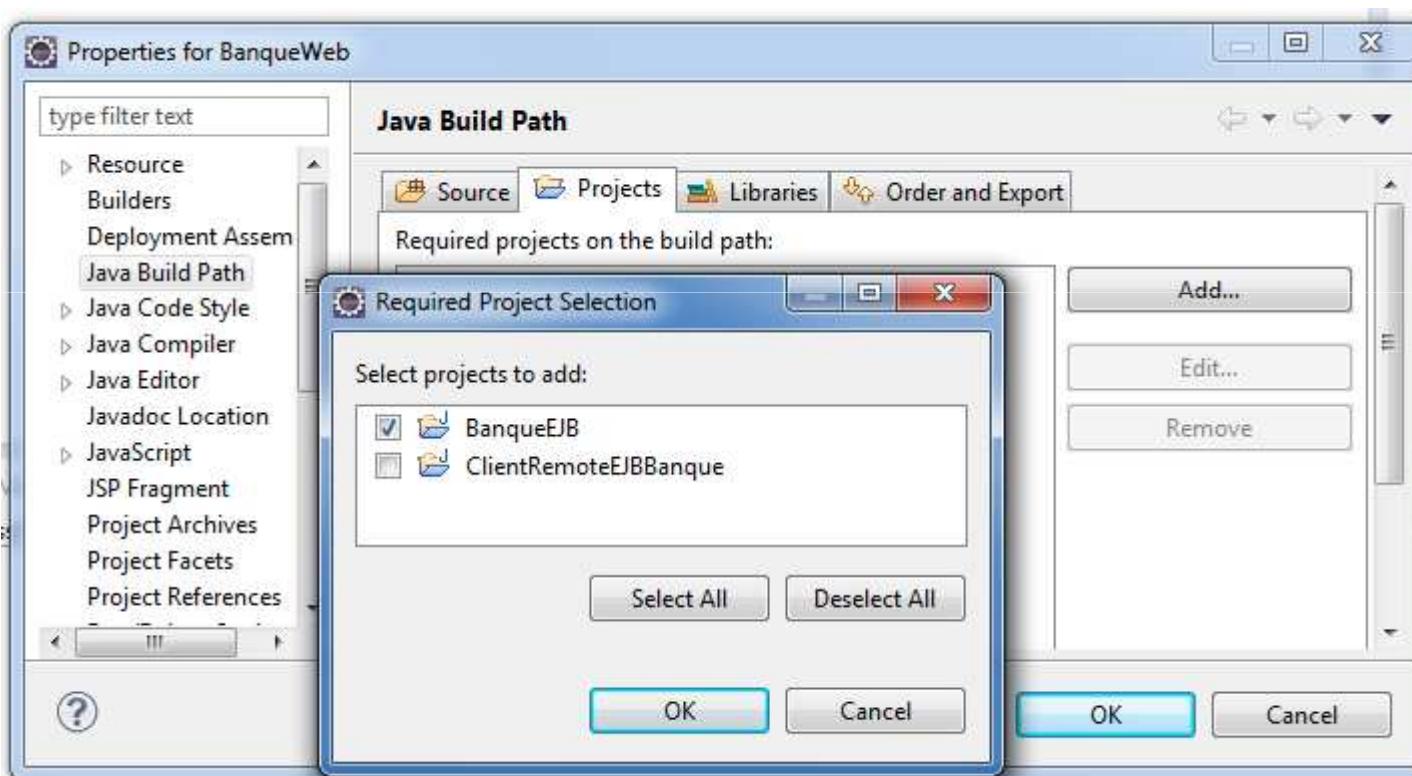


Structure du projet

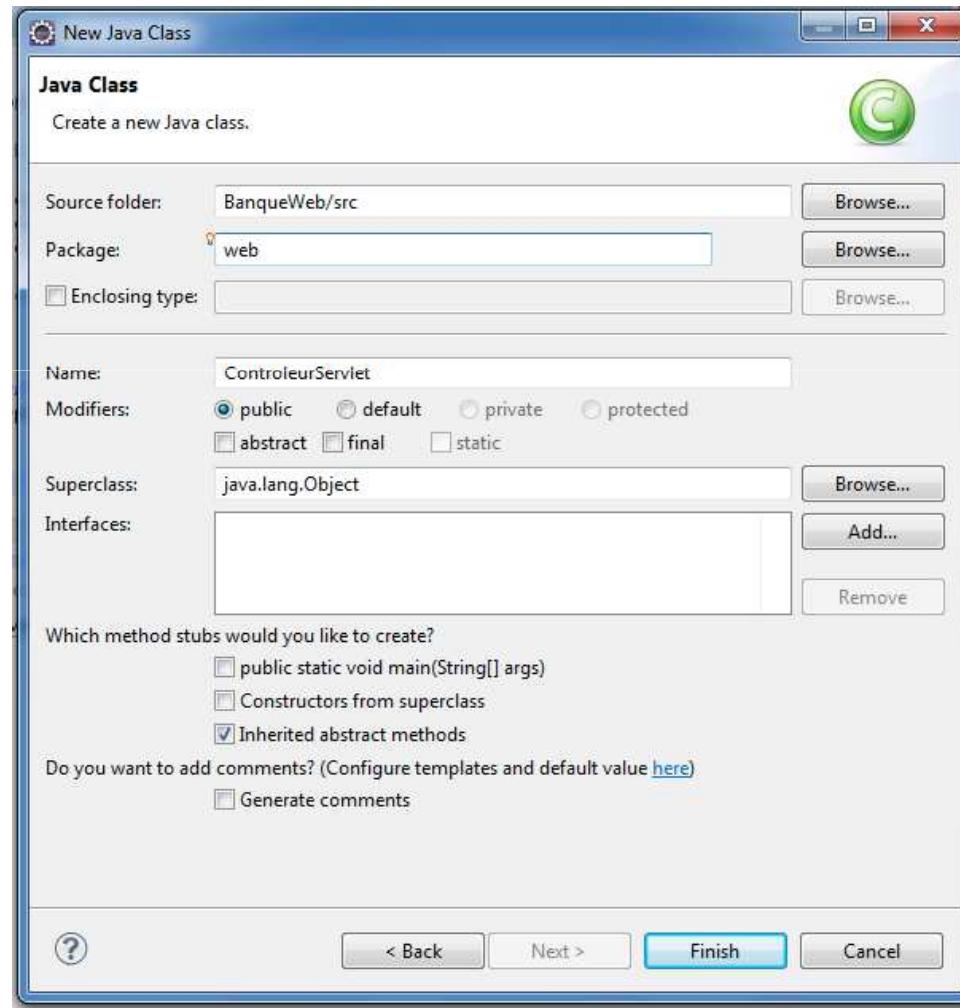


Dépendances du projet

- Propriétés du projet > Java Build Path>Projects > Add



ControleurServlet



med@youssfi.net

Code du contrôleur

```
package web;

import java.io.IOException; import javax.ejb.EJB;
import javax.servlet.ServletException;import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse; import metier.entities.Compte;
import metier.session.IBanqueLocal;

@WebServlet(name="cs",urlPatterns={"/controleur"})
public class ControleurServlet extends HttpServlet {

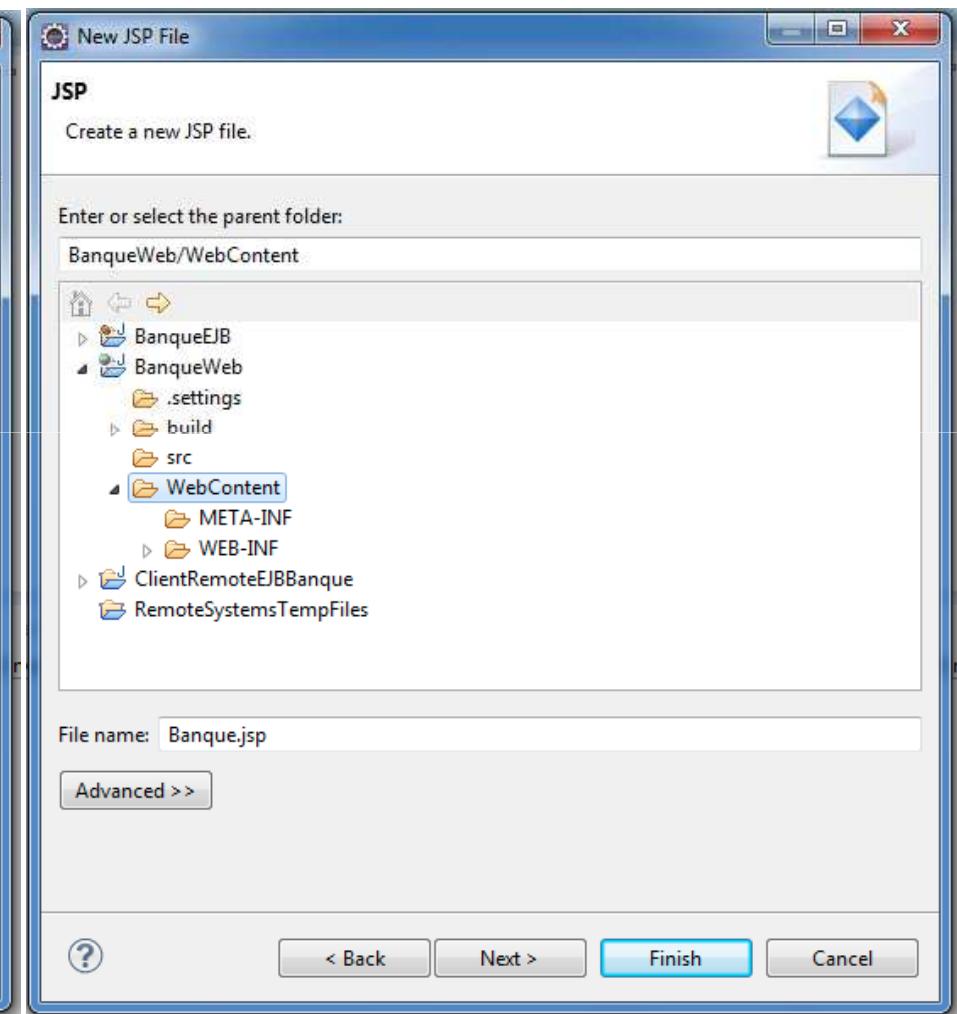
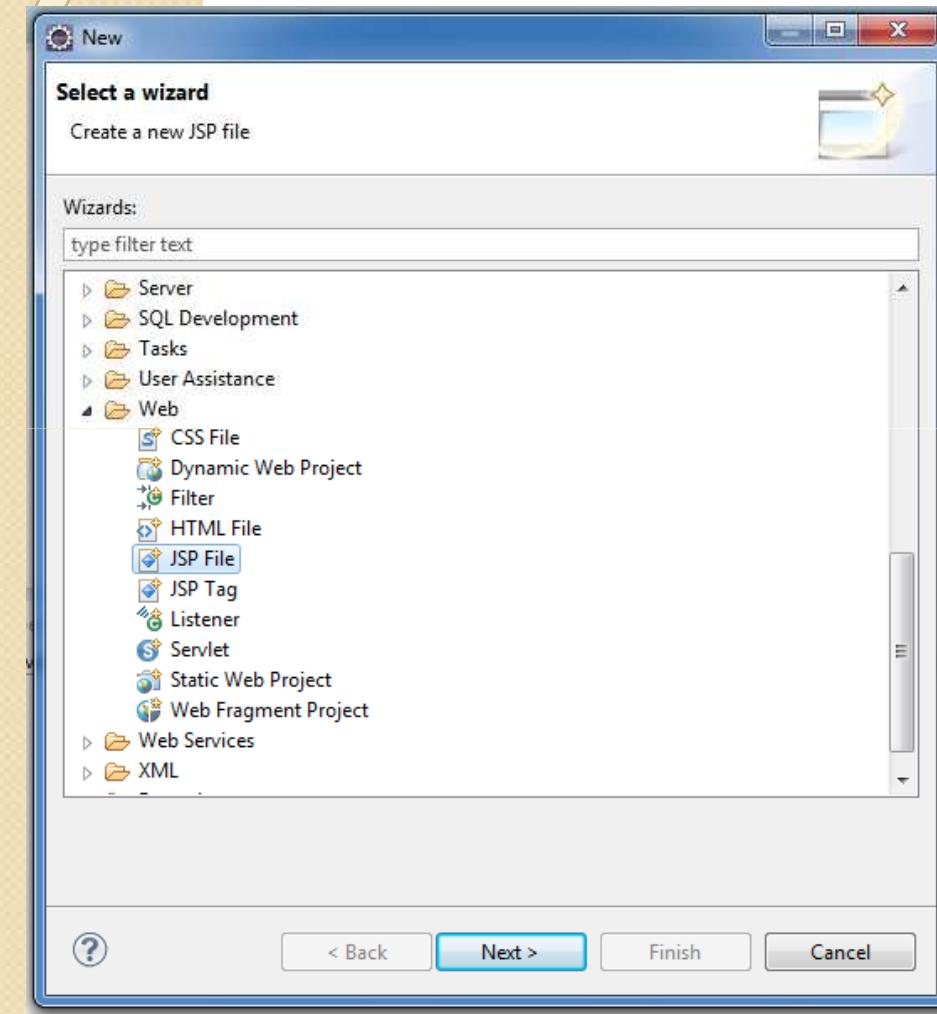
    @EJB
private IBanqueLocal metier;

    @Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    try {String action=request.getParameter("action");
if(action!=null){
if(action.equals("Consulter")){
Long code=Long.parseLong(request.getParameter("code"));
request.setAttribute("code", code);
Compte cp=metier.consulterCompte(code);
request.setAttribute("compte", cp);
}
}
}
}
```

Code du contrôleur (Suite)

```
else if(action.equals("Tous Les comptes")){
    request.setAttribute("comptes", metier.consulterComptes());
}
else if((action.equals("Verser"))||(action.equals("Retirer"))){
    double montant=Double.parseDouble(request.getParameter("montant"));
    Long code=Long.parseLong(request.getParameter
("code"));
    request.setAttribute("montant", montant);
    request.setAttribute("code", code);
    if(action.equals("Verser")){
        metier.verser(code, montant);
    }
    else{
        metier.retirer(code, montant);
    }
    request.setAttribute("compte", metier.consulterCompte(code));
}
}
catch (Exception e) {request.setAttribute("exception", e.getMessage());}
request.getRequestDispatcher("Banque.jsp").forward(request, response);
}
```

La vue JSP



med@youssfi.net

Implémentation de la JSP

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html> <head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
    <div id="formRecherche">
        <form action="contrroleur" method="get">
            <table>
                <tr>
                    <td> Code :</td>
                    <td><input type="number" name="code" value="${code}" required="required"></td>
                    <td>${errCode}</td>
                    <td><input type="submit" name="action" value="Consulter"></td>
                    <td><input type="submit" name="action" value="Tous Les comptes"></td>
                </tr>
            </table>
        </form>
    </div>
```

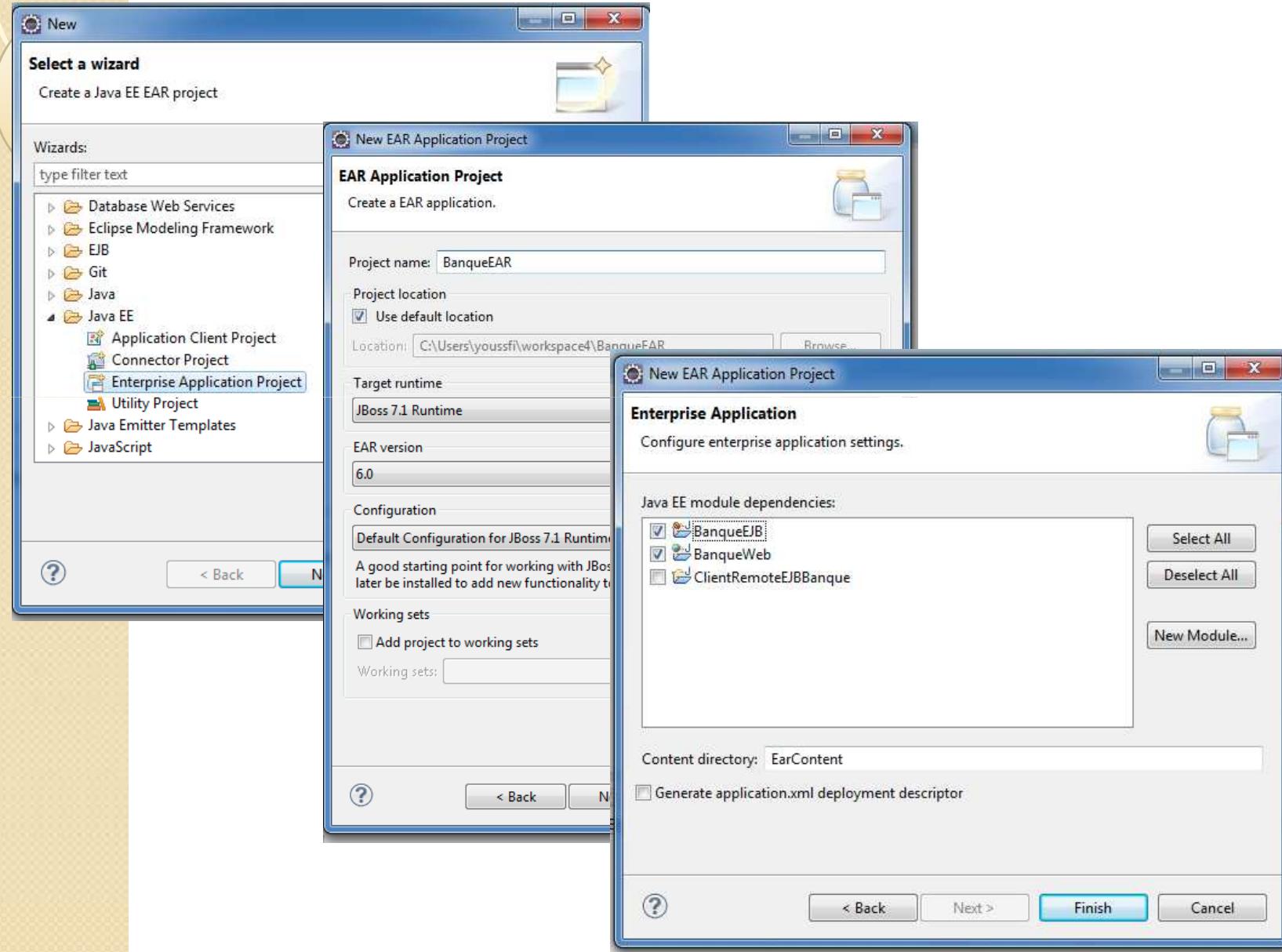
Implémentation de la JSP

```
<c:if test="#{(compte!=null)||(mtMsgErr!=null)}">
    <div id="compte">
        <table>
            <tr> <td>Code:</td><td>${compte.code}</td> </tr>
            <tr><td>Solde:</td><td>${compte.solde}</td> </tr>
            <tr><td>Date Création:</td><td>${compte.dateCreation}</td></tr>
        </table>
    </div>
    <div id="formOperations">
        <form action="contrôleur" method="get">
            <table><tr>
                <td><input type="hidden" name="code" value="${code}"></td>
                <td><input type="number" name="montant" required="required"
value="${montant}"></td>
                <td>${mtMsgERR}</td>
                <td><input type="submit" name="action" value="Verser"></td>
                <td><input type="submit" name="action" value="Retirer"></td>
            </tr></table>
        </form></div>
    </c:if>
```

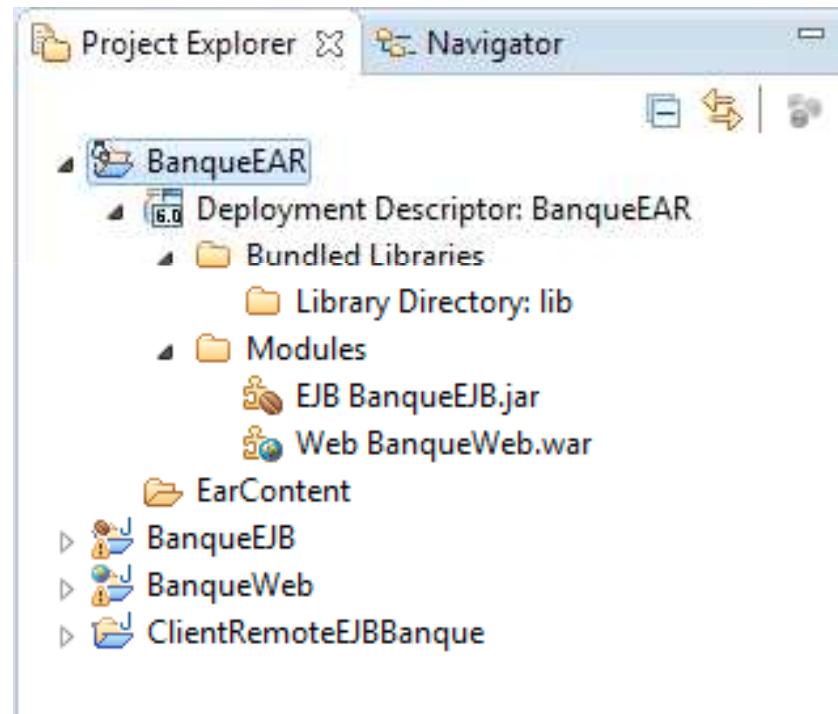
Implémentation de la JSP

```
<c:if test="#{comptes!=null}">
  <div id="ListeComptes">
    <table border="1" width="80%">
      <tr> <th>CODE</th><th>SOLDE</th><th>DATE CREATION</th></tr>
      <c:forEach items="#{comptes}" var="cp">
        <tr>
          <td>${cp.code}</td> <td>${cp.solde}</td> <td>${cp.dateCreation}</td>
        </tr>
      </c:forEach>
    </table>
  </div>
</c:if>
<div id="errors">
  ${exception}
</div>
</body>
</html>
```

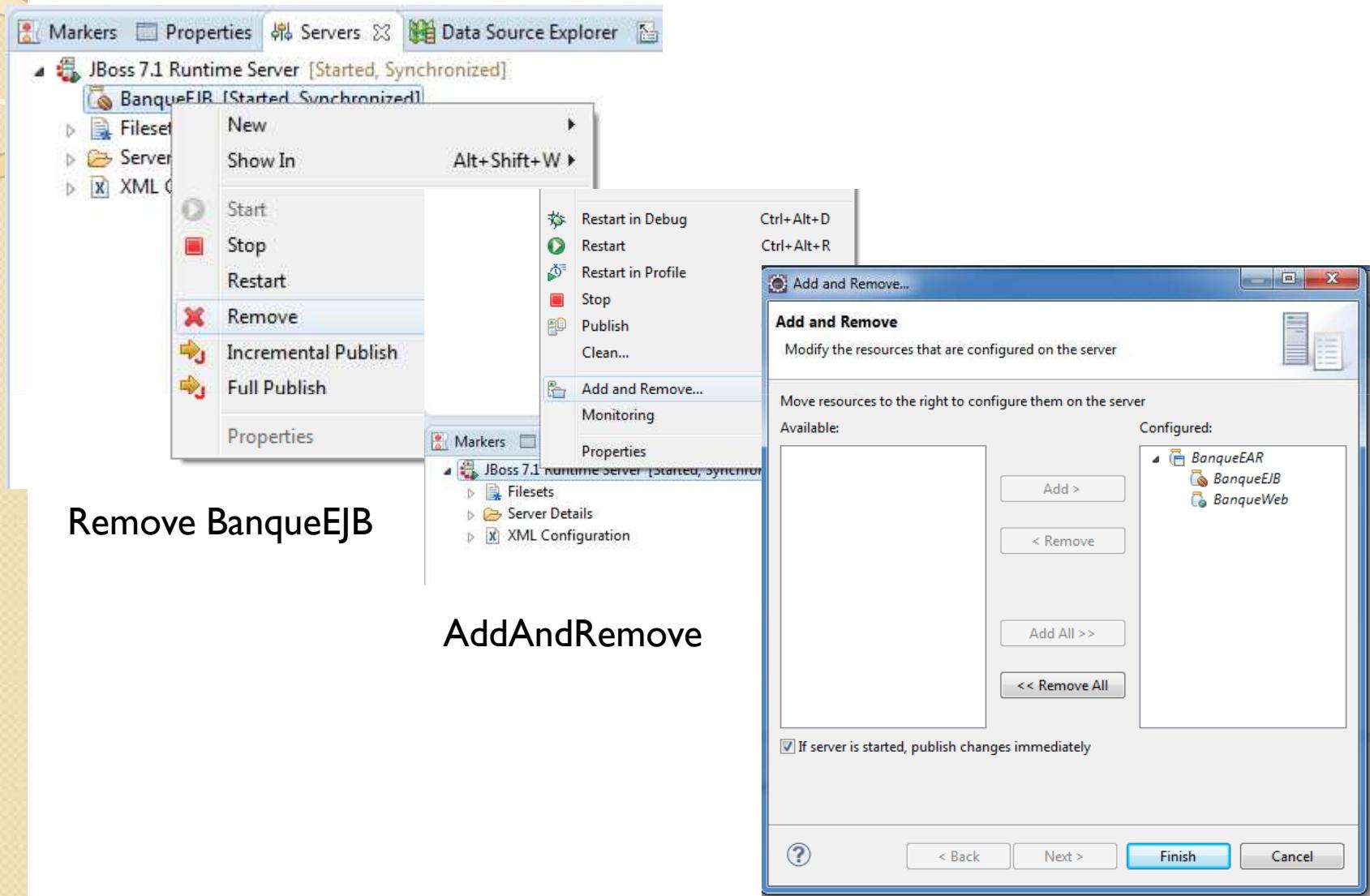
Entrepri\$e Archive Project



Structure du projet EAR



Déployer le Projet EAR



med@youssfi.net Add BanqueEAR

Console du serveur après déploiement

12:22:04,431 INFO

[org.jboss.as.ejb3.deployment.processors.EjbJndiBindingsDeploymentUnitProcessor] (MSC service thread 1-8) JNDI bindings for session bean named BP2 in deployment unit subdeployment "BanqueEJB.jar" of deployment "BanqueEAR.ear" are as follows:

java:global/BanqueEAR/BanqueEJB/BP2!metier.session.IBanqueRemote

java:app/BanqueEJB/BP2!metier.session.IBanqueRemote

java:module/BP2!metier.session.IBanqueRemote

java:jboss/exported/BanqueEAR/BanqueEJB/BP2!metier.session.IBanqueRemote

java:global/BanqueEAR/BanqueEJB/BP2!metier.session.IBanqueLocal

java:app/BanqueEJB/BP2!metier.session.IBanqueLocal

java:module/BP2!metier.session.IBanqueLocal

12:22:04,861 INFO [org.jboss.web] (MSC service thread 1-1) JBAS018210:

Registering web context: /BanqueWeb

12:22:04,907 INFO [org.jboss.as.server] (DeploymentScanner-threads - 2)

JBAS018559: Deployed "BanqueEAR.ear"

Tester le client WEB

A screenshot of a web browser window titled "Insert title here". The URL is "localhost:8080/BanqueWeb/controleur?code=1&montant=4000". The form contains:

Code :

Code: 1
Solde: 11000.0
Date Création: Sun Nov 10 12:30:11 WET 2013

A screenshot of a web browser window titled "Insert title here". The URL is "localhost:8080/BanqueWeb/controleur?code=1&action=Tous+Les+Comptes". The table displays all accounts:

CODE	SOLDE	DATE CREATION
3	2000.0	Sun Nov 10 12:30:11 WET 2013
2	4000.0	Sun Nov 10 12:30:11 WET 2013
1	11000.0	Sun Nov 10 12:30:11 WET 2013

A screenshot of a web browser window titled "Insert title here". The URL is "localhost:8080/BanqueWeb/controleur?code=6&action=Consulte". The form contains:

Code :

java.lang.RuntimeException: Compte introuvable

A screenshot of a web browser window titled "Insert title here". The URL is "localhost:8080/BanqueWeb/controleur?code=2&action=Consulte". The form contains:

Code :

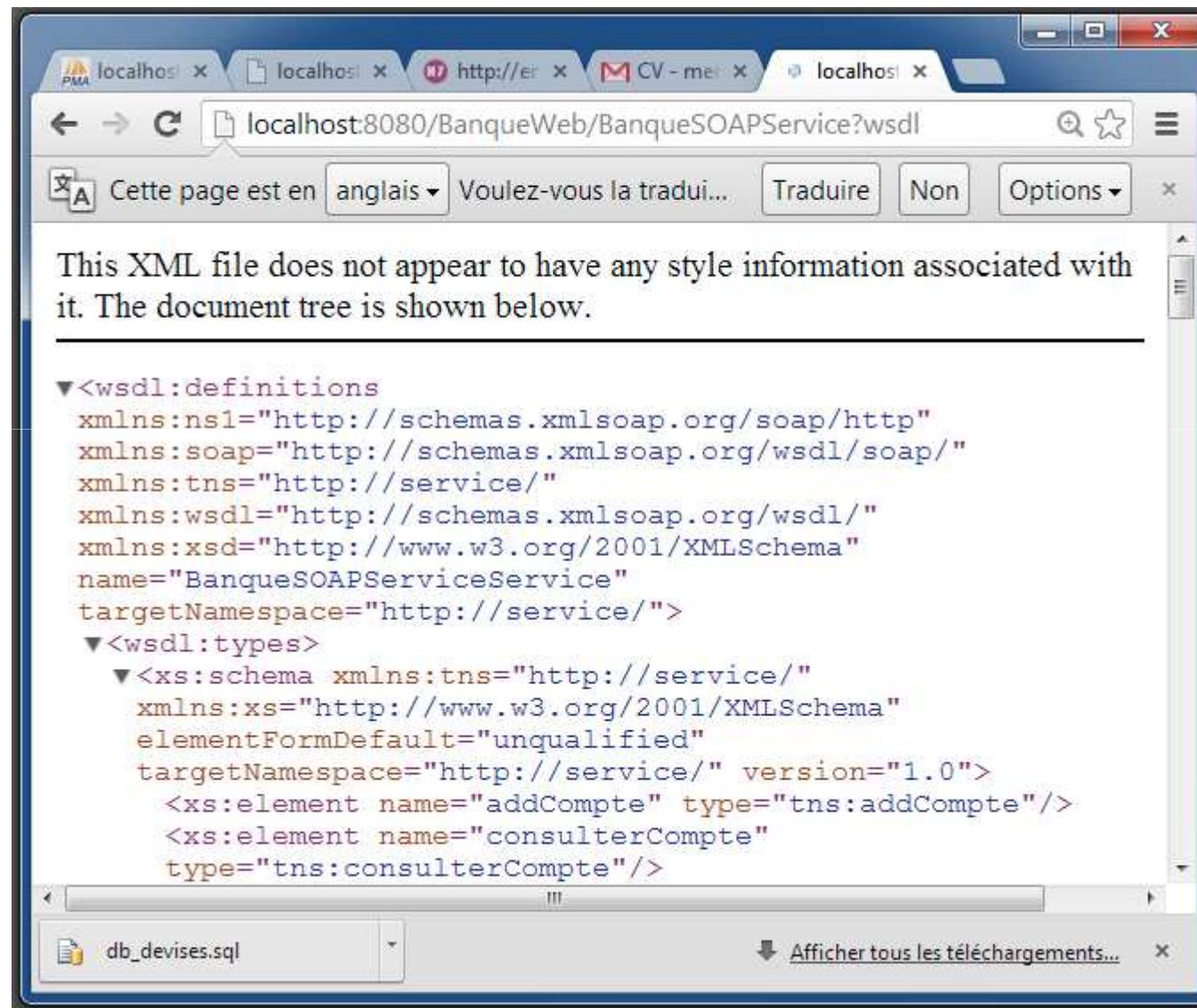
Code: 2 ! Veuillez saisir un nombre.
Solde: 4000.0

Date Création: Sun Nov 10 12:30:11 WET 2013

Web Service

```
package service;
import java.util.*;import javax.ejb.EJB; import javax.jws.*; import
metier.entities.Compte;
import metier.session.IBanqueLocal;
@WebService
public class BanqueSOAPService {
    @EJB(beanName="BP3")
    private IBanqueLocal metier;
    @WebMethod
    @Oneway
    public void addCompte(@WebParam(name="solde") double solde){
        metier.addCompte(new Compte(null,solde,new Date()));
    }
    public List<Compte> consulterComptes(){
        return metier.consulterComptes();
    }
}
```

WSDL



The screenshot shows a web browser window displaying a WSDL (Web Services Description Language) document. The URL in the address bar is `localhost:8080/BanqueWeb/BanqueSOAPService?wsdl`. The page content is as follows:

```
<?xml version="1.0"?>
<wsdl:definitions
    xmlns:ns1="http://schemas.xmlsoap.org/soap/http"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://service/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    name="BanqueSOAPServiceService"
    targetNamespace="http://service/">
    <wsdl:types>
        <xsd:schema xmlns:tns="http://service/"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="unqualified"
            targetNamespace="http://service/" version="1.0">
            <xsd:element name="addCompte" type="tns:addCompte"/>
            <xsd:element name="consulterCompte"
                type="tns:consulterCompte"/>
        </xsd:schema>
    </wsdl:types>

```

Tester les méthodes avec Oxygen xml editor

Requête SOAP pour ajouter un compte

```
<SOAP-ENV:Envelope xmlns:SOAP-  
ENV="http://schemas.xmlsoap.org/soap/envelope/">  
    <SOAP-ENV:Header/>  
    <SOAP-ENV:Body>  
        <addCompte xmlns="http://service/">  
            <solde xmlns="">5000</solde>  
        </addCompte>  
    </SOAP-ENV:Body>  
</SOAP-ENV:Envelope>
```

Tester les méthodes avec Oxygen xml editor

Requête SOAP pour consulter tous les comptes

```
<SOAP-ENV:Envelope xmlns:SOAP-  
ENV="http://schemas.xmlsoap.org/soap/envelope/">  
  <SOAP-ENV:Header/>  
  <SOAP-ENV:Body>  
    <consulterComptes xmlns="http://service/" />  
  </SOAP-ENV:Body>  
</SOAP-ENV:Envelope>
```

Réponse SOAP

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">  
  <soap:Body>  
    <ns2:consulterComptesResponse xmlns:ns2="http://service/">  
      <return>  
        <code>1</code>  
        <dateCreation>2013-11-12T14:46:01Z</dateCreation>  
        <solde>3000.0</solde>  
      </return>  
      <return>  
        <code>2</code>  
        <dateCreation>2013-11-12T14:46:01Z</dateCreation>  
        <solde>29005.0</solde>  
      </return>  
    </ns2:consulterComptesResponse>  
  </soap:Body>  
</soap:Envelope>
```

Web Service

```
@Oneway  
public void verser(@WebParam(name="code")Long code,  
@WebParam(name="montant")double montant){  
    metier.verser(code, montant);  
}  
  
@Oneway  
public void retirer(@WebParam(name="code")Long code,  
@WebParam(name="montant")double montant){  
    metier.retirer(code, montant);  
}  
  
public Compte consulterCompte(@WebParam(name="code")Long code){  
    return metier.consulterCompte(code);  
}  
}
```

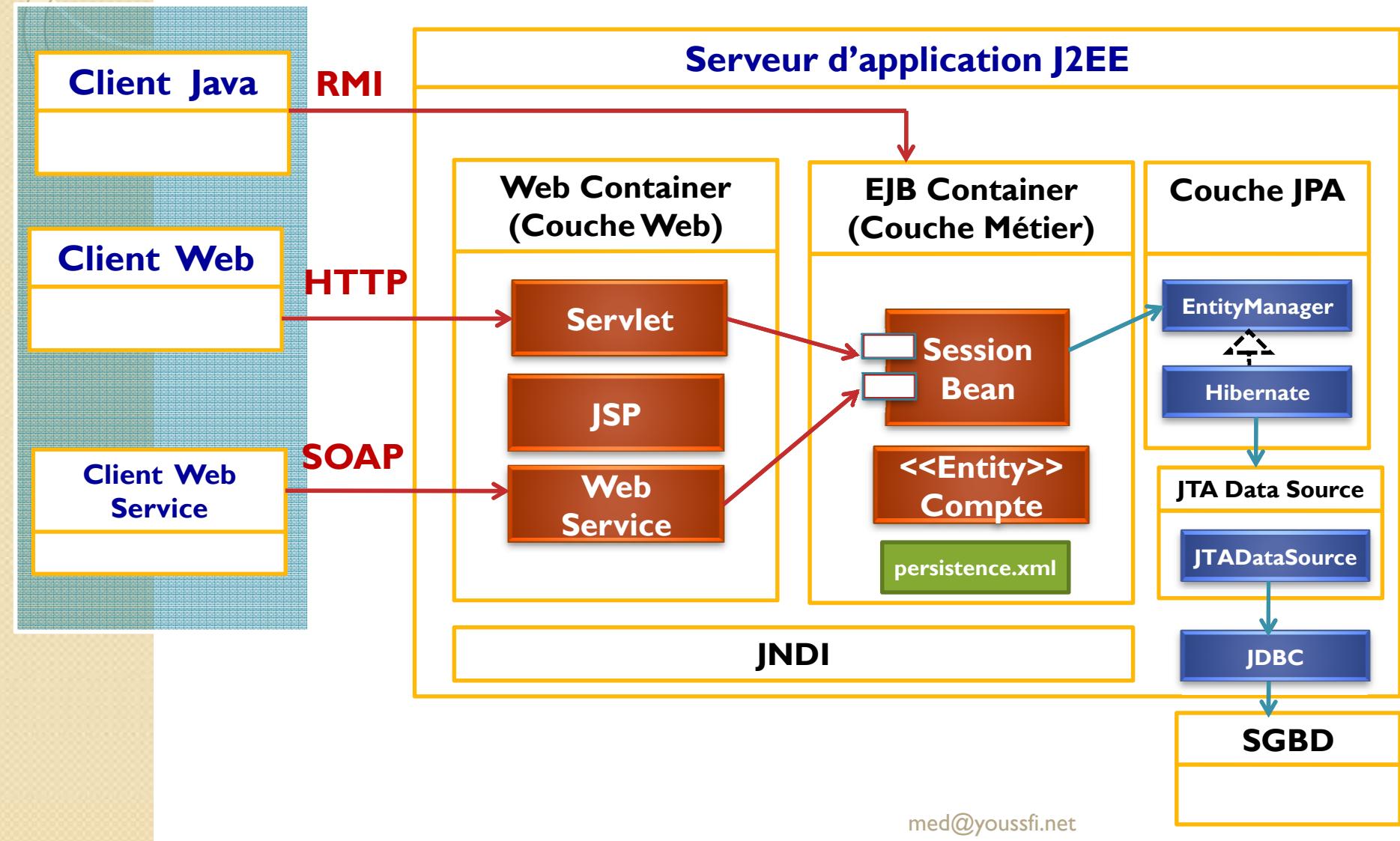


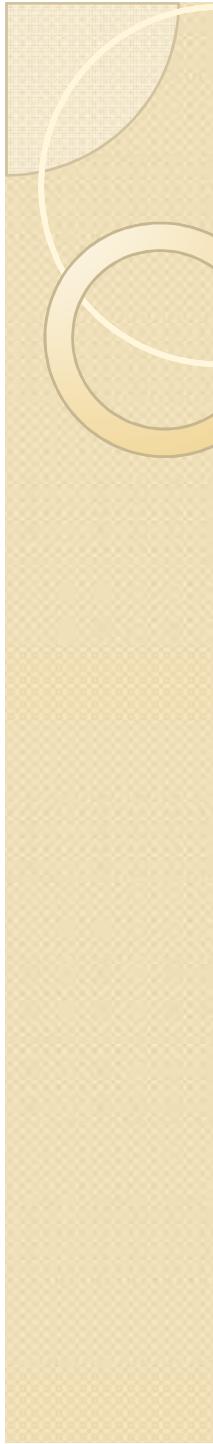
Persistiance des données

EJB Entity

med@youssf.net

Architecture





Entity Bean

- **Les Entity Beans**

- Représentent des objets persistant dont l'état est stocké dans une unité de persistance (base de donnée).
- Représentent les structures de données manipulée par l' application
- Sont des composants sérialisables qui peuvent être utilisés pour échanger des données entre applications distribuées.
- Gestion via JPA2 (*EntityManager*) et sessions beans
- Survivent au crash du serveur d'application



Entity Beans

- Les entités dans les spécifications de l'API Java Persistence permettent d'encapsuler les données d'une occurrence d'une ou plusieurs tables.
- Ce sont de simples **POJO (Plain Old Java Object)**.
- Un objet Java de type POJO mappé vers une table de la base de données grâce à des méta data via JPA est nommé bean entité (Entity bean).
- Un bean entité est tout d'abord un java bean ; une classe qui doit :
 - Implémenter l'interface `java.io.Serializable`
 - Posséder **un constructeur sans argument**
 - Posséder les Getters et Setters



Entity Beans

- En plus l'EJB Entity doit respecter les spécifications suivantes :
 - Être déclaré avec l'annotation `@javax.persistence.Entity`
 - Posséder au moins une propriété déclarée comme clé primaire avec l'annotation `@Id`
- Le bean entity est composé de propriétés qui seront mappées sur les champs de la table de la base de données sous-jacente.
- Chaque propriété encapsule les données d'un champ d'une table.
- Ces propriétés sont utilisables au travers de simples accesseurs (getter/setter).
- Une propriété particulière est **la clé primaire** qui sert d'identifiant unique dans la base de données mais aussi dans le POJO.
- Elle peut être de type primitif ou de type objet. La déclaration de cette clé primaire est obligatoire.



Mapping entre le bean Entity et la table

- La description du mapping entre le bean entité et la table peut être faite de deux façons :
 - Utiliser des annotations
 - Utiliser un fichier XML de mapping
- L'API propose plusieurs annotations pour supporter un mapping O/R assez complet.



Annotations du mapping entre le bean Entity et la table

- **@Table**
 - Préciser le nom de la table concernée par le mapping. Par défaut c'est le nom de la classe qui sera considérée
- **@Column**
 - Associer un champ de la colonne à la propriété. Par défaut c'est le nom de la propriété qui sera considérée.
- **@Id**
 - Associer un champ de la table à la propriété en tant que clé primaire
- **@GeneratedValue**
 - Demander la génération automatique de la clé primaire au besoin
- **@Basic**
 - Représenter la forme de mapping la plus simple. Cette annotation est utilisée par défaut
- **@Transient**
 - Demander de ne pas tenir compte du champ lors du mapping

Exemple d'EJB Entity : Compte.java

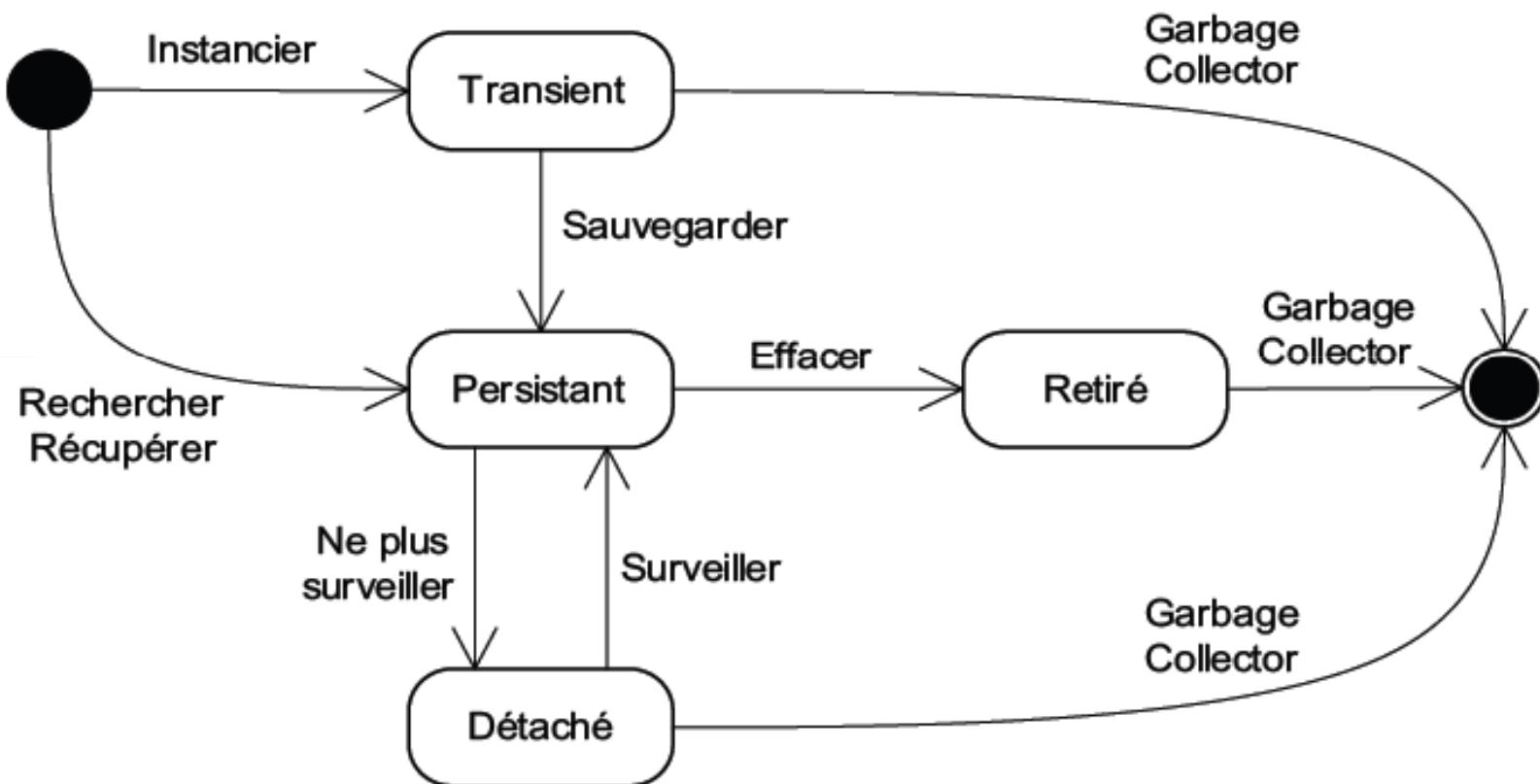
```
package metier.entities;
import java.io.Serializable;
import java.util.Date;
import javax.persistence.*;
@Entity
@Table(name="COMPTE")
public class Compte implements Serializable {
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
@Column(name="CODE")
private Long code;
@Column(name="SOLDE")
private double solde;
@Temporal(TemporalType.TIMESTAMP)
@Column(name="DATE_CREATION")
private Date dateCreation;
// Getters et Setters
// Constructeur sans paramètre
// Constructeur avec paramètres
}
```



Persistante Via JPA

- JPA 2 propose un modèle standard de persistante à l'aide des Entity beans
- Les outils qui assureront la persistante (Toplink, Hibernate, EclipseLink, etc.) sont intégrés au serveur d'application et devront être compatibles avec la norme JPA 2.
 - Java EE 6 repose à tous les niveaux sur de « l'injection de code » via des annotations de code
 - Souvent, on ne fera pas de « new », les variables seront créées/initialisées par injection de code.

Cycle de vie d'un EJB Entity





Objet Persistant

- Un objet *persistent* est un objet qui possède son image dans le datastore et dont la durée de vie est potentiellement infinie.
- Pour garantir que les modifications apportées à un objet sont rendues persistantes, c'est-à-dire sauvegardées, l'objet est surveillé par un « traqueur » d'instances persistantes.
- Ce rôle est joué par le gestionnaire d'entités.



Etat Transient

- Un objet *transient* est un objet qui n'a pas son image stockée dans le datastore.
- Il s'agit d'un objet « temporaire », qui meurt lorsqu'il n'est plus utilisé par personne. En Java, le garbage collector le ramasse lorsque aucun autre objet ne le référence.



Etat Détaché

- Un objet détaché est un objet qui possède son image dans le datastore mais qui échappe temporairement à la surveillance opérée par le gestionnaire d'entités.
- Pour que les modifications potentiellement apportées pendant cette phase de détachement soient enregistrées, il faut effectuer une opération manuelle pour *merger* cette instance au gestionnaire d'entités.



Etat Retiré

- Un objet *retiré* est un objet actuellement géré par le gestionnaire d'entités mais programmé pour ne plus être persistant.
- À la validation de l'unité de travail, un ordre SQL delete sera exécuté pour retirer son image du datastore.

Cycle de vie d'un EJB Entity

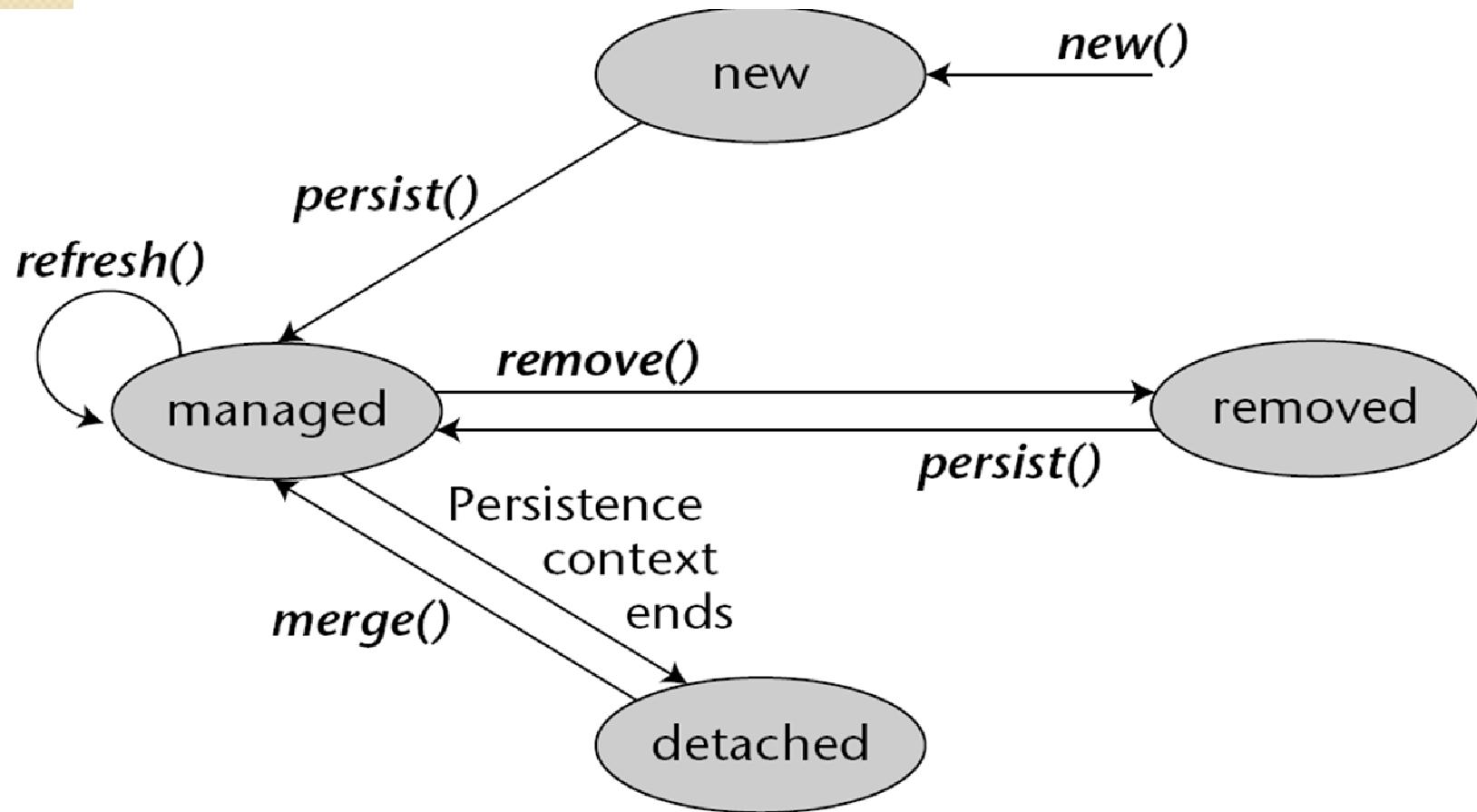


Figure 6.3 Entity life cycle.



Gestion des entités par EntityManager

- EntityManager est une interface définie dans JPA.
- Chaque framework ORM possède sa propre implémentation de cette interface.
- EntityManager définit les méthodes qui permettent de gérer le cycle de vie de la persistance des Entity.
 - La méthode **persist()** permet rendre une nouvelle instance d'un EJB Entity persistante. Ce qui permet de sauvegarder son état dans la base de données
 - La méthode **find()** permet de charger une entité sachant sa clé primaire.
 - La méthode **createQuery()** permet de créer une requête EJBQL qui permet de charger une liste d'entités selon des critères.
 - La méthode **remove()** permet de programmer une entité persistante pour la suppression.
 - La méthode **merge()** permet de rendre une entité détachée persistante.



Gérer la persistance des entités avec EntityManager fourni par JPA

```
package metier.session;

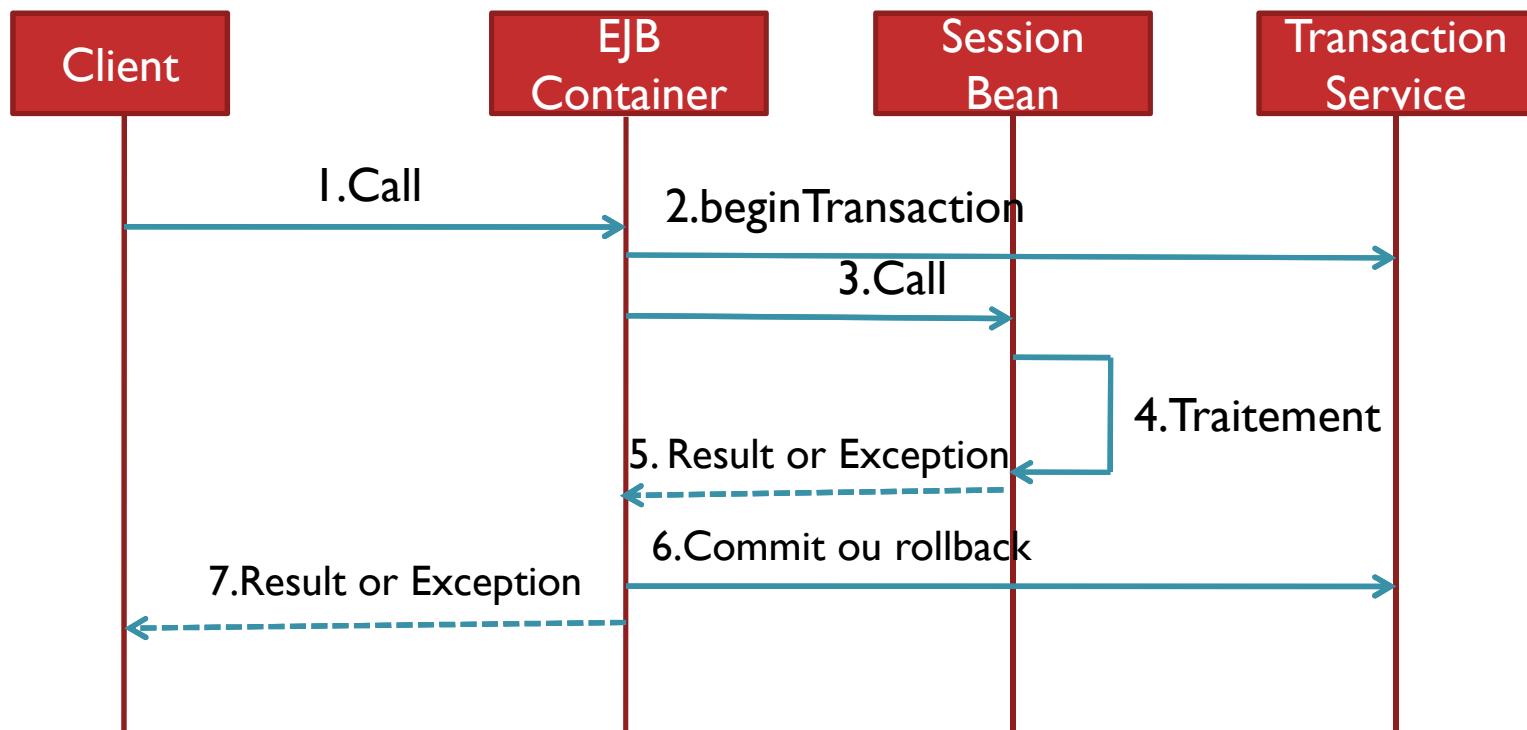
import java.util.List; import javax.ejb.Stateless;
import javax.persistence.*; import metier.entities.Compte;
@Stateless(name="BP3")
public class BanqueEjbJpaImpl implements IBanqueLocal,IBanqueRemote {
    @PersistenceContext(unitName="UP_BP")
    private EntityManager em;
    @Override
    public void addCompte(Compte c) {
        em.persist(c);
    }
    @Override
    public List<Compte> consulterComptes() {
        Query req=em.createQuery("select c from Compte c");
        return req.getResultList();
    }
}
```

Gérer la persistance des entités avec EntityManager

```
@Override  
public Compte consulterCompte(Long code) {  
    Compte cp=em.find(Compte.class,code);  
    if(cp==null) throw new RuntimeException("Ce compte n'existe pas");  
    return cp;  
}  
  
@Override  
public void verser(Long code, double montant) {  
    Compte cp=this.consulterCompte(code);  
    cp.setSolde(cp.getSolde()+montant);  
    em.persist(cp);  
}  
  
@Override  
public void retirer(Long code, double montant) {  
    Compte cp=this.consulterCompte(code);  
    if(cp.getSolde()<montant) throw new RuntimeException("Solde insuffisant");  
    cp.setSolde(cp.getSolde()-montant);  
}
```

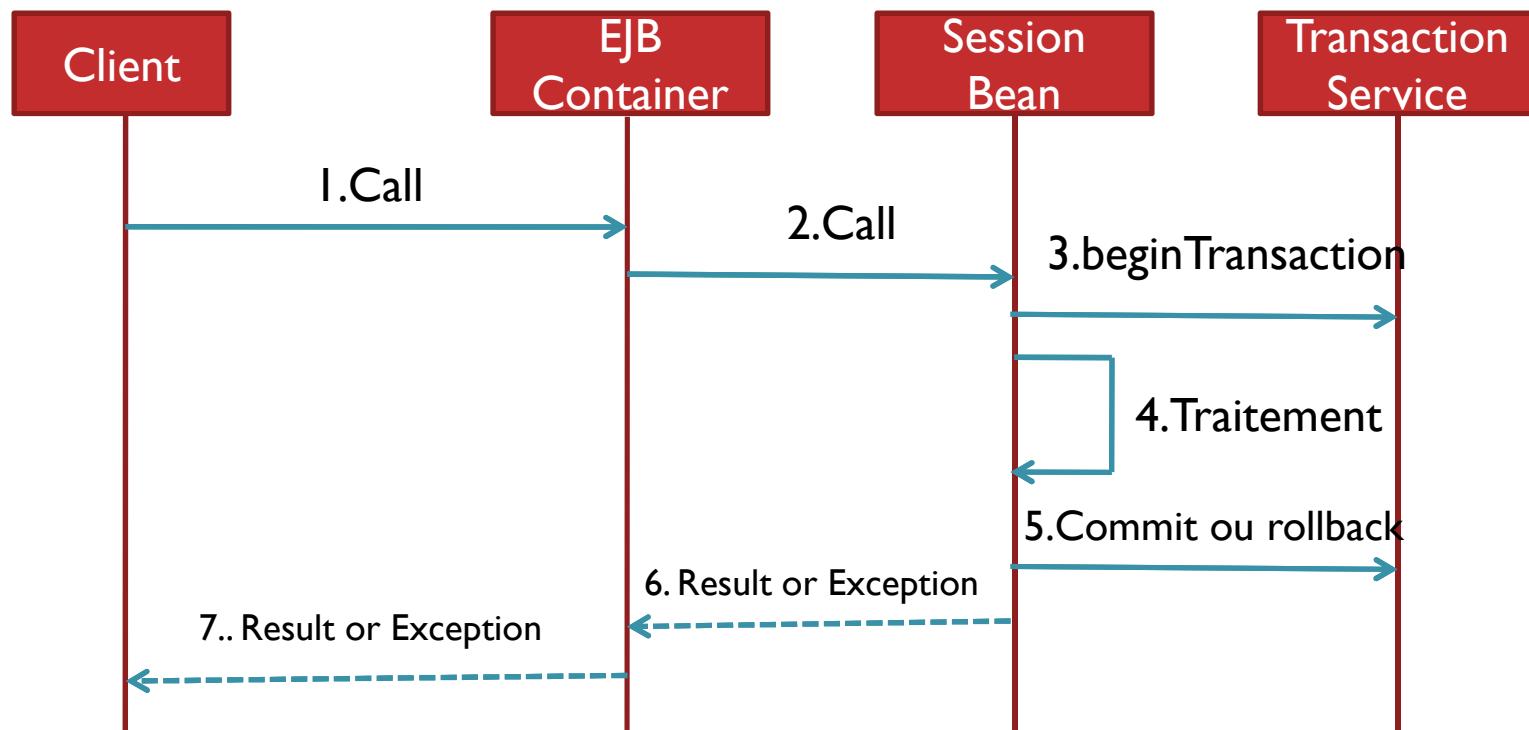
Gestion par le container

- C'est le mode par défaut
- Le bean est automatiquement enrôlé (*enrolled*) dans une transaction...
- Le container fait le travail...

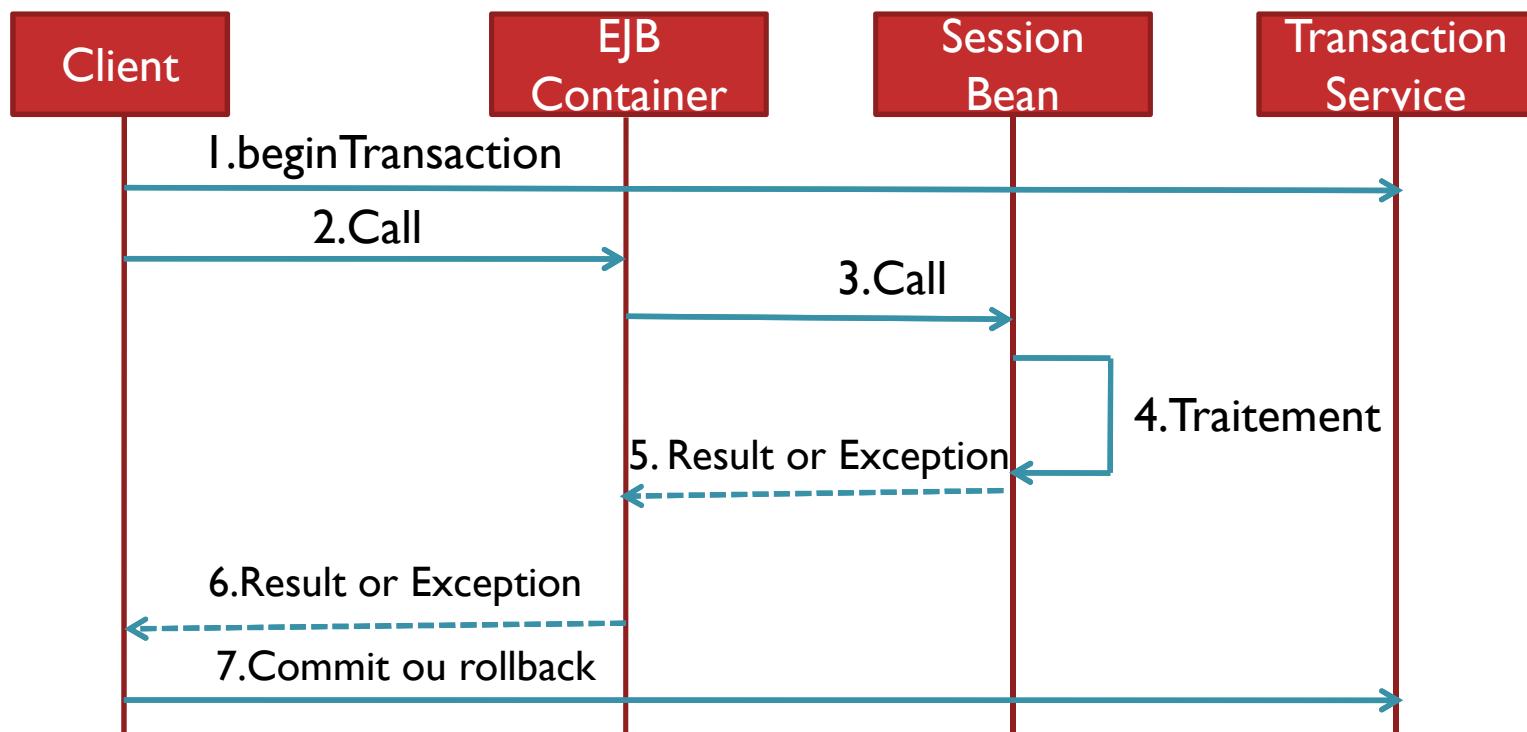


Gestion des transactions par programmation

- Responsable : le développeur de bean
- Il décide dans son code du *begin*, du *commit* et du *abort*

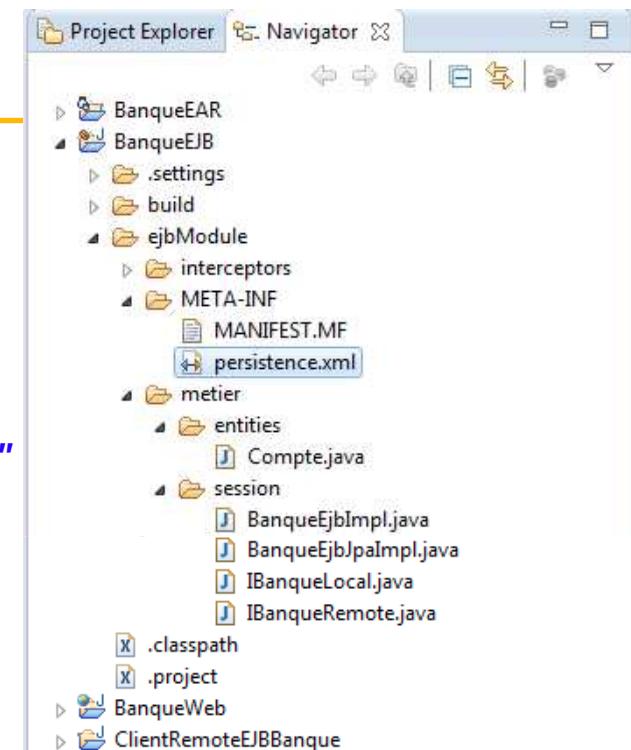


Transactions initiées par le client



Configurer l'unité de persistance : persistence.xml

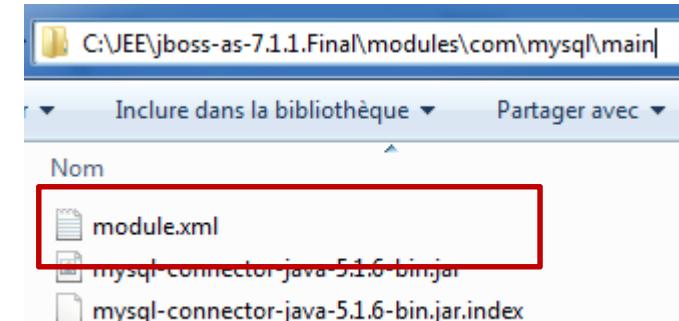
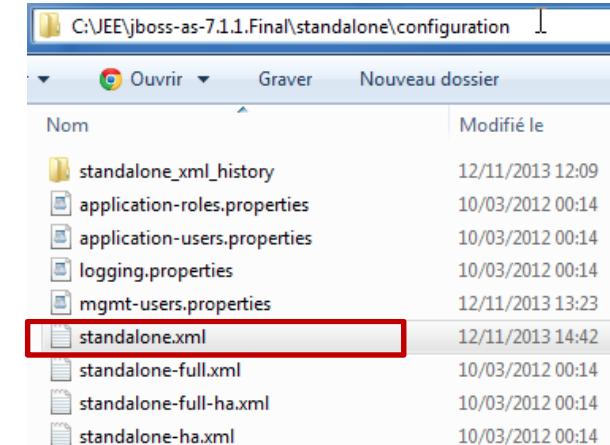
```
<?xml version="1.0" encoding="UTF-8"?>
<persistence
    xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
    version="1.0">
    <persistence-unit name="UP_BP">
        <jta-data-source>java:/dsBanque</jta-data-source>
        <properties>
            <property name="hibernate.hbm2ddl.auto"
value="update"/>
        </properties>
    </persistence-unit>
</persistence>
```



Déployer le datasource pour jboss 7

Standalone.xml

```
<datasources>
    <datasource jndi-name="java:/dsbanque" pool-name="dsBanque" enabled="true">
        <connection-url>jdbc:mysql://localhost:3306/db_banque</connection-url>
        <driver-class>com.mysql.jdbc.Driver</driver-class>
        <driver>mysql</driver>
        <security>
            <user-name>root</user-name>
            <password></password>
        </security>
        <validation>
    </datasource>
    <drivers>
        <driver name="h2" module="com.h2database.h2">
            <xa-datasource-class>org.h2.jdbcx.JdbcDataSource</xa-datasource-class>
        </driver>
        <driver name="mysql" module="com.mysql"/>
    </drivers>
</datasources>
```





Gérer les associations et l'héritage entre les entités

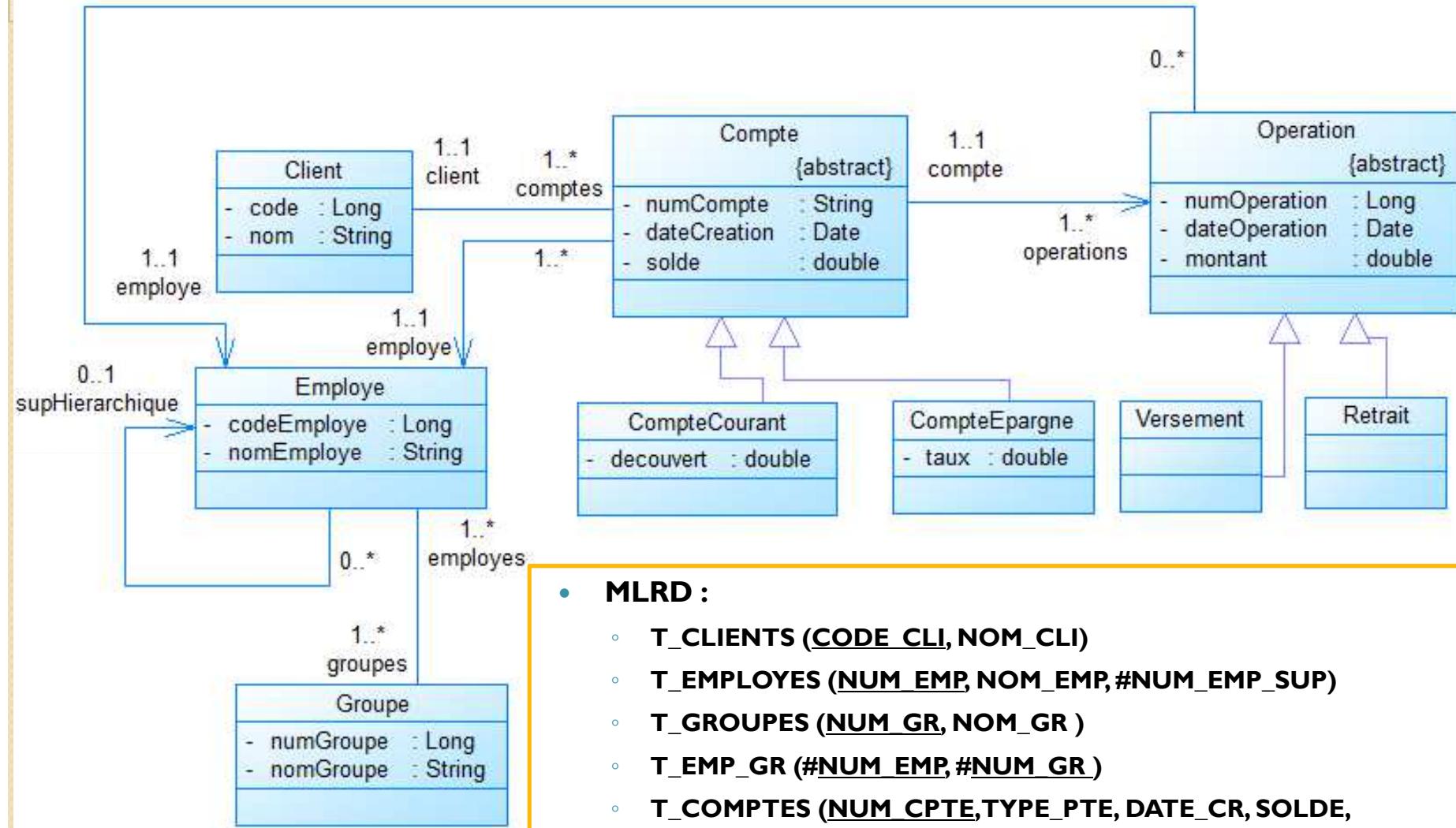
- Associations
 - **@OneToMany**
 - **@ManyToOne**
 - **@ManyToMany**
 - **@OneToOne**
- Héritage
 - Une table par hiérarchie
 - Une table pour chaque classe concrète
 - Une table pour la classe parente et une table pour chaque classe fille



Exemple de problème

- On souhaite créer une application qui permet de gérer des comptes bancaire.
 - Chaque compte est défini un numéro, un solde et une date de création
 - Un compte courant est un compte qui possède en plus un découvert
 - Un compte épargne est un compte qui possède en plus un taux d'intérêt.
 - Chaque compte appartient à un client et créé par un employé.
 - Chaque client est défini par son code et son nom
 - Un employé est défini par son code et son nom
 - Chaque employé possède un supérieur hiérarchique.
 - Chaque employé peut appartenir à plusieurs groupes
 - Chaque groupe, défini par un code est un nom, peut contenir plusieurs employés.
 - Chaque compte peut subir plusieurs opérations.
 - Il existe deux types d'opérations : Versement et Retrait
 - Chaque opération est effectuée par un employé.
 - Une opération est définie par un numéro, une date et un montant.

Diagramme de classes et MLDR



- **MLRD :**
- **T_CLIENTS (CODE_CLI, NOM_CLI)**
- **T_EMPLOYES (NUM_EMP, NOM_EMP, #NUM_EMP_SUP)**
- **T_GROUPES (NUM_GR, NOM_GR)**
- **T_EMP_GR (#NUM_EMP, #NUM_GR)**
- **T_COMPTES (NUM_CPT, TYPE_PTE, DATE_CR, SOLDE, #NUM_EMP, #CODE_CLI)**
- **T_OPERATIONS (NUM_OP, TYPE_OP, DATE_OP, MONTANT, #NUM_EMP, #NUM_CPT)**

Entity Client

```
package banque.metier;
import java.io.Serializable; import java.util.Collection;
import javax.persistence.*;
@Entity
@Table(name="CLIENTS")
public class Client implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="CODE_CLI")
    private Long codeClient;
    @Column(name="NOM_CLI")
    private String nomClient;

    @OneToMany(mappedBy="client",fetch=FetchType.LAZY,cascade=CascadeType.ALL)
    private Collection<Compte> comptes;
    // Getters et Setters
    // Constructeur sans param et avec params
}
```

Entity Employe

```
package banque.metier;
import java.io.Serializable; import java.util.Collection;
import javax.persistence.*;
@Entity
public class Employe implements Serializable{
    @Id
    @GeneratedValue
    private Long numEmploye;
    private String nomEmploye;
    private double salaire;
    @ManyToOne
    @JoinColumn(name="NUM_EMP_SUP")
    private Employe supHierarchique;
    @ManyToMany
    @JoinTable(name="EMP_GROUPES",joinColumns =
    @JoinColumn(name = "NUM_EMP"),
    inverseJoinColumns = @JoinColumn(name = "NUM_GROUPE"))
    private Collection<Groupe> groupes;
    // Getters et Setters
    // Constructeur sans param et avec params
}
```

Entity Groupe

```
package banque.metier;  
import java.io.Serializable; import  
java.util.Collection;  
import javax.persistence.*;  
@Entity  
public class Groupe implements Serializable {  
    @Id  
    @GeneratedValue  
    private Long numGroupe;  
    private String nomGroupe;  
    @ManyToMany(mappedBy="groupes")  
    private Collection<Employe> employes;  
    // Getters et Setters  
    // Constructeur sans param et avec params  
}
```

Entity Compte

```
package banque.metier;
import java.io.Serializable; import java.util.Collection;
import javax.persistence.*;
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="TYPE_CPTE",discriminatorType=DiscriminatorType.STRING,length=2)
public abstract class Compte implements Serializable {
    @Id
    private String numCompte;
    private Date dateCreation;
    private double solde;
    @ManyToOne
    @JoinColumn(name="CODE_CLI")
    private Client client;
    @ManyToOne
    @JoinColumn(name="NUM_EMP")
    private Employe employe;
    @OneToMany(mappedBy="compte")
    private Collection<Operation> operations;
    // Getters et Setters
    // Constructeur sans param et avec params
}
```

Entity CompteCourant

```
package banque.metier;
import java.io.Serializable;
import java.util.Collection;
import javax.persistence.*;
@Entity
@DiscriminatorValue("CC")
public class CompteCourant extends Compte{
    private double decouvert;
    // Getters et Setters
    // Constructeur sans param et avec params
}
```

Entity CompteEpargne

```
package banque.metier;
import java.io.Serializable;
import java.util.Collection;
import javax.persistence.*;
@Entity
@DiscriminatorValue("CE")
public class CompteEpargne extends Compte {
private double taux;
    // Getters et Setters
    // Constructeur sans param et avec params
}
```

Entity Operation

```
package banque.metier;
import java.io.Serializable;
import java.util.Collection;
import javax.persistence.*;
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="TYPE_OP",discriminatorType=DiscriminatorType.
    STRING,length=2)
public abstract class Operation implements Serializable {
    @Id
    @GeneratedValue
    private Long numOperation;
    private Date dateOperation;
    private double montant;
    @ManyToOne
    @JoinColumn(name="NUM_CPTE")
    private Compte compte;
    @ManyToOne
    @JoinColumn(name="NUM_EMP")
    private Employe employe;
    // Getters et Setters
    // Constructeur sans param et avec params
}
```

Entity Versement

```
package banque.metier;  
import java.io.Serializable;  
import java.util.Collection;  
import javax.persistence.*;  
  
@Entity  
@DiscriminatorValue("V")  
public class Versement extends Operation{  
    // Constructeur sans param et avec params  
}
```

Entity Retrait

```
package banque.metier;  
import java.io.Serializable;  
import java.util.Collection;  
import javax.persistence.*;  
  
@Entity  
@DiscriminatorValue("R")  
public class Retrait extends Operation {  
    // Constructeur sans param et avec params  
}
```

Interface Remote de l'EJB Session

```
package banque.metier.session;
import java.util.List;
import javax.ejb.Remote;
import banque.metier.*;
@Remote
public interface BanqueRemote {
    public void addClient(Client c);
    public void addEmploye(Employe e,Long numEmpSup);
    public void addGroupe(Groupe g);
    public void addEmployeToGroupe(Long idGroupe,Long idEmp);
    public void addCompte(Compte c,Long numCli,Long numEmp );
    public void addOperation(Operation op,String numCpte,Long numEmp);
    public Compte consulterCompte(String numCpte);
    public List<Client> consulterClientsParNom(String mc);
    public List<Client> consulterClients();
    public List<Groupe> consulterGroupes();
    public List<Employe> consulterEmployes();
    public List<Employe> consulterEmployesParGroupe(Long idG);
    public Employe consulterEmploye(Long idEmp);
}
```



Interface Local de l'EJB Session

```
package banque.metier.session;
import java.util.List;
import javax.ejb.Local;
import banque.metier.*;
@Local
public interface BanqueLocal {
    public void addClient(Client c);
    public void addEmploye(Employe e,Long numEmpSup);
    public void addGroupe(Groupe g);
    public void addEmployeToGroupe(Long idGroupe,Long idEmp);
    public void addCompte(Compte c,Long numCli,Long numEmp );
    public void addOperation(Operation op,String numCpte,Long numEmp);
    public Compte consulterCompte(String numCpte);
    public List<Client> consulterClientsParNom(String mc);
    public List<Client> consulterClients();
    public List<Groupe> consulterGroupes();
    public List<Employe> consulterEmployes();
    public List<Employe> consulterEmployesParGroupe(Long idG);
    public Employe consulterEmploye(Long idEmp);
}
```

EJB Session Stateless

```
package banque.metier.session;
import java.util.List;import javax.ejb.Stateless;
import javax.persistence.*; import banque.metier.*;
@Stateless(name="BANQUE")
public class BanqueEJBImpl implements BanqueRemote,BanqueLocal {
@PersistenceContext(name="UP_CATAL")
private EntityManager em;
@Override
public void addClient(Client c) {
    em.persist(c);
}
@Override
public void addEmploye(Employe e, Long numEmpSup) {
    Employe empSup;
    if(numEmpSup!=null){
        empSup=em.find(Employe.class, numEmpSup);
        e.setSupHierarchique(empSup);
    }
    em.persist(e);
}
```

EJB Session Stateless

```
@Override  
public void addGroupe(Groupe g) {  
    em.persist(g);  
}  
  
@Override  
public void addEmployeToGroupe(Long idGroupe, Long idEmp) {  
    Employe emp=em.find(Employe.class, idEmp);  
    Groupe g=em.find(Groupe.class, idGroupe);  
    emp.getGroupes().add(g);  
    g.getEmployes().add(emp);  
}  
  
@Override  
public void addCompte(Compte c, Long numCli, Long numEmp) {  
    Client cli=em.find(Client.class, numCli);  
    Employe e=em.find(Employe.class, numEmp);  
    c.setClient(cli);  
    c.setEmploye(e);  
    em.persist(c);  
}
```

EJB Session Stateless

```
@Override  
public void addOperation(Operation op, String numCpte, Long  
    numEmp) {  
    Compte c=em.find(Compte.class, numCpte);  
    Employe emp=em.find(Employe.class, numEmp);  
    op.setEmploye(emp);  
    op.setCompte(c);  
    em.persist(op);  
}  
  
@Override  
public Compte consulterCompte(String numCpte) {  
    Compte cpte=em.find(Compte.class, numCpte);  
    if(cpte==null) throw new RuntimeException("Compte "+numCpte+"  
        n'existe pas");  
    cpte.getOperations().size();  
    return cpte;  
}
```

EJB Session Stateless

```
@Override  
public List<Client> consulterClientsParNom(String mc) {  
    Query req=em.createQuery("select c from Client c where c.nom like  
        :mc");  
    req.setParameter("mc", "%" + mc + "%");  
    return req.getResultList();  
}  
  
@Override  
public List<Client> consulterClients() {  
    Query req=em.createQuery("select c from Client c");  
    return req.getResultList();  
}  
  
@Override  
public List<Groupe> consulterGroupes() {  
    Query req=em.createQuery("select g from Groupe g");  
    return req.getResultList();  
}
```

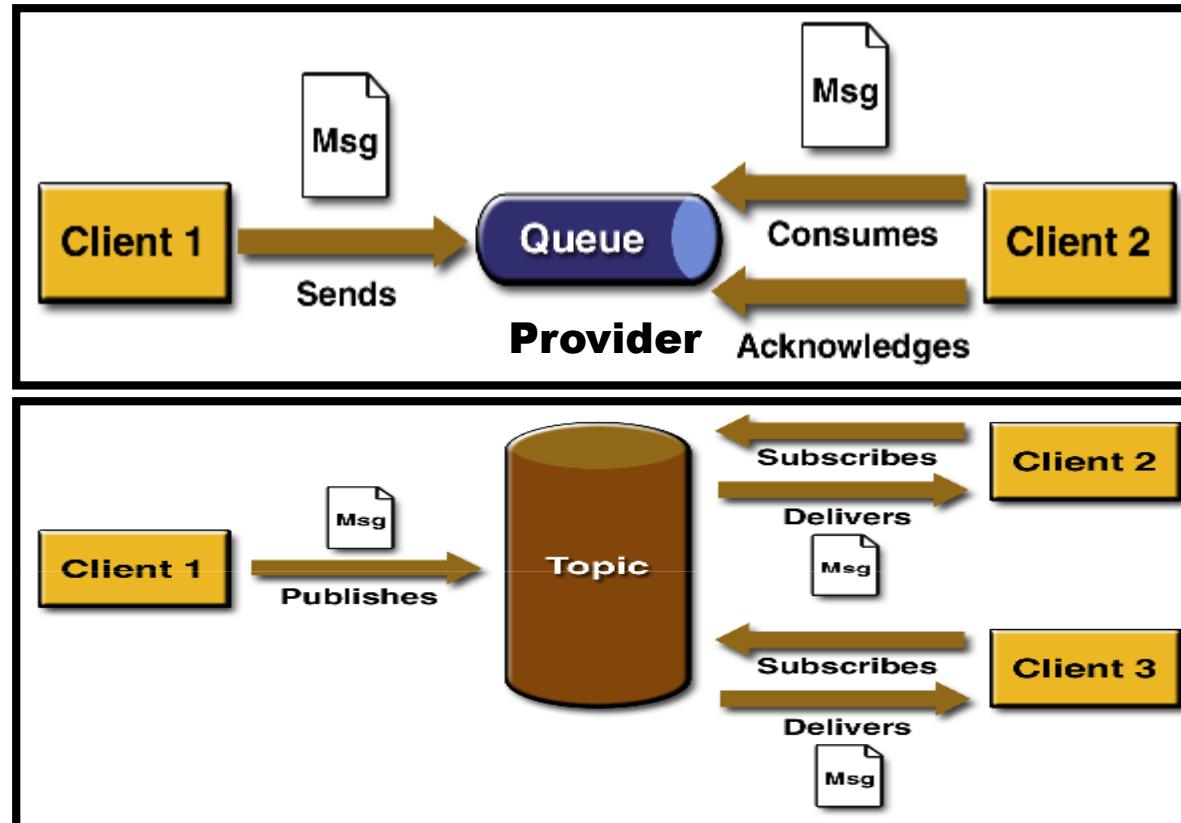
EJB Session Stateless

```
@Override  
public List<Employe> consulterEmployes() {  
    Query req=em.createQuery("select eg from Employe e");  
    return req.getResultList();  
}  
  
@Override  
public List<Employe> consulterEmployesParGroupe(Long idG) {  
    Query req=em.createQuery("select e from Employe e where  
        e.groupes.numGroupe=:x");  
    req.setParameter("x", idG);  
    return req.getResultList();  
}  
  
@Override  
public Employe consulterEmploye(Long idEmp) {  
    Employe e=em.find(Employe.class,idEmp);  
    if(e==null) throw new RuntimeException("Employe "+idEmp+"  
        n'existe pas");  
    return e;  
}  
}
```



JMS et MDB (Message Driven Bean)

JMS : Java Message Service



- **JMS**, ou **Java Message Service**, est une API d'échanges de messages pour permettre un dialogue entre applications via un fournisseur (**Provider**) de messages.
- L'application cliente envoie un message dans une **liste d'attente**, sans se soucier de la disponibilité de cette application.
- Le client a, de la part du fournisseur de messages, une garantie de **qualité de service** (certitude de remise au destinataire, délai de remise, etc.).



Définition des messages

- Les **messages** sont un moyen de **communication** entre les composants ou les applications.
- Les messages peuvent être envoyés par :
 - N'importe quel composant J2EE :
 - les applications clientes (Client Lourd)
 - les EJB,
 - les composants Web (Servlet, Web Service),
 - Une application J2EE interne / externe: interconnexion d'applications J2EE distribuées
 - Une autre application externe non J2EE : interconnexion d'applications J2EE et du système d'information.



L'API Java Message Service

- JMS est une API, spécifiée en 1998, pour la création, l'envoie et la réception des messages de façon :
 - **Asynchrone** : un client reçoit les messages dès qu'ils se connectent ou lorsque le client est disponible et sans avoir à demander si un message est disponible.
 - **Fiable** : le message est délivré une fois et une seule.



Quand utiliser JMS?

- JMS peut être utilisé lorsque :
 - les composants n'ont pas besoin de connaître les interfaces des autres composants,
 - mais uniquement transmettre un événement, une information.
- Exemple : une entreprise automobile
 - Le stock envoie un message à la production lorsque le nombre de véhicule répertorié est en dessous d'un seuil.
 - La production envoie un message aux unités pour assembler des parties d'une automobile.
 - Les unités envoient des messages à leurs stocks internes pour récupérer les parties.
 - Les unités envoient des messages aux gestionnaire de commandes pour commander les parties chez le fournisseur.
 - Le stock peut publier un catalogue pour la force de vente



Comment fonctionne JMS avec J2EE ?

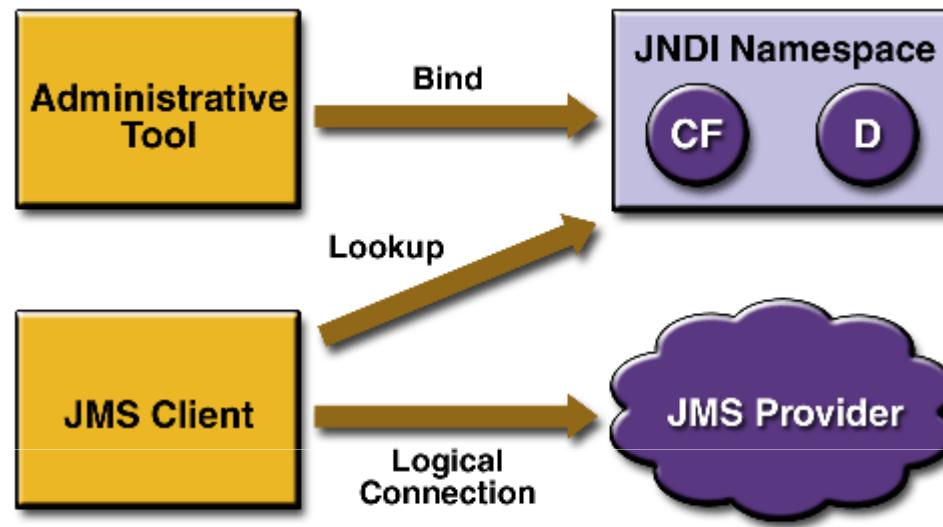
- JMS peut accéder aux **Messaging-oriented middleware (MOM)**, tels que
 - MQSeries d'IBM,
 - JBoss Messaging,
 - One Message Queue de Sun Microsystem, ...
- Elle fait partie intégrante de J2EE à partir de la version 1.3.
- Réception et envoie de messages de type :
 - **Synchrones** : applications clientes , EJBs, composants Web (sauf applets, hors spécification)
 - **Asynchrones** : applications clients, les beans orientés messages.
- Les composants messages participent aux **transactions distribuées** des autres composants.



L'architecture JMS

- Une application JMS est composée des parties suivantes :
 - Un *JMS provider* est un système de gestion de messages qui implémente les interfaces JMS. Une plateforme J2EE v.1.3 incluse un JMS provider.
 - Les *clients JMS* sont les composants Java, qui produisent et consomment des messages.
 - Les *Messages* sont les objets qui communiquent des informations entre les clients JMS.
 - Les *Administered objects* sont des objets JMS pré-configurés créés par un administrateur pour l'utilisation de clients. Les deux types d'objets sont les *destinations* et les *connection factories*.
 - Les *Native clients* sont des programmes qui utilisent une API native de gestion de message à la place de JMS.

L'architecture JMS



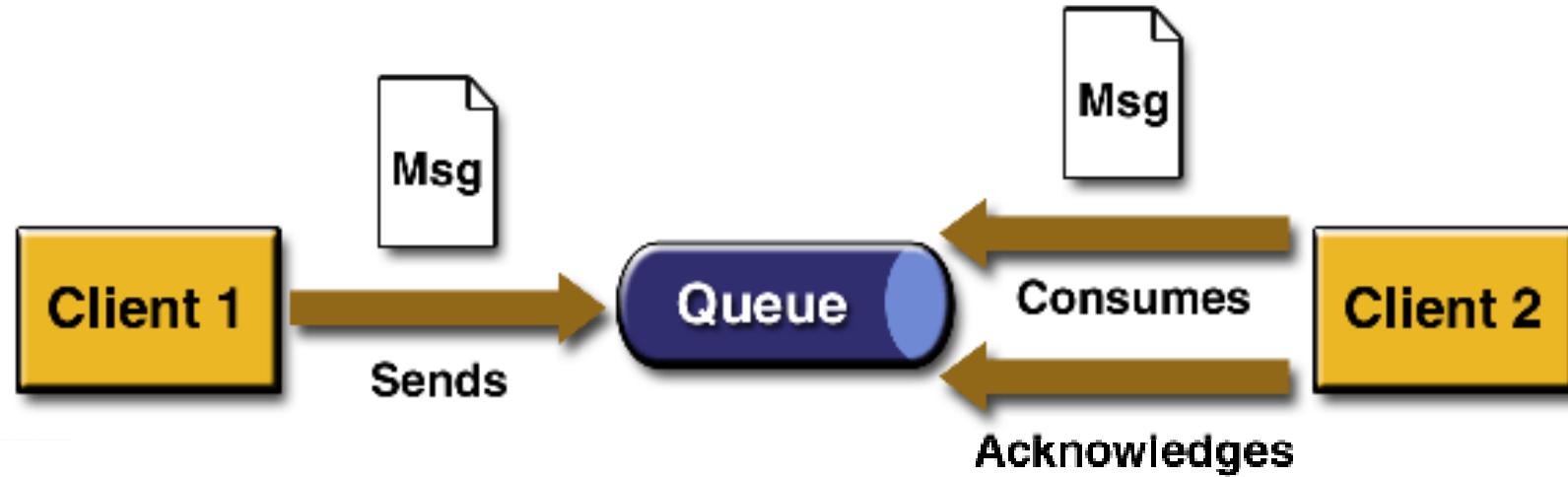
- Les *Administrative tools* permettent de lier les destinations et les connections factories avec JNDI.
- Un *client JMS* peut alors rechercher les objets administrés dans JNDI et établir une connexion logique à travers le JMS Provider.



Les protocoles de communication

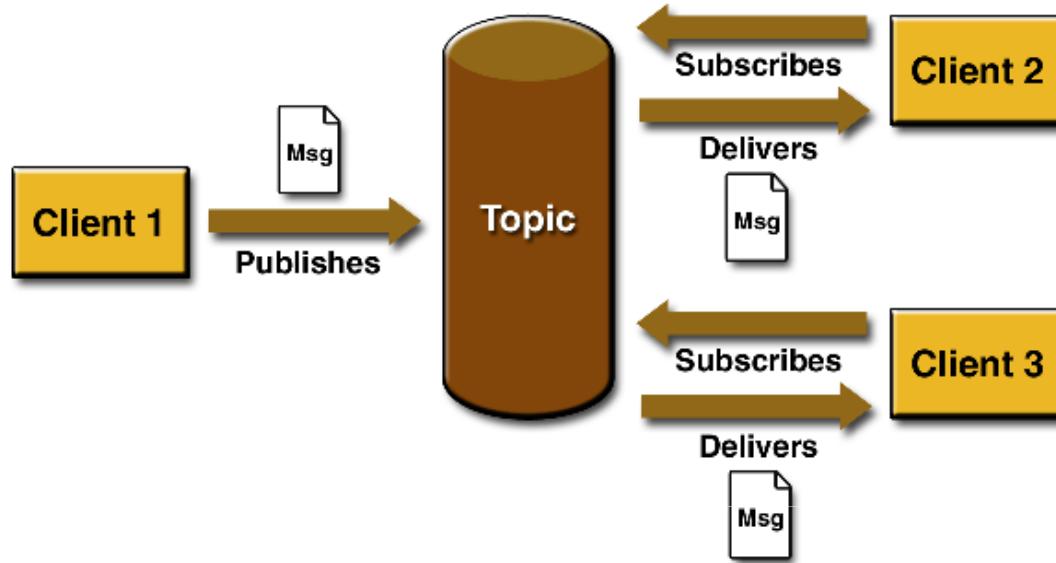
- Avant l'arrivée de JMS, les produits proposaient une gestion de messages selon deux types de protocoles :
 - un protocole **point-à-point (Queue)**,
 - un protocole **publier/souscrire (Topic)**
- Les JMS Provider compatible J2EE 1.3 doivent obligatoirement implémenter ces deux protocoles

Le protocole point-à-point



- Ce protocole est basé sur le concept de **queue de messages**, d'envoyeurs et de receveurs.
- Chaque message est adressé à une queue spécifique et les clients consomment leurs messages.
- Ce protocole doit être utilisé pour s'assurer **qu'un seul client consommera le message**.

Le protocole publier/souscrire (Topic)



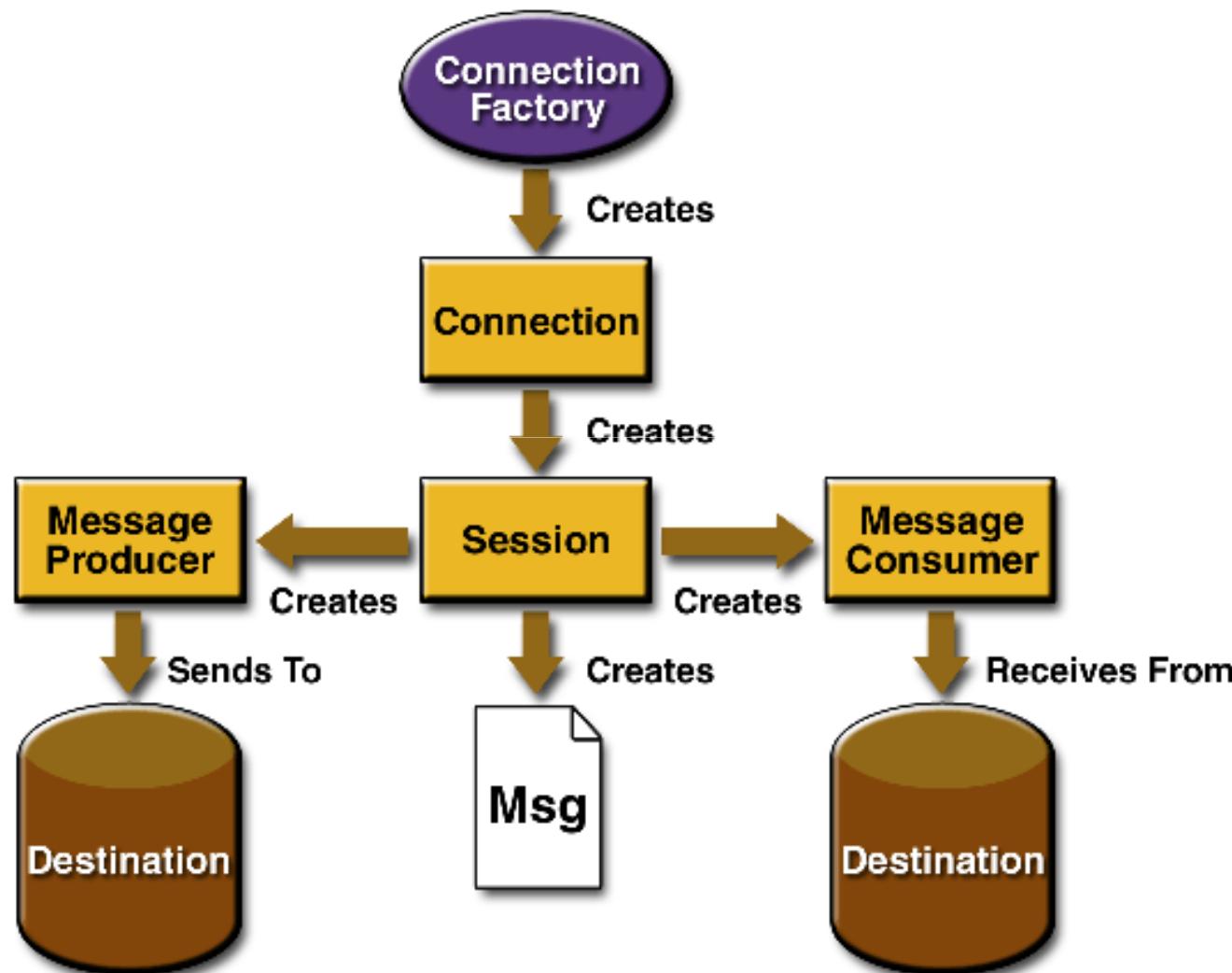
- Les clients sont anonymes et adressent les messages.
- Le système distribue le message à l'ensemble des souscripteurs puis l'efface.
- Ce protocole est efficace lorsqu'un message doit être envoyé à zéro, un ou plusieurs clients.
- Il existe deux types de souscription : temporaire et durable.
 - Dans le premier cas, les consommateurs **reçoivent les messages tant qu'ils sont connectés au sujet**.
 - Dans le cas d'une souscription durable, on oblige le fournisseur à **enregistrer les messages** lors d'une déconnexion, et à les envoyer lors de la nouvelle connexion du consommateur.



La consommation de messages

- Un client consomme les messages de façon :
 - **Synchrone** : en appelant la méthode `receive`, qui est bloquante jusqu'à ce qu'un message soit délivré, ou qu'un délai d'attente soit expiré.
 - **Asynchrone** : un client peut s'enregistrer auprès d'un écouteur de message (`listener`). Lorsqu'un message arrive, JMS le délivre en invoquant la méthode `onMessage` du listener, lequel agit sur le message.

Le modèle de programmation JMS





Mise en place des composants JMS

- Il existe un certain nombre de composants qui s'occupe de la gestion globale de JMS, et de permettre ainsi une communication asynchrone entre applications clientes.
 - **ConnectionFactory** et **Destination** :
 - Pour travailler avec JMS, la première étape consiste d'abord à se connecter au fournisseur JMS. Pour cela, nous devons
 - récupérer un objet ConnectionFactory via JNDI qui rend ainsi la connexion possible avec le fournisseur.
 - Une ConnectionFactory fournit une connexion JMS au service de routage de message.
 - L'autre élément à récupérer est la destination.
 - Les destinations (Destination) sont des objets qui véhiculent les messages.
 - JMS comporte deux types de destination, comme nous venons de le découvrir, les Queue et les Topic.

Création du context JNDI

- Voici le code java coté client JMS :

```
Context ctx = new InitialContext();
ConnectionFactory cf =
    (ConnectionFactory)ctx.lookup("ConnectionFactory");
Destination destination = (Destination)ctx.lookup("queue/MaFile");
```

- Voici une autre écriture en utilisant les annotations dans un session bean.

```
@Resource(mappedName="ConnectionFactory")
private ConnectionFactory fabrique;
@Resource(mappedName="queue/MaFile")
private Destination destination;
```

- Pour obtenir une ConnectionFactory, une Queue, ou un Topic, il faut les rechercher par leur nom dans l'annuaire JNDI ou utiliser l'injection. Cela suppose donc que ces ressources soient préalablement mis en œuvre et qu'elles soient recensées au travers du service d'annuaire JNDI.



Mise en place des composants JMS

- Les propriétés pour la création du contexte **JNDI** sont dépendantes du fournisseur utilisé, et à ce titre, nous devons utiliser la même démarche que pour la communication par les beans session.
- Il est donc nécessaire d'initialiser ce contexte avec tous les bons paramètres requis.
- Le plus facile est de placer ces différents paramètres, dans un fichier de configuration dont le nom est bien précis (**jndi.properties**) et qui doit être placé dans le répertoire racine du projet.

jndi.properties

- Pour GlassFish

- **java.naming.factory.initial=com.sun.enterprise.naming.SerialInitContextFactory**
- **java.naming.factory.url.pkgs=com.sun.enterprise.naming**
- **java.naming.factory.state=com.sun.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl**
- **org.omg.CORBA.ORBInitialHost=portable**
- **org.omg.CORBA.ORBInitialPort=3700**

- Pour Jboss 7

- **java.naming.provider.url=tcp://localhost:3035**
- **java.naming.factory.initial=org.exolab.jms.jndi.InitialContextFactory**
- **java.naming.security.principal=admin**
- **java.naming.security.credentials=password**



Connection et Session

- L'objet ConnectionFactory permet de créer une connexion avec le fournisseur JMS.
- Une fois la connexion créée, elle est ensuite utilisée pour créer une session.

```
Connection connexion = cf.createConnection();
Session session = connexion.createSession(true, 0);
```

- La méthode createSession() prend deux paramètres :
 - Si vous désirez pouvoir travailler avec plusieurs messages pour une même session, vous devez autoriser le mode transactionnel dans le premier argument de la fonction.
 - Le deuxième est utile pour savoir si vous désirez qu'un accusé réception soit renvoyé afin de préciser que message est bien arrivé à sa destination. Dans l'affirmative, vous devez utiliser la constante **Session.AUTO_ACKNOWLEDGE** dont la valeur est **0**.



MessageProducer et MessageConsumer

- La dernière étape nous sert à préciser le sens du transfert du message, est-ce pour envoyer ou est-ce pour recevoir ? Deux objets correspondent à ces deux situations, respectivement
 - **MessageProducer** et **MessageConsumer** :

```
MessageProducer envoi = session.createProducer(destination);
```

```
MessageConsumer rec = session.createConsumer(destination);
```

- Chacune des méthodes de l'objet session prend en paramètre la destination sur laquelle l'objet est connecté.

Message

- Dans JMS, un message est un objet Java qui doit implémenter l'interface `javax.jms.Message`.
- Il est composé de trois parties :
 - L'en-tête (**header**) : qui se compose des informations de **destination, d'expiration, de priorité, date d'envoi**, etc.
 - Les propriétés (**properties**) : qui représentent les caractéristiques fonctionnelles du message.
 - Le corps du message (**body**) : qui contient les données à transporter.

Entête du message

- Liste des propriétés de l'entête d'une message JMS :

Nom	Description
JMSMessageID	identifiant unique de message
JMSCorrelationID	Utilisé pour associer de façon applicative deux messages par leur identifiant.
JMSDeliveryMode	Il existe deux modes d'envoi : persistent (le message est délivré une et une seule fois au destinataire, c'est-à-dire que même au cas de panne du fournisseur, le message sera délivré) et non persistant (le message peut ne pas être délivré en cas de panne puisqu'il n'est pas rendu persistant).
JMSDestination	File d'attente destinataire du message.
JMSExpiration	Date d'expiration du message.
JMSPriority	Priorité du message. Cet attribut indique la priorité de façon croissante à partir de 0 (les messages de niveau 9 ont plus de priorité que les messages de niveau 0).
JMSRedelivered	Booléen qui signifie que le message a été redélivré au destinataire.
JMSReplyTo	File d'attente de réponse du message.
JMSTimestamp	L'heure d'envoi du message est affecté automatiquement par le fournisseur.



Les propriétés d'un message JMS

- Cette section du message est optionnelle et agit comme une extension des champs d'en-tête.
- Les propriétés d'un message JMS sont des couples (nom, valeur), où la valeur est un type de base du langage Java (entiers, chaînes de caractères, booléens, etc.).
- L'interface javax.jms.Message définit des accesseurs pour manipuler ces valeurs.
- Ces données sont généralement positionnées par le client avant l'envoi d'un message et, comme nous le verrons par la suite, peuvent être utilisées pour filtrer les messages.

Corps d'un message JMS

- Le corps du message, bien qu'optionnel, est la zone qui contient les données.
- Ces données sont formatées selon le type du message qui est défini par les interfaces suivantes (qui héritent toutes de javax.jms.Message) :

Interface	Description
javax.jms.BytesMessage	Pour les messages sous forme de flux d'octets.
javax.jms.TextMessage	Echange de données de type texte.
javax.jms.ObjectMessage	Messages composés d'objets Java sérialisés.
javax.jms.MapMessage	Echange de données sous la forme clé/valeur. La clé doit être une String et la valeur de type primitif.
javax.jms.StreamMessage	Echange de données en provenance d'un flux.

javax.jms.BytesMessage

- **Exemple :**

```
BytesMessage message=session.createBytesMessage();
message.writeInt(15);
message.writeDouble(-6.78);
message.writeBoolean(true);
envoi.send(message);
```



javax.jms.TextMessage

- **Exemple :**

```
TextMessage message2=session.createTextMessage();
```

```
message2.setText("Bienvenue");
```

```
envoi.send(message2);
```

javax.jms.ObjectMessage

- **Exemple :**

- **Une classe Serializable Personne :**

```
public class Personne implements Serializable {  
}
```

- Code JMS pour envoyer un ObjectMessage qui contient deux objet de type Personne :

```
Personne p1=new Personne();  
  
Personne p2=new Personne();  
  
ObjectMessage message3=session.createObjectMessage();  
message3.setObject(p1);  
message3.setObject(p2);  
envoi.send(message3);
```



javax.jms.MapMessage

- Ce type de message, MapMessage, permet d'envoyer et de recevoir des informations suivant le système clé/valeur. Ainsi, nous retrouvons les mêmes méthodes que pour le type BytesMessage, mais à chaque fois, nous devons préciser la clé sous forme de chaîne de caractères. Par ailleurs, les méthodes sont plutôt des accesseurs getXxx() et setXxx() :

- **Exemple :**

```
MapMessage message4=session.createMapMessage();  
message4.setInt("code", 210);  
message4.setString("fonction", "ingenieur");  
message4.setDouble("salaire",22000.00);  
envoi.send(message4);
```

javax.jms.StreamMessage

- Ce type de message, StreamMessage, est vu cette fois-ci comme un flux.
- Il ressemble beaucoup d'ailleurs au types DataOutputStream et DataInputStream
- **Exemple :**

```
StreamMessage message5=session.createStreamMessage();
message5.writeInt(15);
message5.writeDouble(-6.78);
message5.writeBoolean(true);
envoi.send(message4);
```



Comment envoyer un message

- Pour envoyer un message, nous avons déjà tout recensé. Nous avons juste à revoir l'ensemble des éléments à mettre en œuvre.
 - Tout d'abord, la fabrique de connexion (ConnectionFactory) et la destination (Destination) doivent être connues par le client JMS.
 - Une fois la référence de la ConnectionFactory obtenue, on se connecte au provider (fournisseur) JMS via l'objet Connection.
 - A partir de cette connexion, nous devons obtenir une session (Session).
 - A partir de cette session, nous devons créer un MessageProducer qui va permettre d'envoyer des messages auprès d'une destination.
 - La session permet également de créer le message suivant le type choisi.



Comment envoyer un message

- Context ctx = new InitialContext();
- ConnectionFactory fabrique = (ConnectionFactory)ctx.lookup("ConnectionFactory");
- Destination destination = (Destination)ctx.lookup("queue/MaFile");
- Connection connexion = fabrique.createConnection();
- Session session = connexion.createSession(true, Session.AUTO_ACKNOWLEDGE);
- MessageProducer envoi = session.createProducer(destination);
- MessageConsumer reception = session.createConsumer(destination);
- BytesMessage message=session.createBytesMessage();
- message.writeInt(15);
- message.writeDouble(-6.78);
- message.writeBoolean(true);
- envoi.send(message);



Comment recevoir un message

- Le consommateur du message est le client capable d'être à l'écoute d'une file d'attente (ou d'un sujet), et de traiter les messages à leur réception.
- En effet, le client doit être constamment à l'écoute (`listener`) et, à l'arrivée d'un nouveau message, il doit pouvoir le traiter.
- Pour cela, l'application doit appeler la méthode `onMessage()` de l'interface `javax.jms.MessageListener`.
- Celle-ci est très spécialisée et permet ainsi la réception asynchrone des messages.
- Charge au développeur d'implémenter cette interface pour réaliser le traitement adéquat lors de la réception d'un message.



Comment recevoir un message

- Voici la procédure à suivre :
 - Tout d'abord, comme l'envoi d'un message, la fabrique de connexion (ConnectionFactory) et la destination (Destination) doivent être connues par le client JMS.
 - Une fois la référence de la ConnectionFactory obtenue, le consommateur doit se connecter au provider (fournisseur) JMS via l'objet Connection.
 - A partir de cette connexion, nous devons obtenir une session (Session).
 - A partir de la session, on crée un MessageConsumer qui va permettre de consommer les messages.
 - Pour ce faire, nous associons un listener MessageListener pour traiter les messages de façon asynchrone. Ainsi, à chaque réception d'un nouveau message, la méthode onMessage() est automatiquement invoquée et peut effectuer le traitement désiré.
 - Attention : à ce stade, il ne faut surtout pas oublier de démarrer la connexion avec la méthode start() sinon aucun message ne sera reçu.

Comment recevoir un message

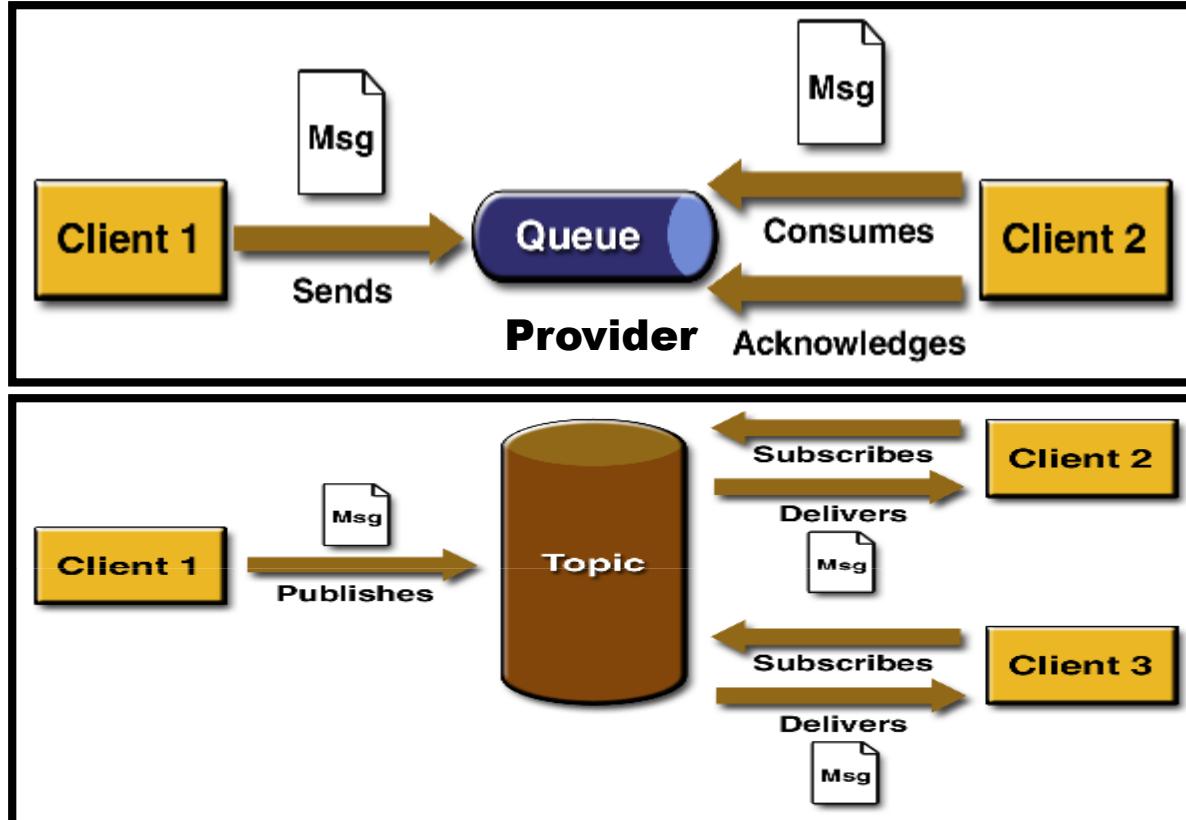
```
import javax.jms.*;import javax.naming.*;
public class Reception implements MessageListener {
    public Reception() throws Exception {
        Context ctx = new InitialContext();
        ConnectionFactory fournisseur = (ConnectionFactory)
ctx.lookup("ConnectionFactory");
        Destination destination =
(Destination)ctx.lookup("queue/maFile");
        Connection connexion = fournisseur.createConnection();
        Session session = connexion.createSession(false,
Session.AUTO_ACKNOWLEDGE);
        MessageConsumer reception =
session.createConsumer(destination);
        reception.setMessageListener(this);
        connexion.start();
    }
}
```



Comment recevoir un message

```
public void onMessage(Message arg) {  
    try {  
        TextMessage message = (TextMessage) arg;  
        System.out.println(message.getText());  
    }  
    catch (Exception ex) { }  
}  
  
public static void main(String[] args) throws Exception {  
    new Reception();  
}  
}
```

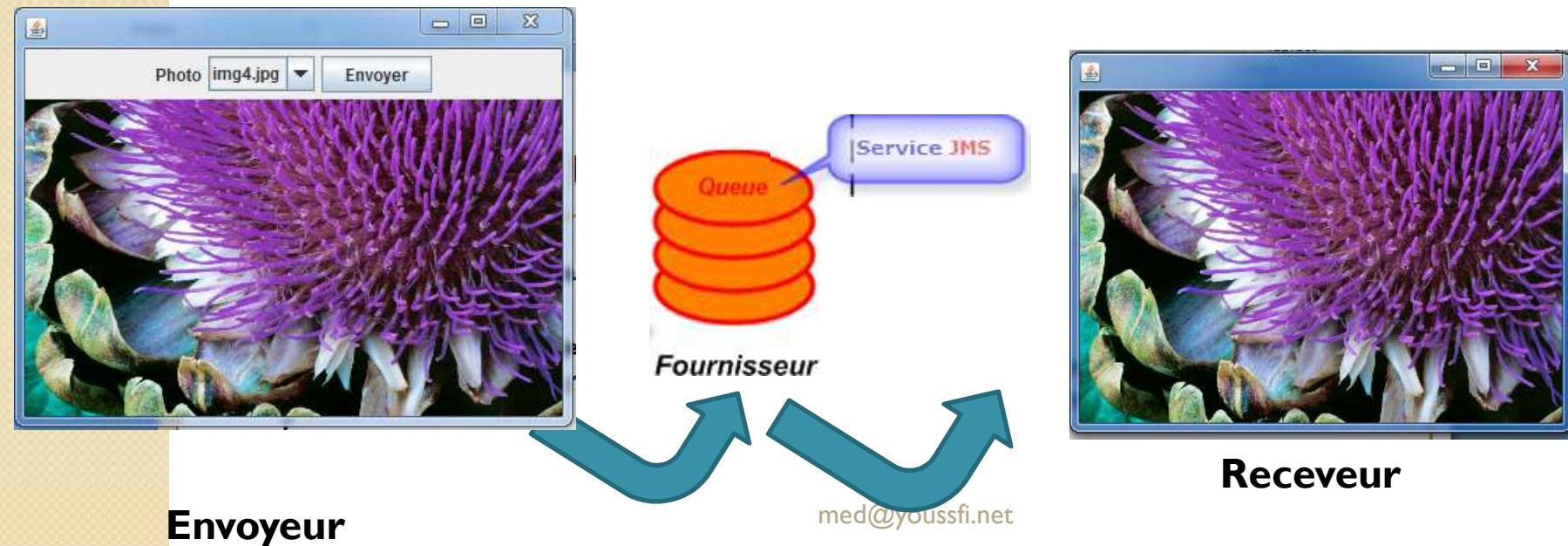
JMS : Java Message Service



- **JMS**, ou **Java Message Service**, est une API d'échanges de messages pour permettre un dialogue entre applications via un fournisseur (**Provider**) de messages.
- L'application cliente envoie un message dans une **liste d'attente**, sans se soucier de la disponibilité de cette application.
- Le client a, de la part du fournisseur de messages, une garantie de **qualité de service** (certitude de remise au destinataire, délai de remise, etc.).

Deux applications clientes en communication JMS

- Nous allons mettre en oeuvre nos nouvelles connaissances sur une communication asynchrone entre deux applications clientes au travers du service JMS. Pour illustrer ces différents mécanismes d'échange, nous allons créer deux applications clientes fenêtrées.
 - La première doit récupérer des photos présentes sur le poste local et les afficher ensuite dans la zone principale de la fenêtre. Ainsi, vous avez la possibilité de choisir la photo qui vous plaît afin de l'envoyer au service de messagerie asynchrone JMS.
 - La deuxième, sur un autre poste client, en attente d'éventuels messages venant du même fournisseur de messagerie, affiche la photo envoyée par la première application.

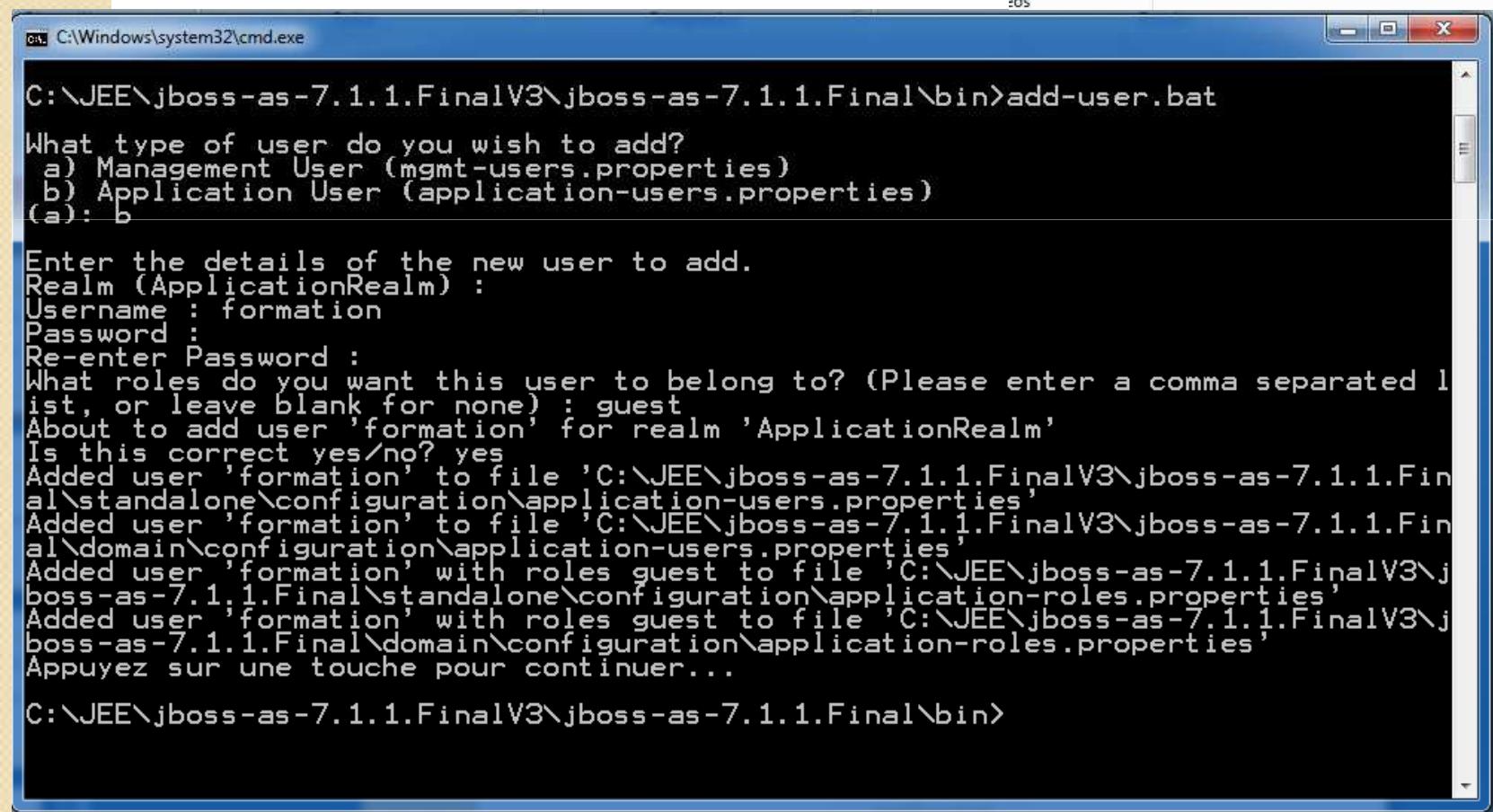


Configurer la destination : standalone-full-ha.xml

```
<jms-destinations>
    <jms-queue name="testQueue">
        <entry name="queue/test"/>
        <entry name="java:jboss/exported/jms/queue/test"/>
    </jms-queue>
    <jms-topic name="testTopic">
        <entry name="topic/test"/>
        <entry name="java:jboss/exported/jms/topic/test"/>
    </jms-topic>
</jms-destinations>
```

Création d'un utilisateur

- Type d'utilisateur : Application User
- Realm (ApplicationRealm):
- Username : formation
- Password : BP



C:\Windows\system32\cmd.exe

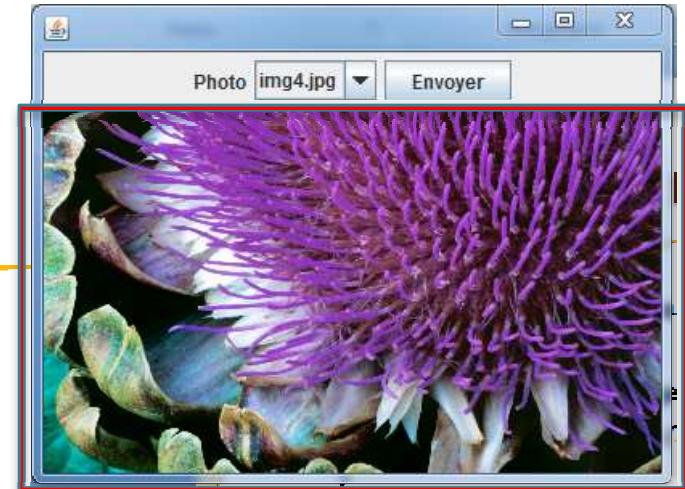
```
C:\JEE\jboss-as-7.1.1.FinalV3\jboss-as-7.1.1.Final\bin>add-user.bat
What type of user do you wish to add?
a) Management User (mgmt-users.properties)
b) Application User (application-users.properties)
(a): b

Enter the details of the new user to add.
Realm (ApplicationRealm) :
Username : formation
Password :
Re-enter Password :
What roles do you want this user to belong to? (Please enter a comma separated list, or leave blank for none) : guest
About to add user 'formation' for realm 'ApplicationRealm'
Is this correct yes/no? yes
Added user 'formation' to file 'C:\JEE\jboss-as-7.1.1.FinalV3\jboss-as-7.1.1.Final\standalone\configuration\application-users.properties'
Added user 'formation' to file 'C:\JEE\jboss-as-7.1.1.FinalV3\jboss-as-7.1.1.Final\domain\configuration\application-users.properties'
Added user 'formation' with roles guest to file 'C:\JEE\jboss-as-7.1.1.FinalV3\jboss-as-7.1.1.Final\standalone\configuration\application-roles.properties'
Added user 'formation' with roles guest to file 'C:\JEE\jboss-as-7.1.1.FinalV3\jboss-as-7.1.1.Final\domain\configuration\application-roles.properties'
Appuyez sur une touche pour continuer...

C:\JEE\jboss-as-7.1.1.FinalV3\jboss-as-7.1.1.Final\bin>
```

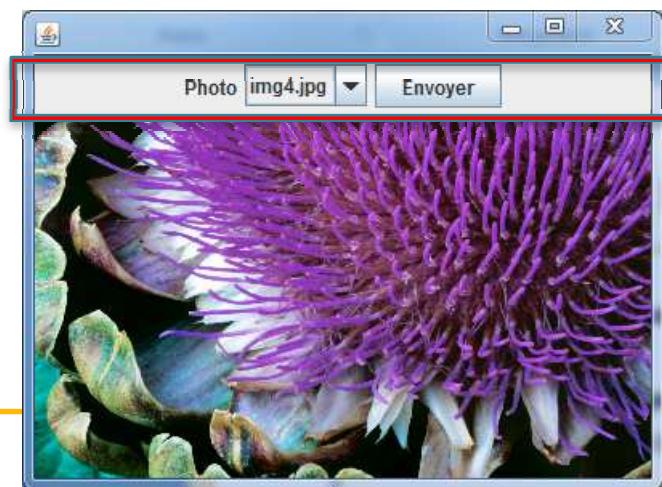
PanelPhoto.java

```
import java.awt.Graphics;
import java.awt.image.BufferedImage;
import javax.swing.JPanel;
public class PanelPhoto extends JPanel {
    private BufferedImage bufferedImage;
    @Override
    public void paint(Graphics g) {
        g.drawImage(bufferedImage, 0, 0, this.getWidth(), this.getHeight(),
        null);
    }
    public BufferedImage getBufferedImage() {
        return bufferedImage;
    }
    public void setBufferedImage(BufferedImage bufferedImage) {
        this.bufferedImage = bufferedImage;
    }
}
```



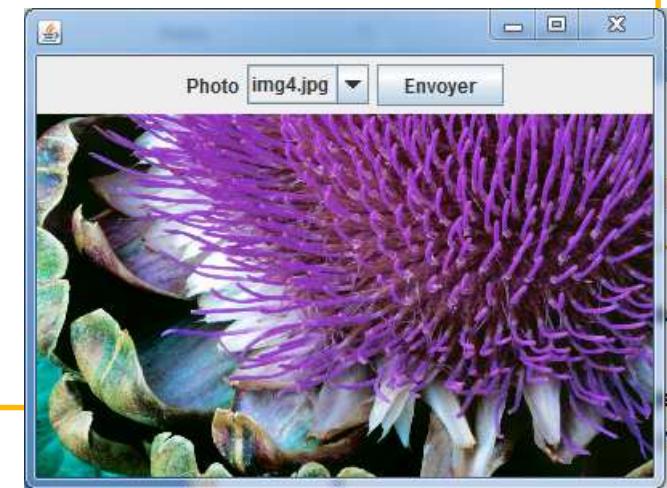
EnvoyerPhoto.java

```
import java.awt.*; import java.awt.event.*;
import java.awt.image.*; import java.io.*;
import java.util.Properties; import javax.imageio.ImageIO;
import javax.jms.*; import javax.naming.*; import javax.swing.*;
public class EnvoyerPhoto extends JFrame {
    private JLabel jLabelPhoto=new JLabel("Photo");
    private JComboBox<String> jComboBoxPhotos;
    private JButton jButtonEnvoyer=new JButton("Envoyer");
    private PanelPhoto panelPhoto=new PanelPhoto();
    private Connection conn;
    private Destination destination;
```



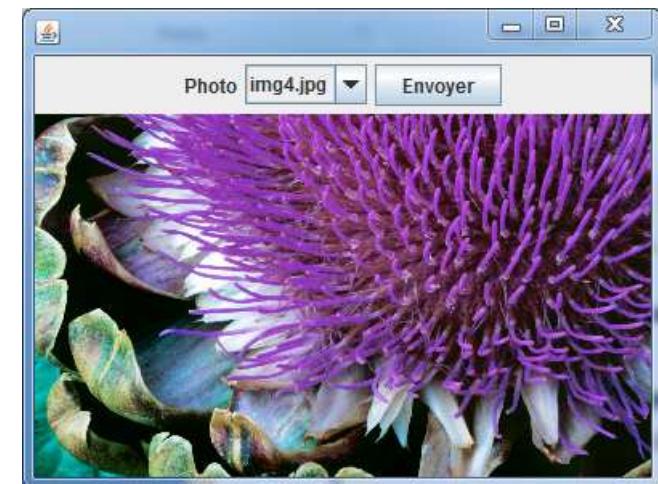
EnvoyerPhoto.java

```
public EnvoyerPhoto() {  
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);  
    this.setLayout(new BorderLayout());  
    JPanel jPanelN=new JPanel();  
    File f=new File("photos");  
    String[] photos=f.list();  
    jComboBoxPhotos=new JComboBox<String>(photos);  
    jPanelN.setLayout(new FlowLayout());  
    jPanelN.add(jLabelPhoto);jPanelN.add(jComboBoxPhotos);  
    jPanelN.add(jButtonEnvoyer);  
    this.add(jPanelN,BorderLayout.NORTH);  
    this.add(panelPhoto,BorderLayout.CENTER);  
    this.setBounds(10, 10,400, 300);  
    this.setVisible(true);
```



EnvoyerPhoto.java

```
init();  
  
jComboBoxPhotos.addActionListener(new ActionListener() {  
  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        try {  
            String photo=(String) jComboBoxPhotos.getSelectedItem();  
            File file=new File("photos/"+photo);  
            BufferedImage bi=ImageIO.read(file);  
            panelPhoto.setBufferedImage(bi);  
            panelPhoto.repaint();  
        } catch (Exception e1) {  
            e1.printStackTrace();  
        }  
    }  
});
```



EnvoyerPhoto.java

```
jButtonEnvoyer.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        try {  
            Session session=conn.createSession(false, QueueSession.AUTO_ACKNOWLEDGE);  
            MessageProducer producer=session.createProducer(destination);  
            File f=new File("photos/"+(String)jComboBoxPhotos.getSelectedItem());  
            FileInputStream fis=new FileInputStream(f);  
            byte[] data=new byte[(int)f.length()];  
            fis.read(data);  
            StreamMessage message=session.createStreamMessage();  
            message.writeString((String)jComboBoxPhotos.getSelectedItem());  
            message.writeInt(data.length);  
            message.writeBytes(data);  
            producer.send(message);  
            //session.commit(); //conn.close();  
        } catch (Exception ex) { ex.printStackTrace(); }  
    }  
});  
}
```



EnvoyerPhoto.java

```
public void init(){
    try {
        Properties p=new Properties();
        p.put(Context.INITIAL_CONTEXT_FACTORY,
               "org.jboss.naming.remote.client.InitialContextFactory");
        p.put(Context.PROVIDER_URL, "remote://localhost:4447");
        p.put(Context.SECURITY_PRINCIPAL, "formation");
        p.put(Context.SECURITY_CREDENTIALS, "BP");
        Context ctx = new InitialContext(p);
        ConnectionFactory factory=(ConnectionFactory)
            ctx.lookup("jms/RemoteConnectionFactory");
        conn=factory.createConnection("formation", "BP");
        destination=(Destination) ctx.lookup("jms/queue/test");
        conn.start();
    } catch (NamingException e) { e.printStackTrace();}
    catch (JMSException e) { e.printStackTrace();}
}
public static void main(String[] args) { new EnvoyerPhoto(); }
}
```

ReceveurPhoto.java

```
import java.awt.*;
import java.awt.image.*;
import java.io.*;
import java.util.Properties;
import javax.imageio.ImageIO;
import javax.jms.*;
import javax.naming.*;
import javax.swing.JFrame;
public class ReceveurJMS extends JFrame {
    private JPanelPhoto jPanelPhoto=new JPanelPhoto();
    public ReceveurJMS() {
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        this.setLayout(new BorderLayout());
        this.add(jPanelPhoto,BorderLayout.CENTER);
        this.setBounds(10, 10, 400, 400);
        this.setVisible(true);
```



ReceveurPhoto.java

```
try {  
    Properties p=new Properties();  
    p.put(Context.INITIAL_CONTEXT_FACTORY,  
        "org.jboss.naming.remote.client.InitialContextFactory");  
    p.put(Context.PROVIDER_URL, "remote://localhost:4447");  
    p.put(Context.SECURITY_PRINCIPAL, "formation");  
    p.put(Context.SECURITY_CREDENTIALS, "BP");  
    Context ctx = new InitialContext(p);  
    ConnectionFactory  
        factory=(ConnectionFactory)ctx.lookup("jms/RemoteConnectionFactory");  
    Connection conn=factory.createConnection("formation", "BP");  
    Destination destination=(Destination) ctx.lookup("jms/queue/test");  
    Session session=conn.createSession(false, QueueSession.AUTO_ACKNOWLEDGE);  
    //MessageProducer producer=session.createProducer(destination);  
    MessageConsumer consumer=session.createConsumer(destination);  
    consumer.setMessageListener(new MessageListener() {
```

ReceveurPhoto.java

```
@Override  
public void onMessage(Message message) {  
    try {  
        StreamMessage m=(StreamMessage) message;  
        String nomPhoto=m.readString();  
        int length=m.readInt();  
        byte[] data=new byte[length];  
        m.readBytes(data);  
        ByteArrayInputStream bais=new ByteArrayInputStream(data);  
        BufferedImage bi=ImageIO.read(bais);  
        jPanelPhoto.setBufferedImage(bi);  
        jPanelPhoto.repaint();  
    } catch (Exception e) { e.printStackTrace(); }  
});  
conn.start();  
} catch (Exception e) { e.printStackTrace(); }  
}  
public static void main(String[] args) { new ReceveurJMS(); }  
}
```





EJB

MESSAGE DRIVEN BEAN

MDB



EJB Message Driven Bean

- Un Message Driven Bean ou MDB est un EJB qui se comporte comme un listener JMS, c'est-à-dire qui reçoit des messages et les traite de manière asynchrone.
- Les MDB se rapprochent des EJB stateless car ils sont, eux aussi, sans état.
- Ils s'exécutent à l'intérieur du conteneur EJB qui assure donc le multithreading, la sécurité ou la gestion des transactions.
- Les MDB sont à l'écoute (listener) d'une file d'attente et se réveillent à chaque arrivée de messages.
- En fait, il faut garder à l'esprit que c'est le conteneur qui est le véritable listener JMS et qu'il délègue au MDB le traitement du message, et plus particulièrement à la méthode `onMessage()` que nous avons déjà utilisée.
- Comme les autres EJB, le MDB peut accéder à tout type de ressources : EJB, JDBC, JavaMail, etc.

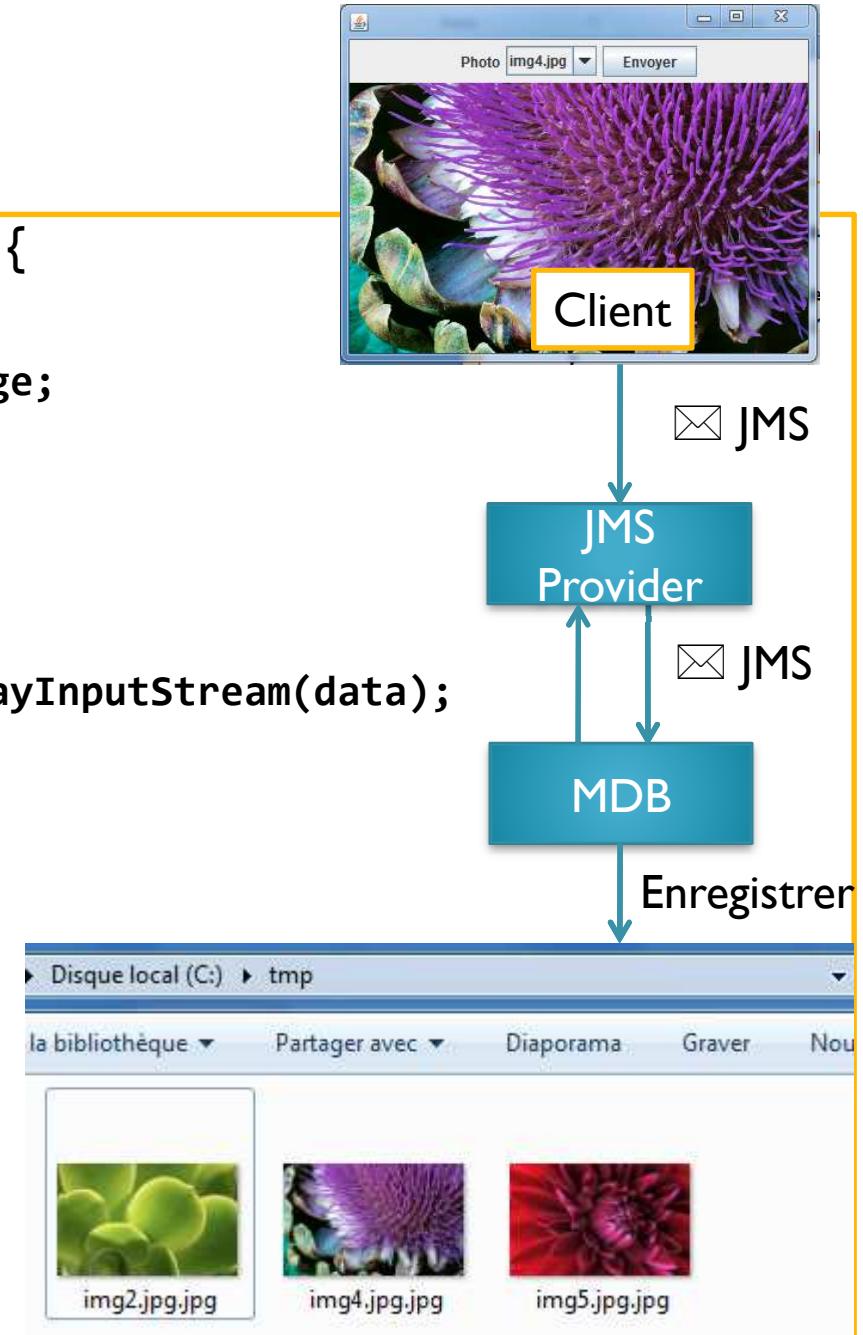


Structure d'un MDB

```
package metier.mdb;  
import java.awt.image.*; import java.io.*; import javax.imageio.ImageIO;  
import javax.jms.*; import javax.ejb.ActivationConfigProperty;  
import javax.ejb.MessageDriven;  
  
@MessageDriven( activationConfig={  
    @ActivationConfigProperty(  
        propertyName="destinationType",  
        propertyValue="javax.jms.Topic"),  
    @ActivationConfigProperty(  
        propertyName="destination",  
        propertyValue="topic/test"  
)})  
public class BanqueMDB implements MessageListener {  
    @Override  
    public void onMessage(Message message) {  
        // Traitement du message  
    }  
}
```

Exemple de traitement du message

```
public void onMessage(Message m) {  
    try {  
        StreamMessage m=(StreamMessage) message;  
        String nomPhoto=m.readString();  
        int length=m.readInt();  
        byte[] data=new byte[length];  
        m.readBytes(data);  
        ByteArrayInputStream bais=new ByteArrayInputStream(data);  
        BufferedImage bi=ImageIO.read(bais);  
        File f=new File("c:/tmp/"+nomPhoto);  
        ImageIO.write(bi, "jpeg", f);  
    } catch (JMSException e) {  
        e.printStackTrace();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```



L'annotation **@MessageDriven** et son attribut **activationConfig**

- Pour Le modèle Queue :

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



L'annotation **@MessageDriven** et son attribut **activationConfig**

- Pour Le modèle Topic :

- Le modèle de messagerie de type "abonnement" oblige les applications clientes à être connectées au sujet (Topic) pour recevoir les messages de celui-ci.
- Si un problème survient, les clients déconnectés perdent les messages émis durant leur déconnexion.
- Cependant, le mode Topic offre la possibilité d'utiliser un abonnement durable. L'intérêt est donc de pouvoir recevoir les messages émis depuis la dernière déconnexion.
- L'utilisation de ce genre d'abonnement doit être précisée au niveau des propriétés du MDB. La propriété à utiliser est **subscriptionDurability**. Les valeurs prises par celle-ci sont : **Durable** ou **NonDurable**. Par défaut, une souscription est **NonDurable** :

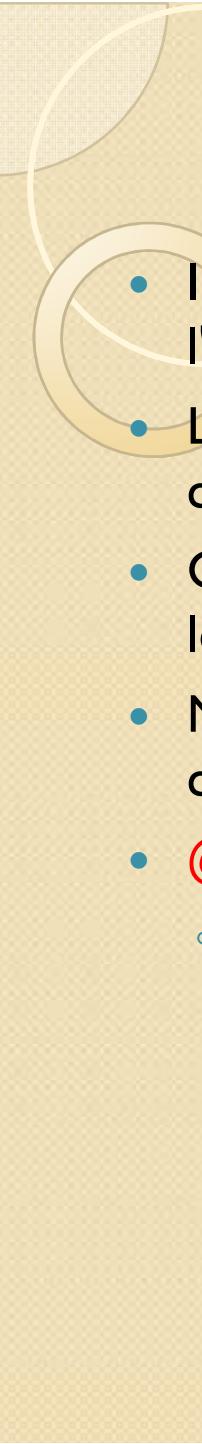
L'annotation @MessageDriven et son attribut activationConfig

- Pour Le modèle Topic :

```
@MessageDriven(  
    activationConfig={  
        @ActivationConfigProperty(  
            propertyName="destinationType",  
            propertyValue="javax.jms.Topic"),  
        @ActivationConfigProperty(  
            propertyName="destination",  
            propertyValue="jms/TestTopic"),  
        @ActivationConfigProperty(  
            propertyName="subscriptionDurability",  
            propertyValue="Durable")  
    })  
  
public class VisionneuseMDB implements MessageListener {  
}
```

L'annotation **@MessageDriven** et son attribut **activationConfig**

- Lorsque la destination est de type Queue, le principe d'abonnement durable n'a aucun sens.
- Par nature, pour ce genre de destination, le facteur "durable" n'a pas d'importance car les messages sont automatiquement stockés et doivent être consommés par un client unique.



Sélecteur de Messages

- Il est possible de préciser certains critères permettant de ne pas recevoir l'ensemble des messages d'une destination.
- Le sélecteur de messages utilise les propriétés du message en tant que critère dans les expressions conditionnelles.
- Ces conditions utilisent des expressions booléennes afin de déterminer les messages à recevoir.
- Nous pouvons par exemple récupérer uniquement les messages qui correspondent à l'utilisation de la méthode afficher par le client :
- **@MessageDriven(**
 - **activationConfig={**
 - **@ActivationConfigProperty(**
 - **propertyName="messageSelector",**
 - **propertyValue="commande='afficher'") }**

Sélecteur de Messages

- Ces sélecteurs se basent sur les propriétés des messages.
- Celles-ci se situent dans l'en-tête du message, donc dépendant du contenu, et sont assignées par le créateur du message.
- Tous les types de message intègrent les méthodes de lecture et d'écriture de propriétés.
- En effet, ces méthodes sont décrites dans la super-interface javax.jms.Message.
- Les types de propriétés se basent sur les primitives Java :boolean, int, short, char...

```
StreamMessage message = session.createStreamMessage();
message.setStringProperty("commande", "afficher");
```

Sélecteur de Messages

- Le système repose sur les mêmes concepts que la sélection des enregistrements avec SQL.
- Vous pouvez utiliser les opérateurs NOT, AND, OR, <, >, BETWEEN, LIKE...
- D'autres fonctionnalités sont possibles. Prenons le cas où nous souhaitons traiter dans le MDB tous les messages concernant le stocker, afficher et traiter des photos :
 - **@ActivationConfigProperty(**
 - **propertyName= "messageSelector" ,**
 - **propertyValue= "commande IN**
['afficher' , 'stocker' , 'traiter']")

Sélecteur de Messages

- Pour récupérer la propriété commande pendant la réception du message, on peut écrire le code suivant:

```
public void onMessage(Message m) {  
    System.out.println("-----");  
    try {  
        String commande=m.getStringProperty("commande");  
        System.out.println("Commande="+commande);  
        StreamMessage message = (StreamMessage) m;  
        if(commande.equals("afficher")){  
            gui.setTitle(message.readString());  
            byte[] octets = new byte[message.readInt()];  
            message.readBytes(octets);  
            gui.viewImage(octets);  
        } catch (Exception ex) {  
            ex.printStackTrace();  
        }  
    }  
}
```



Sélection de message dans une application cliente

- Si la réception des messages est traitée par une application cliente et non plus dans le serveur au moyen d'un **MDB**, il est également possible de choisir les messages à traiter.
- Cela se fait tout simplement lorsque nous créons le **MessageConsumer** au travers de la méthode **Session.createConsumer()**.
- En effet, cette méthode est surchargée. Nous pouvons prendre celle qui possède un deuxième argument correspondant au critère de sélections.
- Le critère de sélection est tout à fait du même ordre que celui que nous avons utilisé dans la rubrique précédente :
 - `MessageConsumer réception = session.createConsumer(destination, "commande='afficher'");`



Accusé de réception

- L'inconvénient d'un traitement asynchrone des messages est qu'il n'y a pas de valeur de retour à l'expéditeur.
- Il est donc difficile pour lui de savoir si le message a bien été transmis
- lorsqu'il souhaite le savoir bien entendu, il existe différentes solutions à ce problème.



Première Solution pour l'accusé de réception

- La première consiste à utiliser des accusés de réception, mécanisme transparent géré par le fournisseur et le conteneur MDB.
- Cela permet à l'application cliente de signaler que le message a bien été reçu.
- Sans cet accusé réception, le message continuera d'être envoyé.
- Ce mécanisme repose sur les transactions au niveau du MDB.
- Lorsque celle-ci est gérée par le conteneur, l'accusé est envoyé à la suite du commit ou non si la transaction échoue.

Première Solution pour l'accusé de réception

- Il existe deux modes d'accusé réception :
 - Auto-acknowledge : L'accusé de réception doit être envoyé dès que le message a été transféré au MDB. Il s'agit d'un envoi automatique lors du commit de la transaction.
 - activationConfig={
 @ActivationConfigProperty(propertyName="acknowledgeMode",
 propertyValue="Auto-acknowledge")}
 - Dups-ok-acknowledge : L'accusé de réception est repoussé à un instant indéterminé et le conteneur choisi ensuite le moment où il a peu de tâches à traiter pour l'envoyer. Cela permet évidemment d'économiser les ressources. Nous pouvons cependant déconseiller cette dernière valeur car le fournisseur pourrait croire que le message n'a pas été traité et déciderait de le transmettre à nouveau (ce qui pourrait entraîner des disfonctionnements). De plus, le coût d'envoi d'un accusé de réception est négligeable que ce soit en capacité de calcul ou en charge réseau.
 - activationConfig={
 @ActivationConfigProperty(propertyName="acknowledgeMode",
 propertyValue="Dups-ok-acknowledge") }

Deuxième Solution pour l'accusé de réception

- La deuxième solution consiste à spécifier la valeur JMSReplyTo dans les paramètres d'en-tête du message.
- Cela permet au destinataire d'envoyer une réponse vers la destination paramétrée.
- Du côté de l'expéditeur, nous définissons la destination de réponse de la manière suivante :
 - **Destination destination = (Destination)ctx.lookup("queue/MaFile");**
...
message.setJMSReplyTo(destination);
message.setText("Bienvenue");
...
- Le MDB recevant le message peut ensuite récupérer cette propriété et envoyer à son tour un message :
 - **Destination destination = message.getJMSReplyTo();**
- Cette méthode diffère de la précédente car elle permet un réel retour d'information concernant le traitement du message. Cette solution est typiquement utilisée dans un système d'expédition de commandes. Le message de retour alerte le système lorsque l'objet désiré est préparé et expédié..



JavaMail

med@youssf.net



API JavaMail

- JavaMail est l'API qui nous permet d'utiliser le courrier électronique.
- L'API JavaMail permet de s'abstraire de tout système de mail et d'utiliser la plupart des protocoles de communication de façon transparente.
- Ce n'est pas un serveur d'e-mails, mais un outil pour interagir avec le serveur de messagerie.
- Les applications développées avec JavaMail sont en réalité comparables aux différentes messageries clientes telle que Outlook, Firebird, etc.
- Cette API propose donc des méthodes pour lire ou envoyer des e-mails, rechercher un message, etc.
- Les classes et les interfaces de cette API sont regroupées dans le paquetage javax.mail.



Principaux protocoles de messagerie

- **SMTP (Simple Mail Transport Protocole)**, protocole qui permet l'envoi d'e-mails vers un serveur.
- **POP3 (Post Office Protocole)**, protocole qui permet la réception d'e-mails. Protocole très populaire sur Internet, il définit une boîte aux lettres unique pour chaque utilisateur.
- **IMAP (Internet Message Acces Protocole)**, protocole qui permet la réception d'e-mails. Ce protocole est plus complexe car il apporte des fonctionnalités supplémentaires : plusieurs répertoires par utilisateur, partage de répertoires entre plusieurs utilisateurs, maintient des messages sur le serveur, etc.
- **NNTP (Network News Transport Protocol)**, protocole utilisé par les forums de discussion (news).



Type MIME

- Le type **MIME** (**Multipurpose Internet Mail Extensions**) est un standard permettant d'étendre les possibilités du courrier électronique, comme la possibilité d'insérer des documents (images, sons, texte, etc.) dans un courrier.



Les destinataires

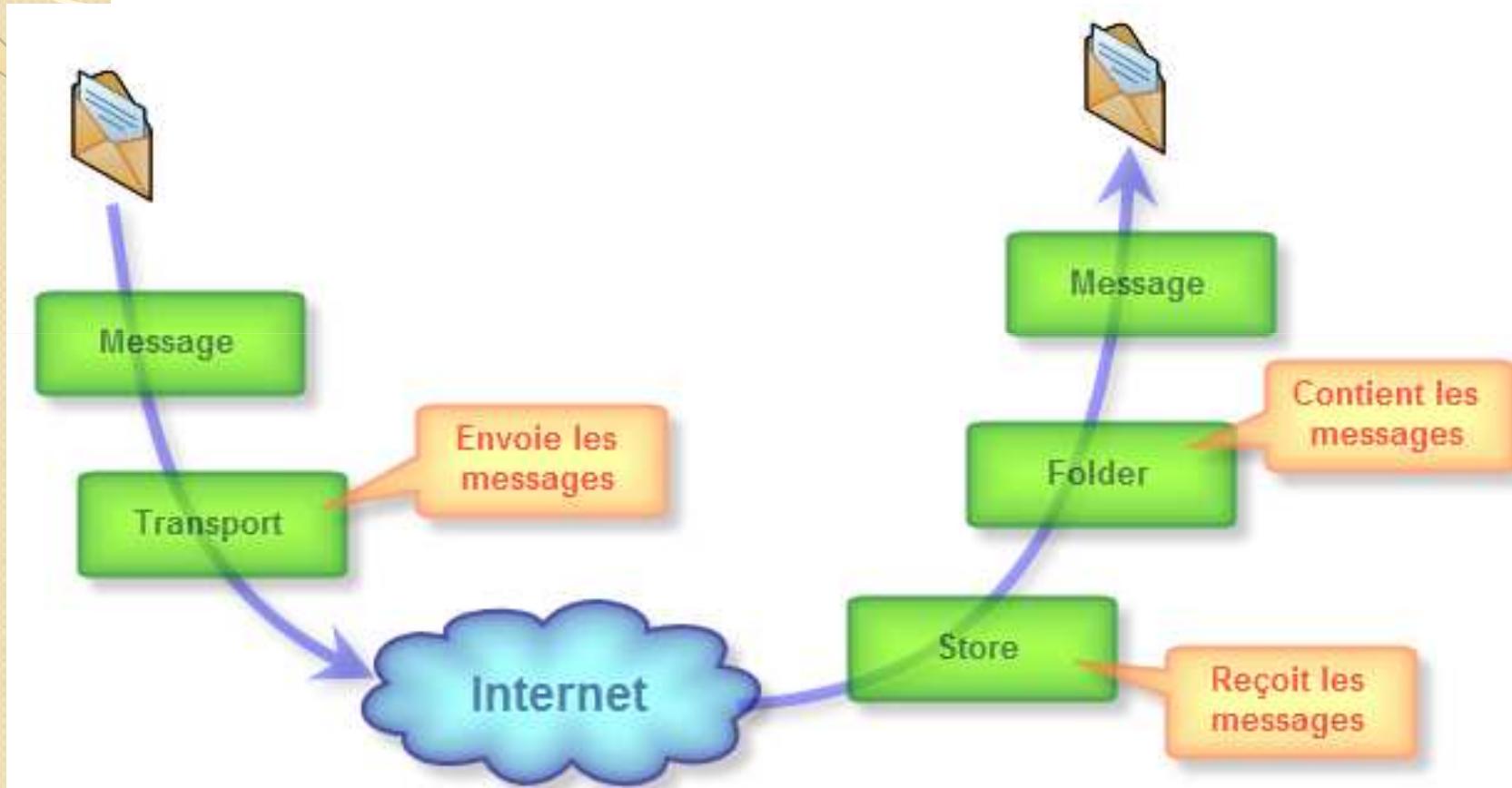
- **Lorsque nous envoyons un e-mail, l'adresse du destinataire peut être typée :**
 - **RecipientType.TO** : destinataire directe
 - **RecipientType.CC** : copie conforme
 - **RecipientType.BCC** : copie cachée



Attention au firewall

- Si vous avez un firewall (pare-feu) sur votre machine, vérifiez bien qu'il autorise le protocole SMTP sur le port 25.
- Sinon, les e-mails seront bloqués et ne pourrons pas être envoyés.

Architecture I



Objet Session

- A la manière de JMS, JavaMail possède une classe **javax.mail.Session** qui établit la connexion avec le serveur de messagerie.
- C'est elle qui encapsule les données liées à la connexion (options de configuration, login, mot de passe, nom du serveur) et à partir de laquelle les actions sont réalisées :
 - **Properties propriétés = new Properties()**
 - **propriétés.put("mail.smtp.host", "mail.youssfi.net");**
 - **propriétés.put("mail.smtp.auth", "true");**
 - **Session session = Session.getInstance(propriétés, null);**



La classe Message

- La classe `javax.mail.Message` est une classe abstraite qui encapsule le contenu du courrier électronique.
- Un message est composé
 - d'un en-tête qui contient l'adresse de l'auteur et du destinataire, le sujet, etc.
 - et d'un corps qui contient les données à envoyer ou à recevoir.
- JavaMail fournit en standard une classe fille nommée `javax.mail.internet.MimeMessage` pour les messages possédant le type MIME.

Méthodes de la classe Message

Méthode	Description
<code>Message(session)</code>	Créer un nouveau message.
<code>Message(Folder, int)</code>	Créer un message à partir d'un message existant.
<code>void addFromAdress(Adress[])</code>	Ajouter des émetteurs au message.
<code>void addRecipient(RecipientType, Address[])</code>	Ajouter des destinataires à un type d'envoi (direct, en copie ou en copie cachée).
<code>Flags getFlags()</code>	Retourne les états du message.
<code>Adress[] getFrom()</code>	Retourne les émetteurs.
<code>int getLineCount()</code>	Retourne le nombre ligne du message.
<code>Address[] getRecipients(RecipientType)</code>	Retourne les destinataires du type fourni en paramètre.
<code>int getSize()</code>	Retourne la taille du message.
<code>String getSubject()</code>	Retourne le sujet du message.
<code>Address getReplyTo()</code>	Renvoie les mails pour la réponse.
<code>Message reply(boolean)</code>	Créer un message pour la réponse : le booléen indique si la réponse ne doit être faite qu'à l'émetteur.
<code>void setContent(Object, String)</code>	Mettre à jour le contenu du message en précisant son type MIME.
<code>void setFrom(Address)</code>	Mettre à jour l'émetteur.
<code>void setRecipients(RecipientsType, Address[])</code>	Mettre à jour les destinataires d'un type.
<code>void setSentDate(Date)</code>	Mettre à jour la date d'envoi.
<code>void setText(String)</code>	Mettre à jour le contenu du message avec le type MIME « <code>text/plain</code> »
<code>void setReply(Address)</code>	Mettre à jour le destinataire de la réponse.
<code>void writeTo(OutputStream)</code>	Envoie le message au format RFC 822 dans un flux. Très pratique pour visualiser le message sur la console en passant en paramètre (<code>System.out</code>)

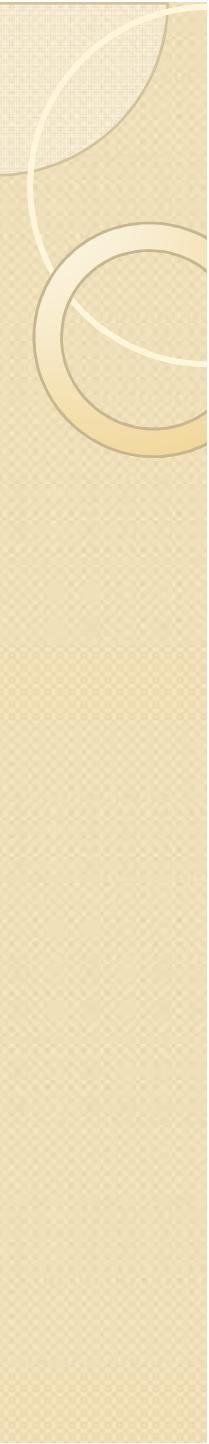


Création d'un message

- `Message message = new MimeMessage(session);`
- `message.setFrom(new InternetAddress("med@youssfifi.net"));`
- `message.setRecipient(Message.RecipientType.TO, new`
- `InternetAddress("sid@dgmail.com"));`
- `message.setSubject("Confirmation de la commande");`
- `message.setText("La commande n°34256 a été bien envoyée.");`
- `message.setSentDate(new Date());`

La classe Transport

- La classe `javax.mail.Transport` se charge d'envoyer le message avec le protocole adéquat.
- Dans notre cas, pour SMTP, il faut obtenir un objet Transport dédié à ce protocole en utilisant la méthode `getTransport("smtp")` d'un objet Session.
- Il faut ensuite établir la connexion en passant le nom du serveur de messagerie, le nom de l'utilisateur et son mot de passe.
- Pour envoyer le message que l'on a créé antérieurement, il faut utiliser la méthode `sendMessage()` en lui passant la liste des destinataires `getAllRecipients()`.
- Enfin, il faut fermer la connexion à l'aide de la méthode `close()` :
 - `Transport transport = session.getTransport("smtp");`
 - `transport.connect("smtp.enst-media.ac.ma", "user", "password");`
 - `transport.sendMessage(message, message.getAllRecipients());`
 - `transport.close();`



La classe Store et Folder

- La classe abstraite **Store** représente un système de stockage de messages ou "messagerie".
- Pour se connecter à cette "messagerie" et ainsi pouvoir consulter vos messages,
 - vous devez obtenir une instance de la classe **Store** avec la méthode **getStore()** de votre **session**, en lui donnant comme paramètre le protocole utilisé.
 - Ensuite vous n'avez plus qu'à vous connecter avec la méthode **connect()**, en lui précisant le nom du serveur, le nom d'utilisateur et le mot de passe.
 - La méthode **close()** permet de libérer la connexion avec le serveur.
 - **Store store = session.getStore("pop3");**
 - **store.connect("smtp.orange.fr", "user", "password");**
 - **...**
 - **store.close();**

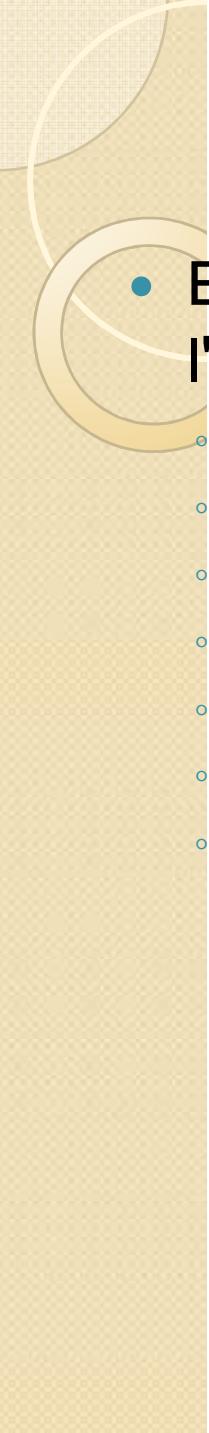
La classe Store et Folder

- La classe abstraite **Folder** représente un répertoire dans lequel les messages sont stockés.
- Pour obtenir un instance de cette classe, il faut utiliser la méthode **getFolder()** d'un objet de type **Store** en lui précisant le nom du répertoire.
- Avec le protocole "POP3" qui ne gère qu'un seul répertoire, le seul possible est "**INBOX**".
- Ensuite, nous appelons la méthode **open()** en précisant le mode d'utilisation :**READ_ONLY** ou **READ_WRITE**.
 - **Folder folder = store.getFolder("INBOX");**
 - **folder.open(Folder.READ_ONLY);**



Obtenir les messages

- Pour obtenir les messages contenus dans le répertoire, il faut appeler la méthode `getMessages()`.
- Cette méthode renvoie un **tableau** de **Message** qui peut être null si aucun message n'est renvoyé.
- Une fois les opérations terminées, il faut fermer le répertoire en utilisant la méthode `close()`.
 - **Message[] messages = folder.getMessages();**
 - **folder.close();**



Obtenir les messages

- En définitive, voici toute la procédure à suivre pour récupérer l'ensemble des messages stockées dans votre messagerie :

- `Store store = session.getStore("pop3");`
- `store.connect("smtp.orange.fr", "utilisateur", "mot-de-passe");`
- `Folder folder = store.getFolder("INBOX");`
- `folder.open(Folder.READ_ONLY);`
- `Message[] messages = folder.getMessages();`
- `folder.close();`
- `store.close();`



Pièces jointes

- Il est possible de joindre avec le mail des ressources sous forme de pièces jointes (attachments).
- Pour cela, nous devons passer par un objet de type **MultiPart**.
- Cet objet contient à son tour des objets **BodyPart**.
- La structure d'un objet **BodyPart** ressemble à celle d'un simple objet **Message**.
- Donc chaque objet **BodyPart** contient des attributs et un contenu.

Pièces jointes

- **Ainsi, pour mettre en oeuvre les pièces jointes, vous devez :**

1. Instancier un objet de type **MimeMessage**.
2. Renseigner les éléments qui composent l'en-tête : **émetteur, destinataire, sujet ...**
3. Instancier un objet de type **MimeMultiPart**.
4. Instancier un objet de type **MimeBodyPart** et alimenter le contenu de l'élément.
5. Ajouter cet objet à l'objet **MimeMultiPart** grâce à la méthode **addBodyPart()**.
6. Répéter l'instanciation et l'alimentation pour chaque ressource à ajouter.
7. Utiliser la méthode **setContent()** du message en passant en paramètre l'objet **MimeMultiPart** pour associer le message et les pièces jointes au mail.



Exemple de message avec pièces jointes

- **Multipart multipart = new MimeMultipart();**
// creation de la partie principale du message
- **BodyPart messageBodyPart = new MimeBodyPart();**
- **messageBodyPart.setText("Texte du courrier électronique");**
- **multipart.addBodyPart(messageBodyPart);**
// creation et ajout de la pièce jointe
- **messageBodyPart = new MimeBodyPart();**
- **DataSource source = new FileDataSource("image.gif");**
- **messageBodyPart.setDataHandler(new DataHandler(source));**
- **messageBodyPart.setFileName("image.gif");**
- **multipart.addBodyPart(messageBodyPart);**
// ajout des éléments au mail
- **message.setContent(multipart);**

- Les classes **DataSource** et **FileDataSource** sont des classes issues du paquetage **javax.activation**s et sont prévues pour une utilisation spécifique de l'API JavaMail. Elles permettent de gérer les différentes données de type MIME associées au courrier électronique.

Exemple de méthode d'envoi

```
private void envoyerCourrier(String affiche, String nom, Date date) throws Exception {
    Properties propriétés = new Properties();
    propriétés.put("mail.smtp.host", "mail.youssfi.net");
    // propriétés.put("mail.smtp.auth", "true");
    javax.mail.Session session = javax.mail.Session.getInstance(propriétés, null);
    javax.mail.Message courrier = new MimeMessage(session);
    courrier.setFrom(new InternetAddress("med@youssfi.net"));
    courrier.setRecipient(javax.mail.Message.RecipientType.TO, new InternetAddress("sid@youssfi.net"));
    courrier.setSubject("Stockage de photos");
    courrier.setSentDate(date);
    javax.mail.Multipart multipart = new MimeMultipart();
    javax.mail.BodyPart corpsCourrier = new MimeBodyPart();
    corpsCourrier.setText(affiche);
    multipart.addBodyPart(corpsCourrier);
    corpsCourrier = new MimeBodyPart();
    DataSource photo = new FileDataSource("J:/Photos/" + nom);
    corpsCourrier.setDataHandler(new DataHandler(photo));
    corpsCourrier.setFileName("J:/Photos/" + nom);
    multipart.addBodyPart(corpsCourrier);
    courrier.setContent(multipart);
    javax.mail.Transport transport = session.getTransport("smtp");
    transport.connect("smtp.wanadoo.fr", "emmanuel.remy", "mot-de-passe");
    transport.sendMessage(courrier, courrier.getAllRecipients());
    transport.close();
}
```



Gestion des transactions



Transactions

- *La plupart des travaux entourant la conception et le développement d'applications d'entreprise implique des décisions sur la façon de coordonner le flux de données persistantes.*
- *Cela comprend*
 - *quand et où cacher les données?*
 - *quand est ce que les rendre persistantes (généralement la base de données),*
 - *comment résoudre les tentatives simultanées d'accéder aux mêmes données et la façon de résoudre les erreurs qui peuvent se produire alors que les données dans la base de données sont dans un état contradictoire.*
- *Une base de données fiable est capable de traiter ces questions à un niveau bas dans le SGBD,*
- *Ces mêmes questions peuvent être résolues dans les niveaux intermédiaires (serveur d'application) et le client, et nécessitent généralement une logique d'application spéciale.*
- *L'un des principaux avantages de l'utilisation EJB 3 est son support des services à l'échelle de l'entreprise, comme la gestion des transactions et de contrôle de sécurité.*



Transactions

- Une transaction est un ensemble d'opérations qui doivent être exécutées en tant qu'unité.
- Ces opérations peuvent être synchrone ou asynchrone, et peuvent impliquer la persistance de données, l'envoi de messages, la validation des cartes de crédit, etc
- Un exemple classique est l'opération de transfert de compte à compte bancaire : on enlève sur un compte, on dépose sur l'autre...
 - Si l'une des deux opérations échoue, perte de cohérence !
 - On veut que soit les deux opérations réussissent, mais si une d'elles échoue, on annule le tout et on remet les comptes dans l'état initial !
 - On dit que les deux opérations forment une seule et même *transaction* !



Transactions

- Lorsque les opérations d'une transaction sont effectuées dans les bases de données ou d'autres ressources qui résident sur des ordinateurs ou des processus séparés, c'est ce qu'on appelle une **transaction distribuée**.
- Ces transactions à l'échelle de l'entreprise nécessitent une coordination spéciale entre les ressources impliquées et peuvent être extrêmement difficiles à programmer de manière fiable.
- C'est là que Java Transaction API (JTA) arrive, assurant l'interface que les ressources peuvent mettre en œuvre et à laquelle ils peuvent être liées, afin de participer à une transaction distribuée.
- Le conteneur EJB possède un gestionnaire de transaction qui supporte JTA et peut aussi participer à des transactions distribuées impliquant d'autres conteneurs EJB, ainsi que les ressources JTA tiers comme de nombreux systèmes de gestion de base de données (SGBD).



Les propriétés ACID d'une transaction

- Les transactions peuvent se présenter avec de différentes formes et tailles et peuvent impliquer des opérations synchrones et asynchrones, mais ils ont toutes quelques caractéristiques fondamentales en commun, connues sous le nom ACID.
- ACID se réfère aux quatre caractéristiques qui définissent une transaction robuste et fiable:
 - Atomicité,
 - cohérence,
 - isolation,
 - durabilité



Atomicité (Atomicity)

- Une transaction est formée par une ou plusieurs opérations qui sont exécutée en tant que groupe. Connue par le nom Unité de travail.
- Atomicité assure qu'à la conclusion de la transaction, ces opérations sont toutes exécuté avec succès (**successful commit**) ou aucune des ces opérations ne s'est exécutée (**successful rollback**).



Cohérence (Consistency)

- Une transaction consistante respecte l'intégrité des données.
- La Consistance assure qu'à la conclusion de la transaction, les données restent dans un état cohérent.
- Par conséquent, les contraintes de la base de données, et les règles de validations logiques ne sont pas violées.



Isolation

- L'isolation des transactions spécifient que le monde extérieur, ne pourra jamais voir les états intermédiaires d'une transaction.
- Les programmes externes visualisant les données manipulées dans la transaction ne pourront pas voir les données modifiées en cours de la transaction qu'après la validation finale de la transaction.



Durabilité (Durability)

- Les données modifiées dans une transaction une fois validée doivent être visibles pour d'autres applications.



Les propriétés ACID d'une transaction

- EJB 3 répond à ces besoins en fournissant un gestionnaire de transactions JTA robuste et une API de métadonnées déclarative qui peut être spécifié sur les composants métier interopérables et portables.
- Pratiquement toutes les applications Java EE ont besoin de services de transaction
- EJB apporte, au développeur de l'application, un gestionnaire de transaction très souple.
- Depuis sa création, le Framework EJB a fourni un moyen pratique pour gérer les transactions et le contrôle d'accès en laissant le développeur définir le comportement sur une base déclarative méthode par méthode.
- Au-delà de ces services fournis par le conteneur, EJB 3 permet aux développeurs de transformer le contrôle de l'application pour définir les limites d'événements de transaction et autres comportements personnalisé.



EJB 3 Transaction services

- Le modèle de transaction EJB 3 est construit sur ce modèle JTA, dans lequel les session beans de session ou d'autres applications clientes fournissent le contexte transactionnel dans lequel les services sont effectuées dans une unité de travail logique.
- Les services dans l'environnement Java EE comprennent
 - la création, la recherche, la mise à jour et de suppression des entités,
 - l'envoi de messages JMS à la file d'attente;
 - l'exécution des BMD,
 - les l'envoi des email;
 - l'invocation de services Web
 - les opérations JDBC.



Traitement par exceptions

- On peut s'en sortir en gérant des exceptions

```
try {  
    // retirer de l'argent du compte 1  
} catch (Exception e){  
    // Si une erreur est apparue, on arrête.  
    return;  
}  
try {  
    // Sinon, on dépose l'argent sur le compte 2  
} catch (Exception e){  
    // Si une erreur est apparue, on s'arrête, mais avant, on redépose  
    // l'argent sur le compte 1  
    return;  
}
```

- Qu'en pensez-vous ?



Traitement par exceptions

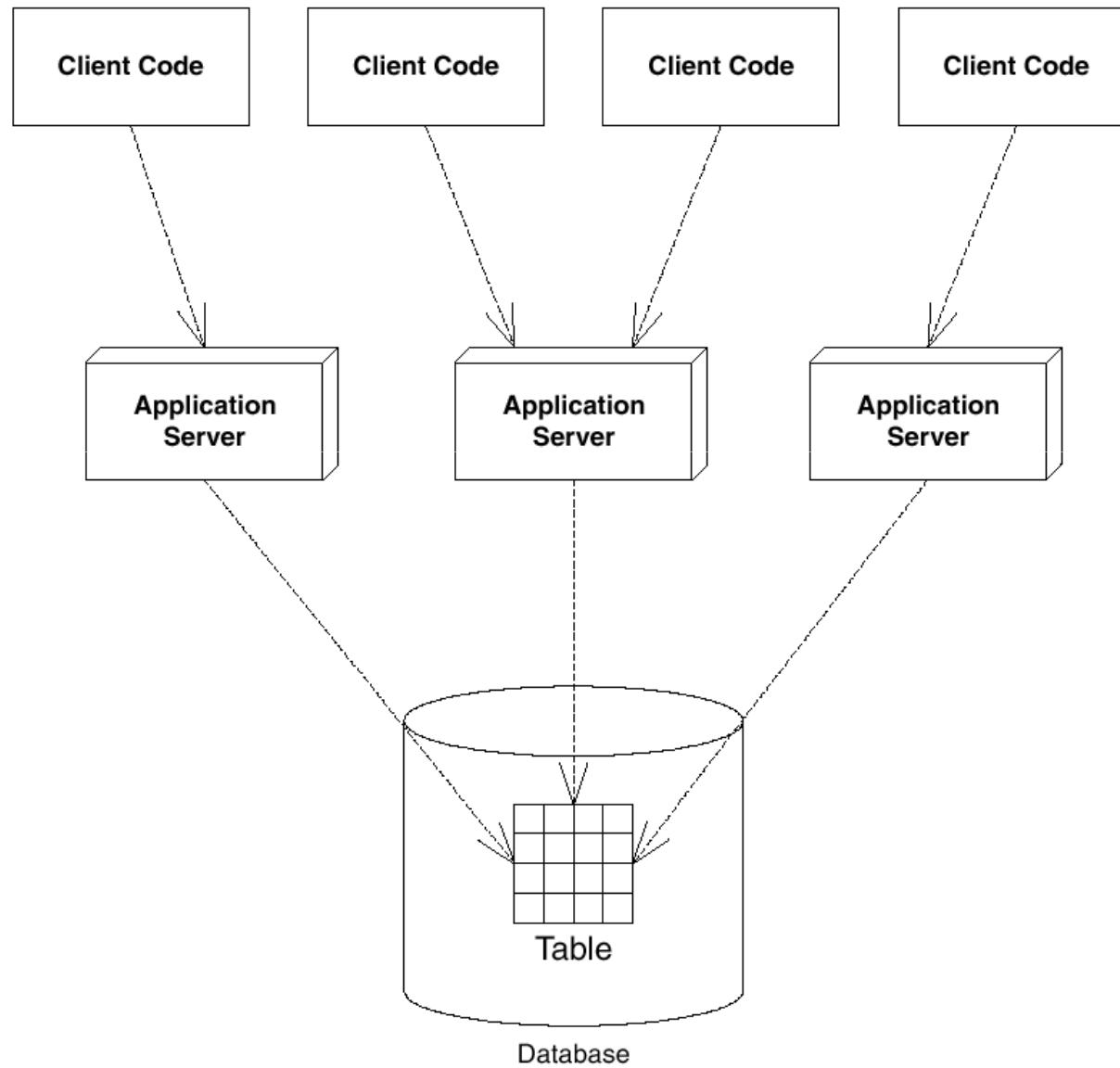
- Et si on échoue la dernière opération et qu'on n'arrive pas à remettre l'argent sur le compte I ?
- Et si au lieu d'un traitement simple on a un gros traitement à faire qui se compose de 20 opérations différentes ?
- Comment on teste tous les cas d'erreur ?



Panne réseau ou panne machine

- Le réseau plante, le client reçoit une *RMI Remote Exception*
- L'erreur est intervenue *avant* qu'on enlève l'argent, ou *après* ?
 - Impossible de savoir
- Peut-être que le SGBD s'est crashé ? La machine ? La BD est peut-être devenue inconsistante ?
- Le traitement par Exception est vraiment inacceptable ici, *il n'est pas suffisant* !

Partage concurrent de données





Partage concurrent de données

- Plusieurs clients consultent et modifient les mêmes données...
- On ne peut tolérer de perte d'intégrité des données !



Modèles de transactions

- Il existe deux modèles
 - 1. *Flat transactions* ou transactions à plat
 - Supportées par les EJBs
 - 2. *Nested transactions* ou transactions imbriquées
 - Non supportées par les EJBs pour le moment...



Flat transactions

- Modèle le plus simple.
- Après qu'une transaction ait démarré, on effectue des opérations...
- Si toutes les opérations sont ok, la transaction est validée (*committed*), sinon elle échoue (*aborted*)
- En cas de *commit*, les opérations sont validées (*permanent changes*)
- En cas d'*abort*, les opérations sont annulées (*rolled back*). L'application est également prévenue...



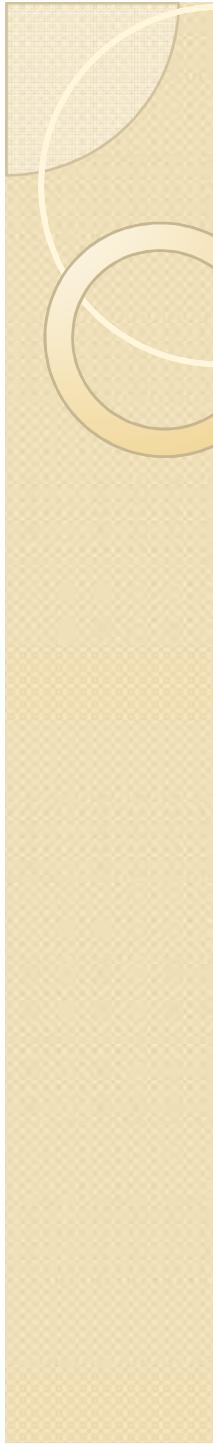
Transactions imbriquées

- Cas d'école : on veut faire un tour du monde
 1. Notre application achète un billet de train de Casablanca à Marseille,
 2. Puis un billet d'avion de Marseille à Londres,
 3. Puis un billet d'avion de Londres à New-York,
 4. L'application s'aperçoit qu'il n'y a plus de billet d'avion disponible ce jour-là pour New-York...
- Tout échoue et on annule toutes les réservations !



Transactions imbriquées

- Avec un modèle de transactions imbriquée, une transaction peut inclure une autre transaction,
- Si on ne trouve pas de vol pour New-York, on essaiera peut-être de prendre une correspondance par Philadelphie...

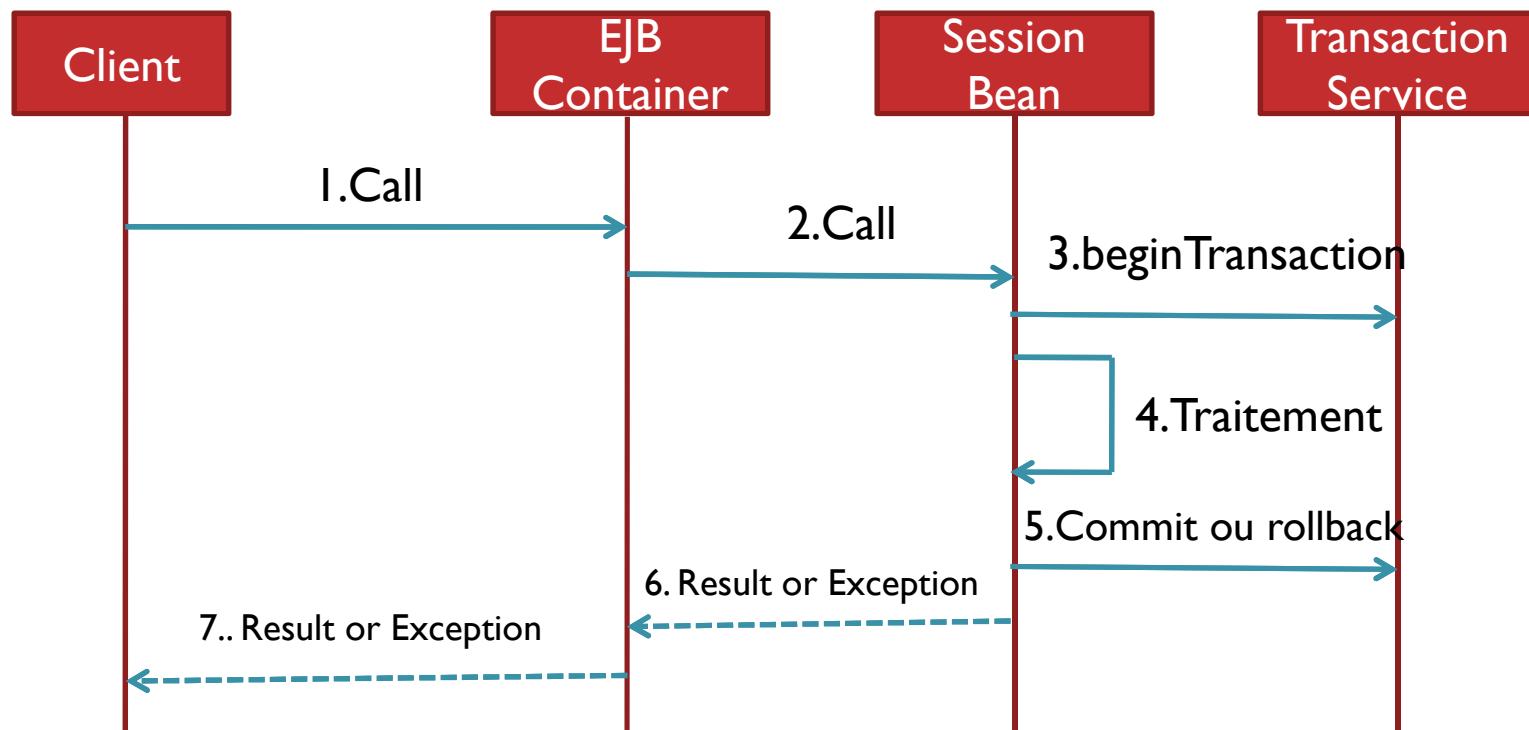


Gestion des transactions avec les EJBs

- Seul le modèle flat est supporté.
- Le code que le développeur écrit, s'il décide de gérer les transactions par programmation, demeurera d'un très haut niveau,
 - Simple vote pour un *commit* ou un *abort*,
 - Le container fait tout le travail en coulisse...
- 3 manières de gérer les transactions
 1. Par **programmation** (au niveau du bean),
 2. De manière **déclarative** (par le container),
 3. De manière **initiée par le client**.

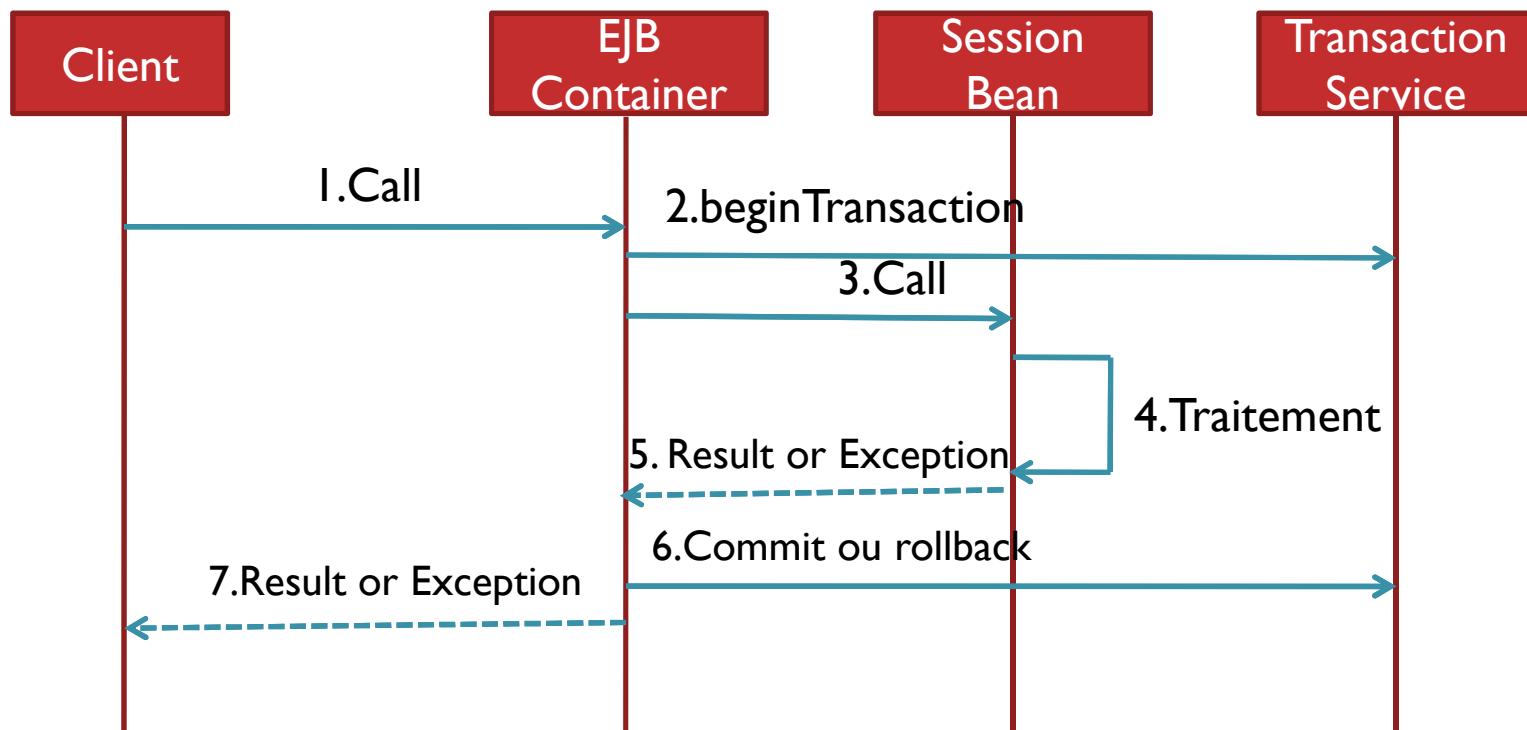
Gestion des transactions par programmation

- Responsable : le développeur de bean
- Il décide dans son code du *begin*, du *commit* et du *abort*

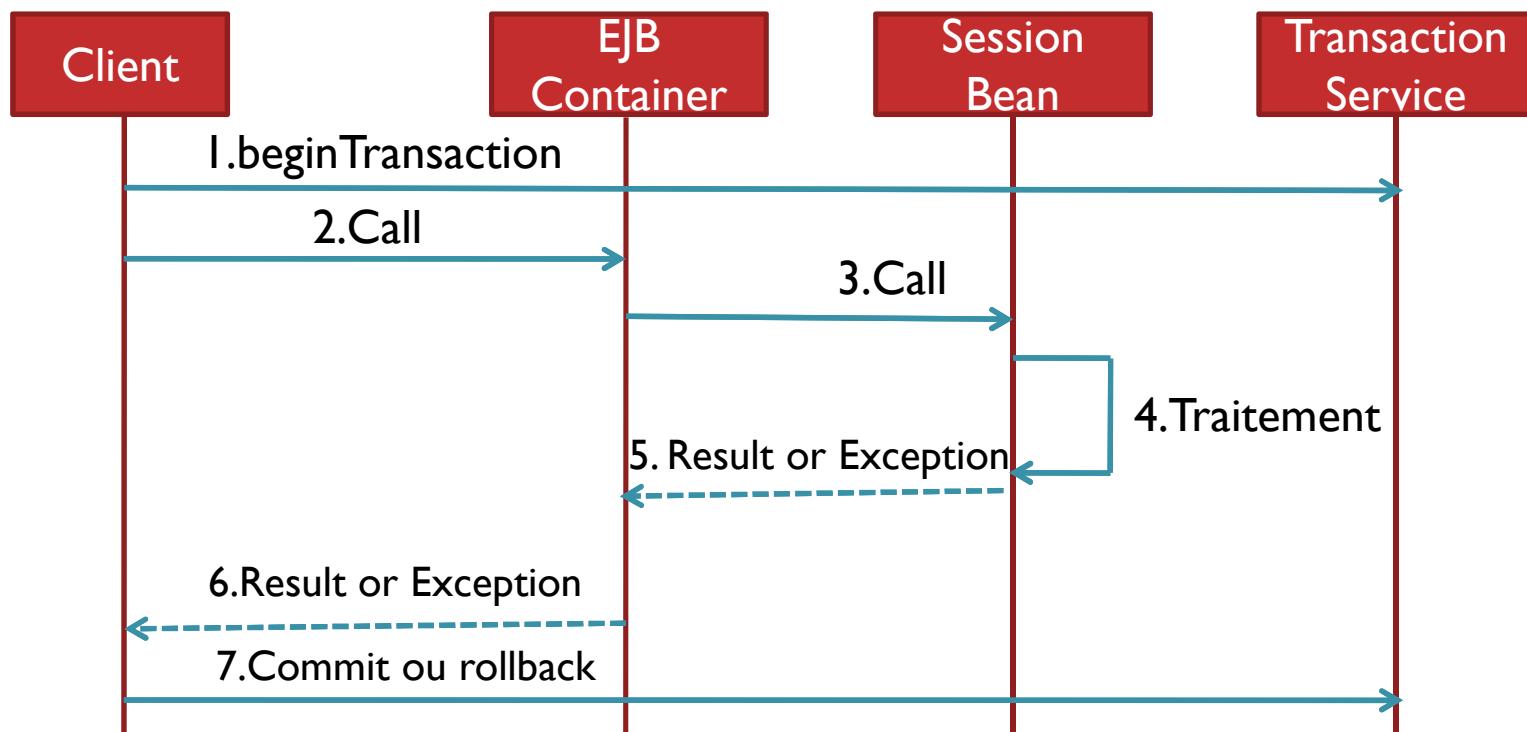


Gestion par le container

- C'est le mode par défaut
- Le bean est automatiquement enrôlé (enrolled) dans une transaction...
- Le container fait le travail...



Transactions initiées par le client





Que choisir ?

- Par programmation : contrôle très fin possible...
- Déclaratif : intéressante, mais granularité importante,
- Contrôlé par le client
 - N'empêche pas de gérer les transactions dans le bean (par programmation ou de manière déclarative)
 - Ajoute une couche de sécurisation en plus, qui permet de détecter les crashes machine, réseau, etc...

Transactions et Message-Driven Beans

- Bean-Managed Transactions
 - La transaction commence et se termine après que le message a été reçu par le MDB.
 - On indique dans le descripteur de déploiement les *acknowledgement modes* pour indiquer au container comment accuser réception...
- Container-Managed Transactions
 - La réception du message s'inscrit dans la même transaction que les appels de méthodes métier du MDB. En cas de problème, la transaction fait un rollback. Le container envoi accusé de réception (*message acknowledgement*)
- Pas de transaction
 - Le container accusera réception après réception. Quand exactement, ce n'est pas précisé...



Transactions et Message-Driven Beans

- Que choisir ?
- Si on décide de ne pas laisser le container gérer les transactions, on a pas de moyen de conserver le message dans la queue de destination si un problème arrive.
 - On choisit Container-Managed Transaction
- Piège : si un MDB est expéditeur et consommateur du message, le tout intervient dans une même transaction
 - Impossible de terminer ! Le message expédié n'est pas mis dans la queue de destination de manière définitive tant que l'envoi n'est pas commité.
 - Donc le message ne peut être consommé! Cqfd.
 - Solution : appeler `commit()` sur l'objet JMS `session` juste après l'envoi.



Spécifier le mode de gestion

- L'annotation **@TransactionManagement**, placée sur la classe du bean, permet de spécifier le mode de gestion des transaction :
 - **@TransactionManagement(TransactionManagementType.CONTAINER)**
 - **@TransactionManagement(TransactionManagementType.BEAN)**

Configuration des transactions

- L'annotation `@javax.ejb.TransactionAttribute` permet de préciser dans quel contexte transactionnel une méthode d'un EJB sera invoquée.
- Cette annotation est incompatible avec la valeur **BEAN** de l'annotation `@TransactionManagement`.
- Elle s'utilise sur une classe d'un EJB ou sur une méthode d'un EJB session.
- Utilisée sur une classe, l'annotation s'applique à toutes les méthodes de l'EJB.
- Elle possède un attribut `value` qui peut avoir plusieurs valeurs possibles :
 - `@TransactionAttribute(value=TransactionAttributeType.REQUIRED)`



Propriété value de `@TransactionAttribute`

- La propriété `value` de l'annotation `@TransactionAttribute` permet de préciser le contexte transactionnel d'invocation d'une méthode de l'EJB. Cet attribut peut prendre plusieurs valeurs :
 - `TransactionAttributeType.MANDATORY`
 - `TransactionAttributeType.REQUIRED` (valeur par défaut)
 - `TransactionAttributeType.REQUIRED_NEW`
 - `TransactionAttributeType.SUPPORTS`
 - `TransactionAttributeType.NOT_SUPPORTED`
 - `TransactionAttributeType.NEVER`



Différents de types de contextes transactionnels

- **NOT_SUPPORTED :**
 - Suspend la propagation de la transaction aux traitements de la méthode et des appels aux autres EJB de ces traitements.
 - Une éventuelle transaction démarrée avant l'appel d'une méthode marquée avec cet attribut est suspendue jusqu'à la sortie de la méthode.
- **SUPPORTS :**
 - La méthode est incluse dans une éventuelle transaction démarrée avant son appel.
 - Cet attribut permet à la méthode d'être incluse ou non dans une transaction



Différents de types de contextes transactionnels

- **REQUIRED :**
 - La méthode doit obligatoirement être incluse dans une transaction.
 - Si une transaction est démarrée avant l'appel de cette méthode, alors la méthode est incluse dans la portée de la transaction.
 - Si aucune transaction n'est définie à l'appel de la méthode, le conteneur va créer une nouvelle transaction dont la portée concerne les traitements de la méthode et les appels aux EJB de ces traitements.
 - La transaction prend fin à la sortie de la méthode (valeur par défaut lorsque l'annotation n'est pas utilisée ou définie dans le fichier de déploiement)
- **REQUIRES_NEW :**
 - Une nouvelle transaction est systématiquement démarrée même si une transaction est démarrée lors de l'appel de la méthode. Dans ce cas, la transaction existante est suspendue jusqu'à la fin de l'exécution de la méthode



Différents de types de contextes transactionnels

- **MANDATORY :**
 - La méthode doit obligatoirement être incluse dans la portée d'une transaction existante avant son appel.
 - Aucune transaction ne sera créée et elle doit obligatoirement être fournie par le client appelant.
 - L'appel de la méthode non incluse dans la portée d'une transaction lève une exception de type `javax.ejb.EJBTransactionRequiredException`
- **NEVER :**
 - La méthode ne doit jamais être appelée dans la portée d'une transaction.
 - Si c'est le cas, une exception de type `EJBException` est levée

Remarques importantes

- Il est fortement recommandé d'utiliser un contexte de persistance ([EntityManager](#)) dans la portée d'une transaction afin de s'assurer que tous les accès à la base de données se font dans un contexte transactionnel. Ceci implique d'utiliser les attributs de transaction [Required](#), [Required_New](#) ou [Mandatory](#).
- Un EJB de type [Message Driven](#) ne peut utiliser que les attributs de transaction [NotSupported](#) et [Required](#). L'attribut [NotSupportedException](#) précise que les messages ne seront pas traités dans une transaction. L'attribut [Required](#) précise que les messages seront traités dans une transaction créée par le conteneur.
- Il n'est pas possible d'utiliser l'attribut [Mandatory](#) avec un EJB qui est proposé sous la forme d'un [service web](#).
- La gestion des attributs de transaction est importante car l'utilisation d'un EJB dans un contexte transactionnel [est coûteux en ressources](#). Il faut bien tenir compte du fait que la valeur par défaut des attributs de transaction est utilisée si aucun attribut n'est précisé et que cet attribut par défaut est [REQUIRED](#), ce qui place automatiquement l'EJB dans un contexte transactionnel.
- Il est donc fortement recommandé d'utiliser un attribut de transaction [NotSupported](#) lorsqu'aucune transaction n'est requise.

Tous les attributs ne s'appliquent pas à tous les beans

TRANSACTION ATTRIBUTE	STATELESS BEAN	STATEFUL SESSION BEAN IMPLEMENTING SESSION SYNCHRONIZATION	ENTITY BEAN	MESSAGE-DRIVEN BEAN
Required	Yes	Yes	Yes	Yes
RequiresNew	Yes	Yes	Yes	No
Mandatory	Yes	Yes	Yes	No
Supports	Yes	No	No	No
NotSupported	Yes	No	No	Yes
Never	Yes	No	No	No



Transactions gérées par programmation

- Plus complexes à manipuler, mais plus puissantes,
- Le développeur doit utiliser Java Transaction API (JTA)



CORBA Object Transaction Service (OTS)

- Dans une transaction de nombreuses parties sont impliquées : le driver de DB, le bean, le container,
- Premier effort pour assurer les transactions dans un système distribué : le service CORBA Object Transaction Service (OTS)
- OTS = ensemble d'interfaces pour le gestionnaire de transactions, le gestionnaire de ressources, etc...
- Très complexe



Java Transaction Service (JTS)

- Sun Microsystems a encapsulé OTS en deux API distinctes
 - JTS s'adresse aux vendeurs d'outils capables d'assurer un service de transaction, elle couvre tous les aspects complexes d'OTS,
 - JTA s'adresse au développeur d'application (vous) et simplifie grandement la gestion de transactions en contexte distribué.



Java Transaction API (JTA)

- JTA permet au programmeur de contrôler la gestion de transaction dans une logique métier.
- Il pourra faire les *begin*, *commit*, *rollback*, etc...
- Il peut utiliser JTA dans des EJB mais aussi dans n'importe quelle application cliente.
- JTA se compose de deux interfaces distinctes.



JTA : deux interfaces

- Une pour les gestionnaires de ressources X/Open XA (hors sujet pour nous)
- Une pour le programmeur désirant contrôler les transactions :

`javax.transaction.UserTransaction`

L'interface

```
public interface  
javax.transaction.UserTransaction {  
    public void begin();  
    public void commit();  
    public int getStatus();  
    public void rollback();  
    public void setRollbackOnly();  
    public void setTransactionTimeout(int);  
}
```

Constantes de la classe

- Constantes renvoyées par getStatus()

```
public interface javax.transaction.Status {  
    public static final int STATUS_ACTIVE;  
    public static final int STATUS_NO_TRANSACTION;  
    public static final int STATUS_MARKED_ROLLBACK;  
    public static final int STATUS_PREPARING;  
    public static final int STATUS_PREPARED;  
    public static final int STATUS_COMMITTING;  
    public static final int STATUS_COMMITTED;  
    public static final int STATUS_ROLLING_BACK;  
    public static final int STATUS_ROLLEDBACK;  
    public static final int STATUS_UNKNOWN;  
}
```

Exemple des gestion des transactions au niveau du bean

```
package metier.session;
import ...
@Stateless(mappedName="BP",name="BP3")
@TransactionManagement(TransactionManagementType.BEAN)
public class BanqueEjbJpaImpl implements IBanqueLocal,IBanqueRemote {
@PersistenceContext(unitName="UP_BP")
private EntityManager em;
@Resource UserTransaction transaction;
@Override
public void effectuerVirement(double mt, Long cpte1, Long cpte2) {
try {
    transaction.begin();
    Compte cp1=consulterCompte(cpte1); Compte cp2=consulterCompte(cpte2);
    if(cp1.getSolde()<mt) transaction.rollback();
    cp1.setSolde(cp1.getSolde()-mt); cp2.setSolde(cp2.getSolde()+mt);
    transaction.commit();
} catch (Exception e) { e.printStackTrace(); }
}
```



Transactions initiées par le client

- C'est la dernière des méthodes présentées au début de ce chapitre...
- Il est nécessaire d'obtenir une référence sur un objet **UserTransaction**, fourni par JTA
 - Via JNDI !
- Faire attention à ce que chaque transaction ne dure pas longtemps !!!
 - Piège classique !

Transactions initiées par le client (servlet par ex)

```
try {  
    // Obtenir une transaction par JNDI  
    Context ctx = new InitialContext();  
    userTran = (javax.transaction.UserTransaction)  
  
    ctx.lookup("java:comp/UserTransaction");  
    userTran.begin();  
    // Operations de la transaction  
    userTran.commit();  
} catch (Exception e) {  
    // Traiter les exceptions.  
    // Certaines peuvent provoquer un rollback.  
}
```



Niveau d'isolation

- Le niveau d'isolation limite la façon dont les transactions multiples et entrelacées interfèrent les unes sur les autres dans une BD multi-utilisateur.
- 3 types de violations possibles
 1. Lecture impropre (ou brouillée),
 2. Lecture ne pouvant être répétée,
 3. Lecture fantôme.



Niveau d'isolation

- Lecture impropre
 - La transaction T1 modifie une ligne, la transaction T2 lit ensuite cette ligne,
 - Puis T1 effectue une annulation (rollback),
 - T2 a donc vu une ligne qui n'a jamais vraiment existé.
- Lecture ne pouvant être répétée
 - T1 extrait une ligne,
 - T2 met à jour cette ligne,
 - T1 extrait à nouveau la même ligne,
 - T1 a extrait deux fois la même ligne et a vu deux valeurs différentes.

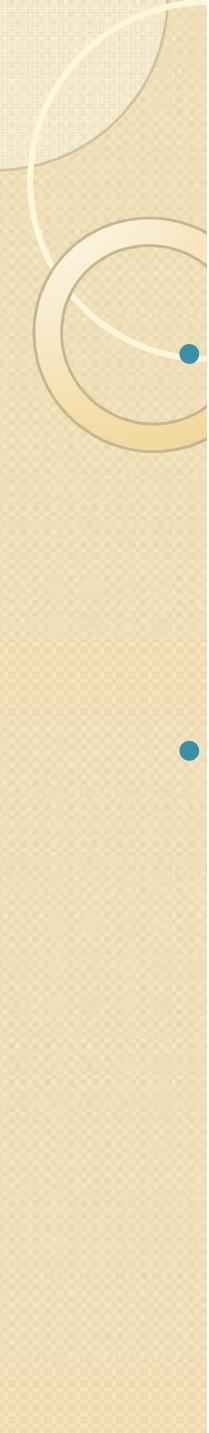


Niveau d'isolation

- Lecture fantôme
 - T1 lit quelques lignes satisfaisant certaines conditions de recherche,
 - T2 insère plusieurs lignes satisfaisant ces mêmes conditions de recherche,
 - Si T1 répète la lecture elle verra des lignes qui n'existaient pas auparavant. Ces lignes sont appelées *des lignes fantômes*.

Niveau d'isolation

Attribut	Syntaxe	Description
Uncommitted	TRANSACTION_READ_UNCOMMITTED	Autorise l'ensemble des trois violations
Committed	TRANSACTION_READ_COMMITTED	Autorise les lectures ne pouvant être répétées et les lignes fantômes, n'autorise pas les lectures brouillées
Repeatable	TRANSACTION_REPEATABLE_READ	Autorise les lignes fantômes mais pas les deux autres violations
Serializable	TRANSACTION_SERIALIZABLE	N'autorise aucune des trois violations



Quel niveau utiliser

- Uncommitted

- Uniquement si on est sûr qu'une transaction ne pourra être mise en concurrence avec une autre.
- Performant mais dangereux !
- A éviter pour les applications mission-critical !

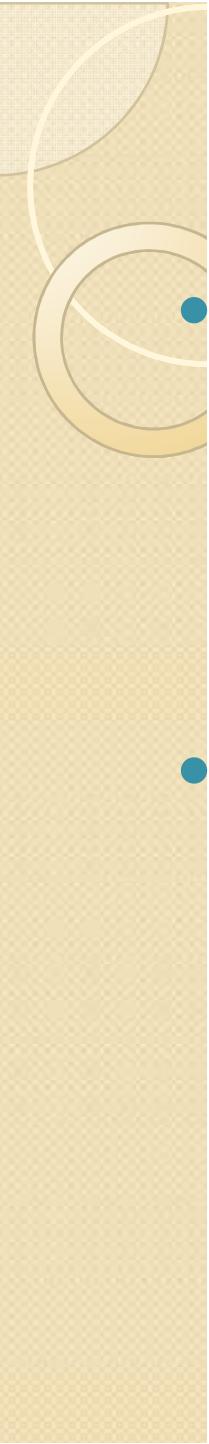
- Committed

- Utile pour les applications qui produisent des rapports sur une base de donnée. On veut lire des données consistantes, même si pendant qu'on les lisait quelqu'un était en train de les modifier.
- Lisent un snapshot des données commitées...
- Niveau d'isolation par défaut de la plupart des BD (Oracle...) et aussi le niveau d'isolation de JPA



Quel niveau utiliser

- Repeatable
 - Lorsqu'on veut pouvoir lire et modifier des lignes, les relire au cours d'une même transaction, sans perte de consistance.
- Serializable
 - Pour les applications mission-critical qui nécessitent un niveau d'isolation absolu, ACID 100% !
 - Attention ,les performances se dégradent à vitesse grand V avec ce mode !



Comment spécifier ces niveaux ?

- Transactions gérées par le bean : appel de `Connection.SetTransactionIsolation(. . .)`.
après avoir récupéré la connexion, par exemple par

```
DataSource ds =
    jndiCtxt.lookup("java:comp/env/jdbc/mabd");
ds.getConnection();
```
- Transactions gérées par le container
 - Non, on ne peut pas spécifier le niveau d'isolation dans le descripteur !
 - On le fera via le driver JDBC, ou via les outils de configuration de la DB ou du container,
 - Problèmes de portabilité !

Impossibilité de spécifier le niveau

Isolation Portability Issues

Unfortunately, there is no way to specify isolation for container-managed transactional beans in a portable way—you are reliant on container and database tools. This means if you have written an application, you cannot ship that application with built-in isolation. The deployer now needs to know about transaction isolation when he uses the container's tools, and the deployer might not know a whole lot about your application's transactional behavior. This approach is also somewhat error-prone, because the bean provider and application assembler need to informally communicate isolation requirements to the deployer, rather than specifying it declaratively in the deployment descriptor.

When we queried Sun on this matter, Mark Hapner, coauthor of the EJB specification, provided this response: “Isolation was removed because the vendor community found that implementing isolation at the component level was too difficult. Some felt that isolation at the transaction level was the proper solution; however, no consensus was reached on a specific replacement semantics.

“This is a difficult problem that unfortunately has no clear solution at this time . . . The best strategy is to develop EJBs that are as tolerant of isolation differences as possible. This is the typical technique used by many optimistic concurrency libraries that have been layered over JDBC and ODBC.”



Deux stratégies

- Lorsqu'on veut gérer les transactions, on doit toujours choisir entre deux stratégies
 - I. Stratégie pessimiste
 - On pense qu'il va y avoir des problèmes, on prend donc un verrou lors des accès BD, on fait notre travail, puis on libère le verrou.
 - 2. Stratégie optimiste
 - Espère que tout va bien se passer.
 - Néanmoins, si la BD détecte une collision, on fait un rollback de la transaction.



Que faire dans le code EJB ???

- Ok, nous avons vu comment spécifier le type de gestion de transaction, gérée par le bean ou le container,
- Nous avons vu les niveaux d'isolations, que l'on spécifie la plupart du temps via les pilotes JDBC,
- Mais que faire en cas de rollback par exemple
 - On ne peut pas re-essayer indéfiniment d'exécuter une transaction, on peut envoyer une Exception au client...
 - On veut également être tenu au courant de ce qu'il se passe pendant l'exécution d'une transaction.



Que faire dans le code EJB ???

- En cas de rollback, si on envoie une Exception au client, que faire de l'état du bean ?
 - Si le bean est stateful, on risque d'avoir un état incorrect (celui qui a provoqué l'échec de la transaction),
 - Lors du design d'un bean, il faut prévoir la possibilité de restaurer un état correct,
 - Le container ne peut le faire pour vous car le traitement est en général spécifique à l'application,
 - Il peut néanmoins vous aider à réaliser cette tâche.



Que faire dans le code EJB ???

- L'EJB peut implementer une interface optionnelle
javax.ejb.SessionSynchronization

```
public interface javax.ejb.SessionSynchronization {  
    public void afterBegin();  
    public void beforeCompletion();  
    public void afterCompletion(boolean committed);  
}
```

- Uniquement pour les session beans stateful dont les transactions sont gérées par le container.



Que faire dans le code EJB ???

- Le container appelle afterCompletion() que la transaction se soit terminée par un *commit* ou par un *abort*
 - *Le paramètre de la méthode nous signale dans quel cas on se trouve*

Que faire dans le code EJB ???

@Stateful

```
public class CountBean implements SessionSynchronization
{
    private int val;
    private int oldVal;

    public CountBean(int val) {
        this.val=val;
        this.oldVal=val;
    }

    public void afterBegin() { oldVal = val; }

    public void beforeCompletion() {}

    public void afterCompletion(boolean committed) {
        if (! committed)
            val = oldVal;
    }

    public int count() {
        return ++val;
    }
}
```