

Enterprise JavaBeans 3.0

Introduction générale

Saloua Ben Yahia

Les promesses des EJB

■ Enterprise JavaBeans

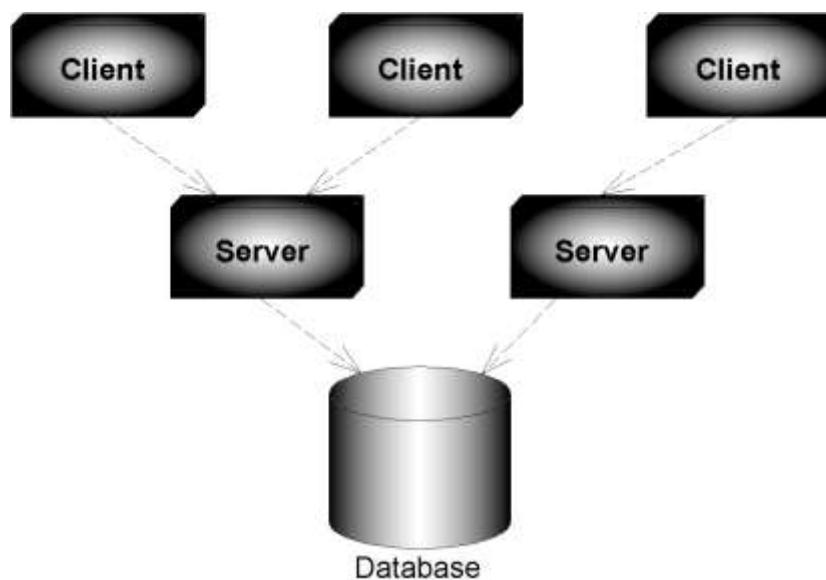
- Standard industriel pour un modèle de composant logiciel distribué,
- Permet d'implémenter des "objets métier" d'une manière propre et réutilisable,
- Pour le développement RAD d'applications côté serveur

■ Questions :

- De quoi a-t-on besoin lorsqu'on développe une application distribuée orientée objet ?
- Qu'est-ce que les EJBs et qu'apportent-elles ?
- Quels sont les acteurs dans l'écosystème EJB ?

Motivation des EJBs

- Considérons : un site de gestion de portefeuille boursier, une application bancaire, un centre d'appel, un système d'analyse de risque
- Nous parlons ici d'applications *distribuées*.



Choses à considérer lorsqu'on construit une application distribuée

- Si on prend une application monolithique et qu'on la transforme en application distribuée, où plusieurs clients se connectent sur plusieurs serveurs qui utilisent plusieurs SGBD, quels problèmes se posent alors ?

Choses à considérer lorsqu'on construit une application distribuée

- Protocoles d'accès distants (CORBA, RMI, IIOP...)
- Gestion de la charge,
- Gestion des pannes,
- Persistence, intégration au back-end,
- Gestion des transactions,
- Clustering,
- Redéploiement à chaud,
- Arrêt de serveurs sans interrompre l'application,
- Gestion des traces, réglages (tuning and auditing),
- Programmation multithread
- Problèmes de nommage
- Sécurité, performances,
- Gestion des états
- Cycle de vie des objets
- Gestion des ressources (Resource pooling) 5
- Requête par message (message-oriented middleware)

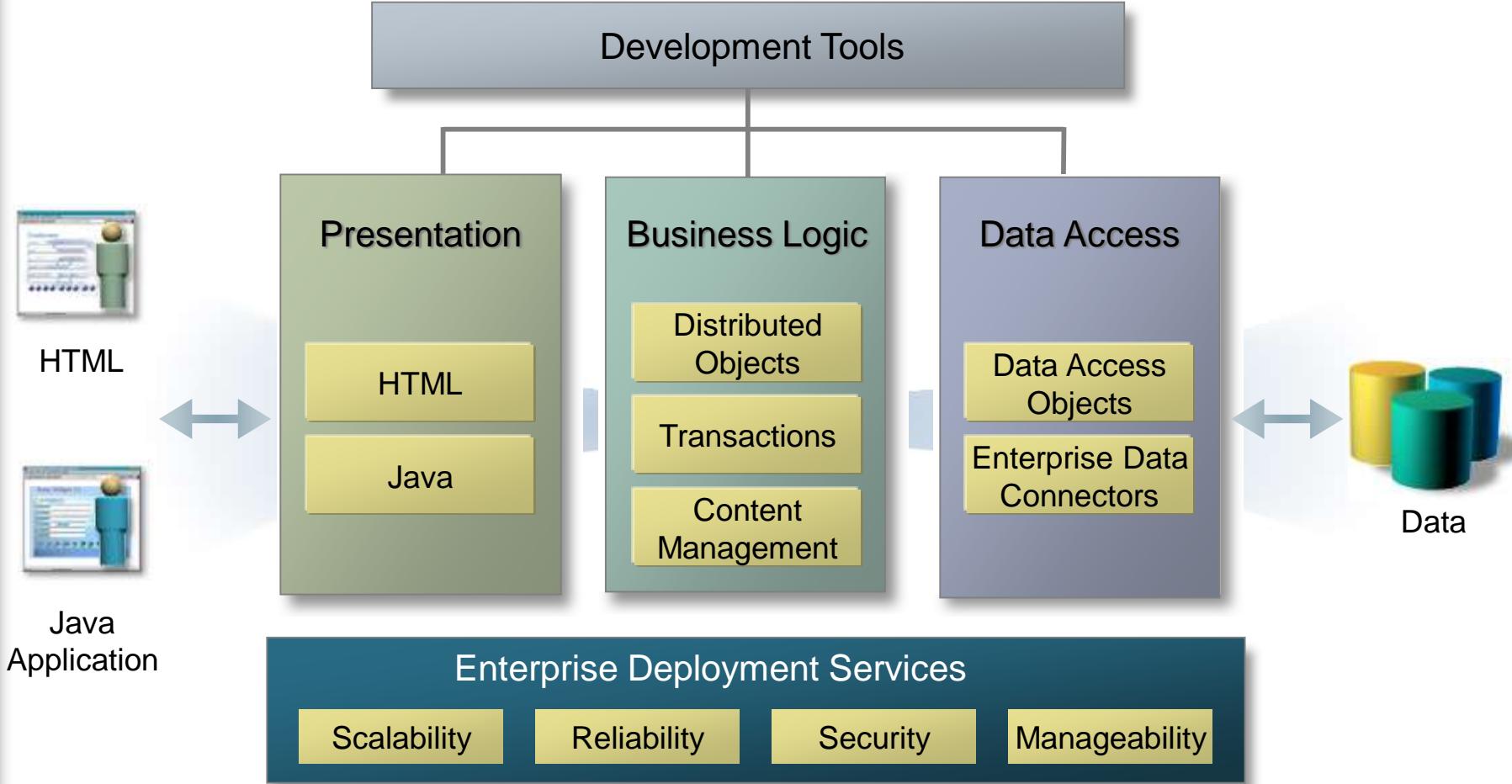
Qui s'occupe de tout ceci : le middleware !

- Dans le passé, la plupart des entreprises programmaient leur propre middleware.
 - Adressaient rarement tous les problèmes,
 - Gros risque : ça revient cher (maintenance, développement)
 - Orthogonal au secteur d'activité de l'entreprise (banque, commerce...)
- Pourquoi ne pas acheter un produit ?
 - Oracle, IBM, BEA... proposent depuis plusieurs années des middleware...
 - Aussi appelés *serveurs d'application*.

Serveur d'application : diviser pour régner !

- Un serveur d'application fournit les services middleware les plus courants,
- Permettent de se focaliser sur l'application que l'on développe, sans s'occuper du reste.
- Le code est déployé sur le serveur d'application.
- Séparation des métiers et des spécificités : d'un côté la logique métier, de l'autre la logique middleware.

Serveurs d'application



Encore mieux !

- Il est possible d'acheter ou de réutiliser une partie de la logique métier !
- Vous développez votre application à l'aide de *composants*.
 - Code qui implémente des interfaces prédéfinies.
 - Sorte de boîte noire.
 - Un bout de logique facilement réutilisable.
 - Un composant n'est pas une application complète.
Juste *un bout*.
 - On assemble les composants comme un puzzle, afin de résoudre des problèmes importants.

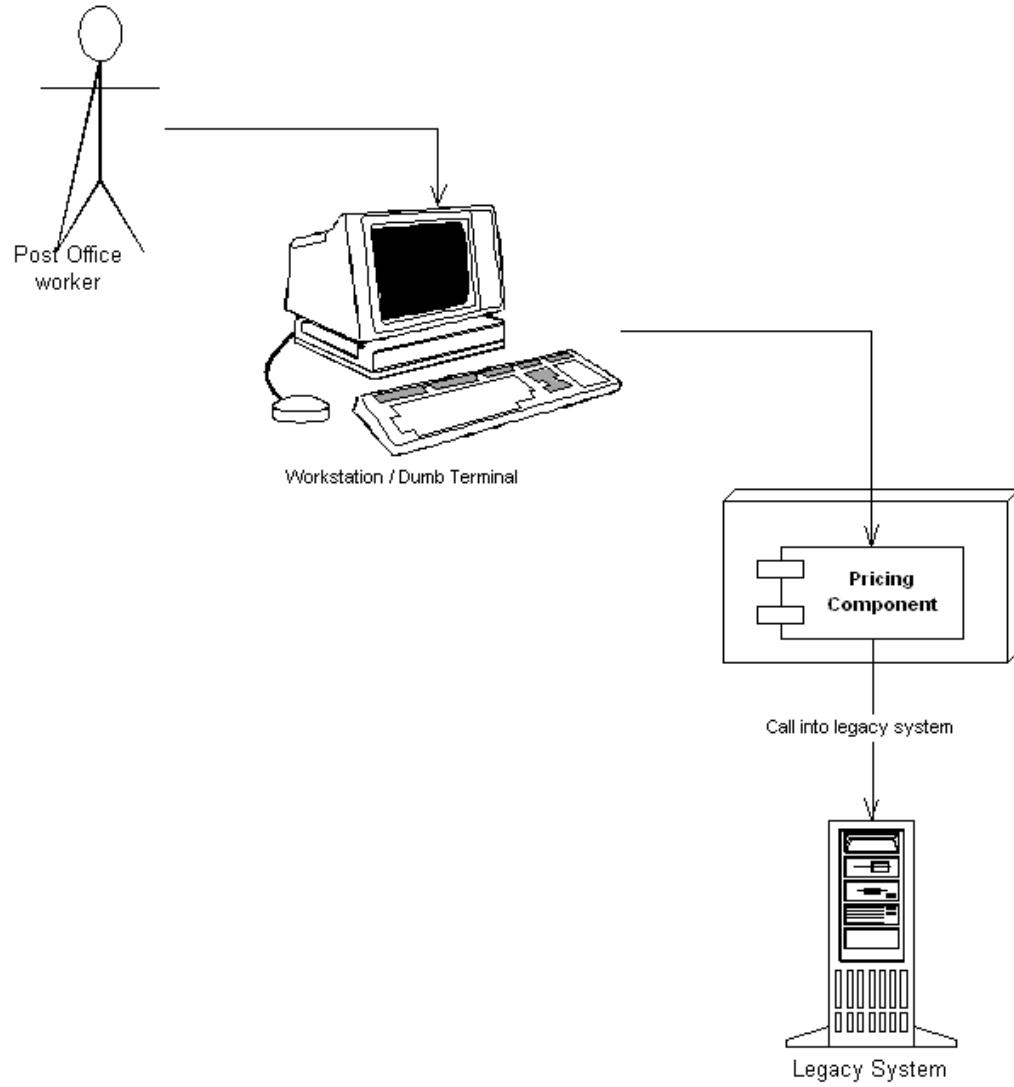
Composant logiciel réutilisable

- Une entreprise peut acheter un composant et l'intégrer avec des composants qu'elle a développé.
 - Par exemple, un composant qui sait gérer des prix.
 - On lui passe une liste de produits et il calcule le prix total.
 - Simple en apparence, car la gestion des prix peut devenir très complexe : remises, promotions, lots, clients privilégiés, règles complexes en fonction du pays, des taxes, etc...

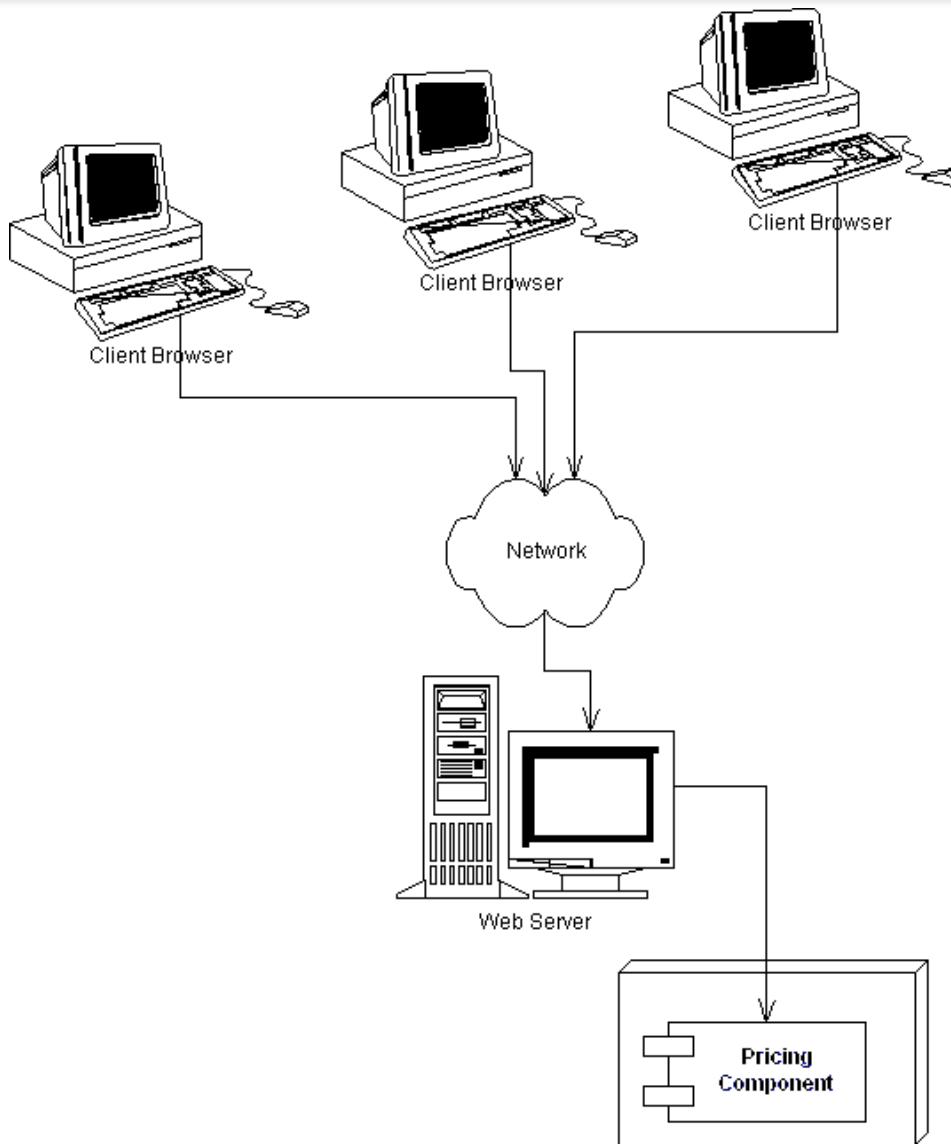
Composant logiciel réutilisable

- Ce composant répond à un besoin récurrent
 - Vente en ligne de matériel informatique,
 - Gestion des coûts sur une chaîne de production automobile,
 - Calcul des prix des expéditions par la poste,
 - Etc...

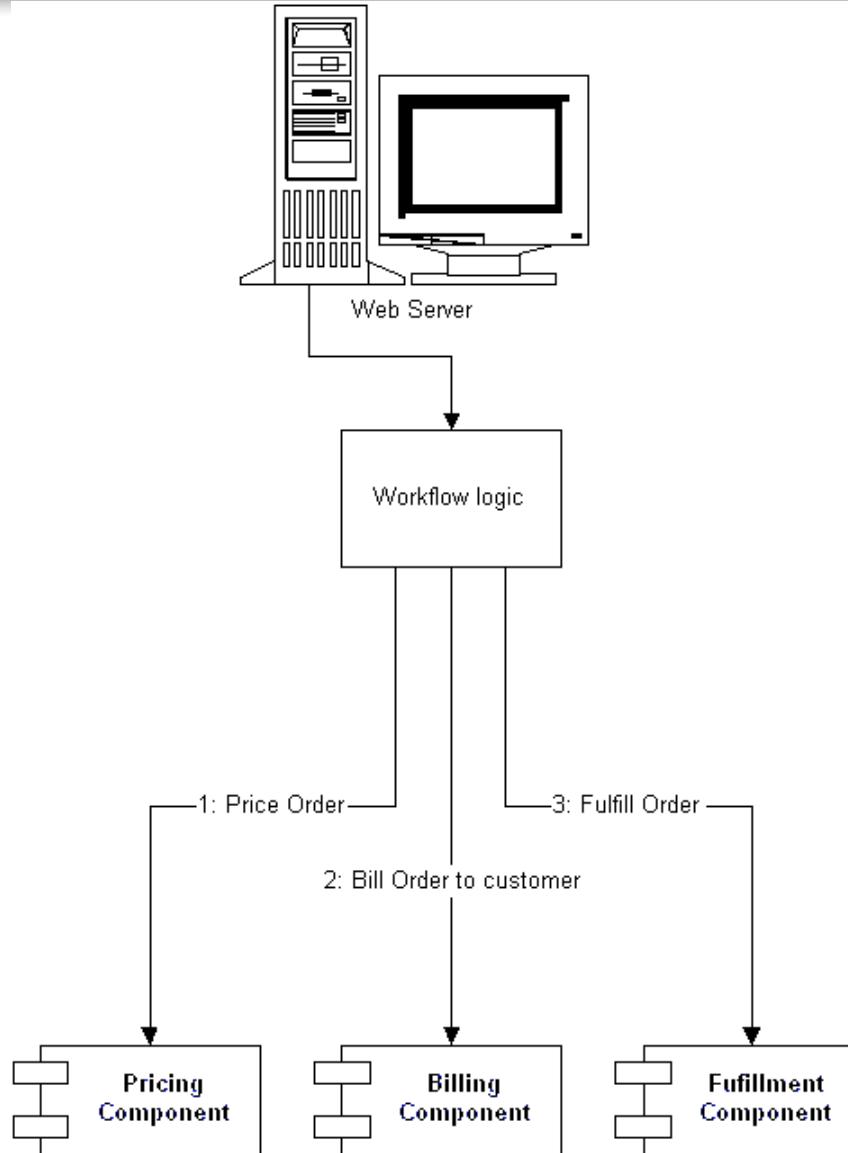
Composant logiciel réutilisable



Composant logiciel réutilisable



Composant logiciel réutilisable



Quel intérêt ?

- Moins d'expertise requise pour répondre à certains points du cahier des charges,
- Développement plus rapide.
- Normalement, les vendeurs de composants assurent un service de qualité (BEA, IBM...)
- Réduction des frais de maintenance.
- Naissance d'un marché des composants.
 - Pas encore l'explosion attendue mais...

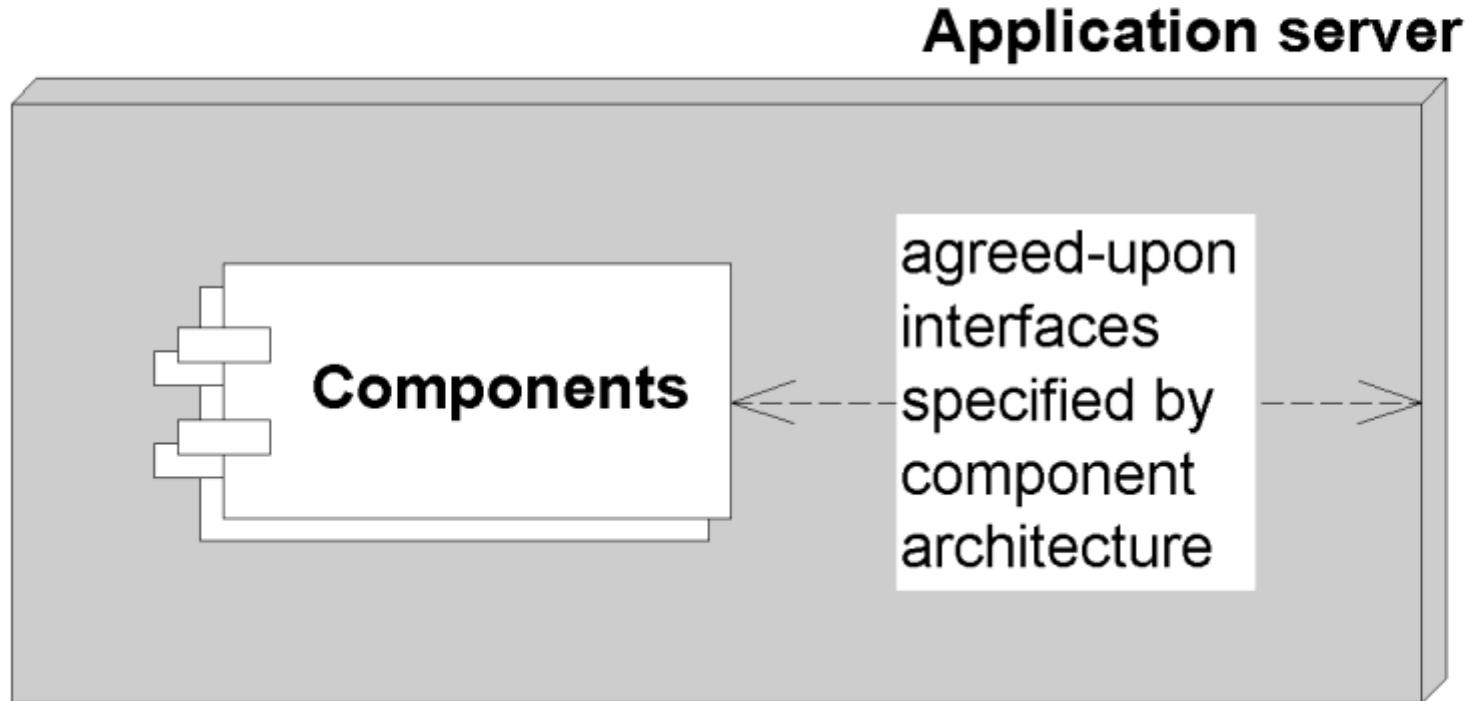
Architectures de composants

- Plus de 50 serveurs d'applications ont vu le jour depuis une dizaine d'années,
- Au début, composants propriétaires uniquement.
 - Pas de cohabitation entre composants développés pour différents serveurs d'application
 - Dépendant d'un fabricant une fois le choix effectué.
- Dur à avaler pour les développeurs java qui prônent la portabilité et l'ouverture !

Architectures de composants

- Nécessité de standardiser la notion de composants
 - Ensemble de définitions d'interfaces entre le serveur d'application et les composants
 - Ainsi n'importe quel composant peut tourner ou être recompilé sur n'importe quel serveur
- Un tel standard s'appelle *une architecture de composants*
 - Penser aux CDs audio, à la télé, au VHS, etc...

Architectures de composants



Enterprise JavaBeans (EJB)

- Le standard EJB est une architecture de composants pour des composants serveur écrits en java.
 1. Adopté par l'industrie. "*Train once, code anywhere*"
 2. Portable facilement
 3. Rapid Application Development (RAD)
- EJB signifie deux choses :
 1. Une spécification
 2. Un ensemble d'interfaces

Pourquoi java ?

■ EJB = uniquement en java

- Séparation entre *l'implémentation* et *l'interface*
- Robuste et sûr : mécanismes + riche API + spécificité du langage (reflexivité, introspection, chargement dynamique)
- Portable

■ Autre possibilités

- Composants Microsoft .NET
- Ruby on rails, Python turbo gears, frameworks java plus légers comme WebWork, Spring, etc.

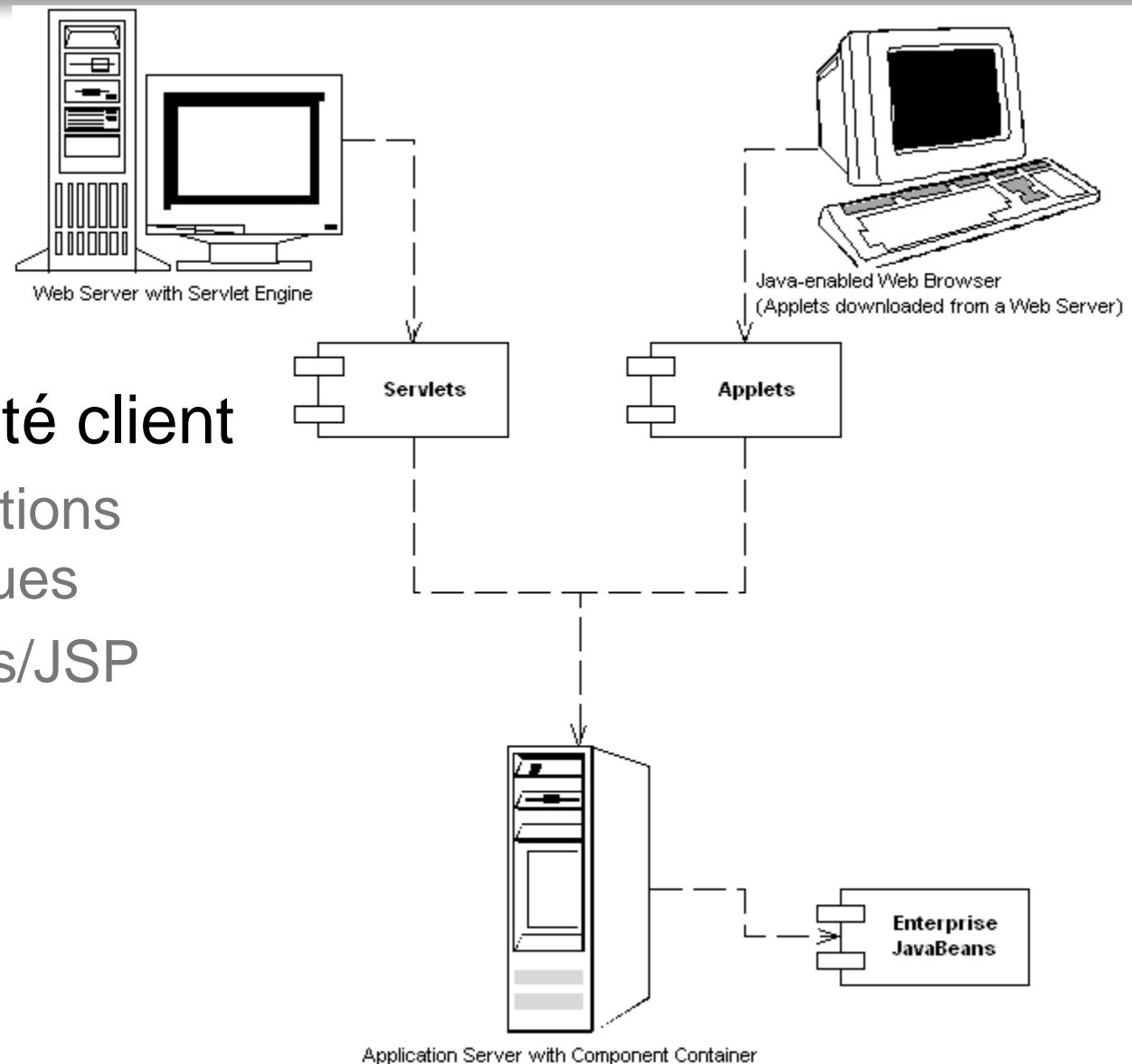
EJB pour développer des composants business

- Implémenter de la logique métier : calcul des taxes sur un ensemble d'achats, envoyer un mail de confirmation après une commande, etc...
- Accéder à un SGBD
- Accéder à un autre système d'information (CICS, COBOL, SAP R/3, etc...)
- Applications web : intégration avec JSP/Servlets
- Web services basés sur XML (SOAP, UDDI, etc...)
 - Exemple : DELL attaque le serveur d'INTEL directement à travers un protocole XML pour réserver des pièces.

EJB ne fournit pas de GUI

■ GUI = côté client

- Applications classiques
- Servlets/JSP



L'écosystème EJB

- Pour déployer et exécuter un projet à base d'EJBs, six métiers sont impliqués

1 - Le fournisseur d'EJBs

- Peut-être un membre de votre équipe, ou bien une entreprise qui vend des EJBs
(www.componentsource.com ou
www.flashline.com pour voir une liste)

L'écosystème EJB

2 - L'assembleur d'application

- Il s'agit de l'architecte de l'application
- Il est client des EJBs achetées ou développées
- Il décide de la combinaison de composants dont il a besoin
- Fournit un GUI à l'application
- Conçoit et développe de nouveau composants
- Conçoit et développe les programmes clients
- Définit le mapping avec les données manipulées par les différents composants
- En général, c'est un expert en Génie Logiciel, en UML et en développement Java.
- Il peut s'agir d'un intégrateur de systèmes, d'un consultant, d'une équipe de développeurs/concepteurs maison...

L'écosystème EJB

3 - Le déployeur d'EJBs

- Après que l'application ait été assemblée, elle doit être déployée sur un ou plusieurs serveurs d'application
- Attention à la sécurité (firewall, etc...)
- Branchement de services annexes (LDAP, Lotus Notes, Microsoft Active Directory, etc...) sur le serveur d'applications.
- Choix du hardware, des SGBD, etc...
- Paramétrage du serveur d'application, optimisation des performances...
- Il *adapte* les composants et le serveur à l'application
- Il peut être une équipe ou une personne, un consultant ou un vendeur d'hébergement de serveurs d'applications.
- Exemples aux USA : www.hostJ2EE.com ou www.loudcloud.com

L'écosystème EJB

4 - L'administrateur système

- Vérifie le bon fonctionnement de l'application en exploitation.
- Il utilise les outils de monitoring des serveurs d'application.
- Il effectue la maintenance hardware et software (lancement, arrêt) du système.
- Certains serveurs d'application savent téléphoner et appeler l'administrateur système en cas de problème.
 - Ponts avec les outils de Tivoli, Computer Associates, ... via JMX.

L'écosystème EJB

5 - Le fournisseur du serveur d'application et des *containers*

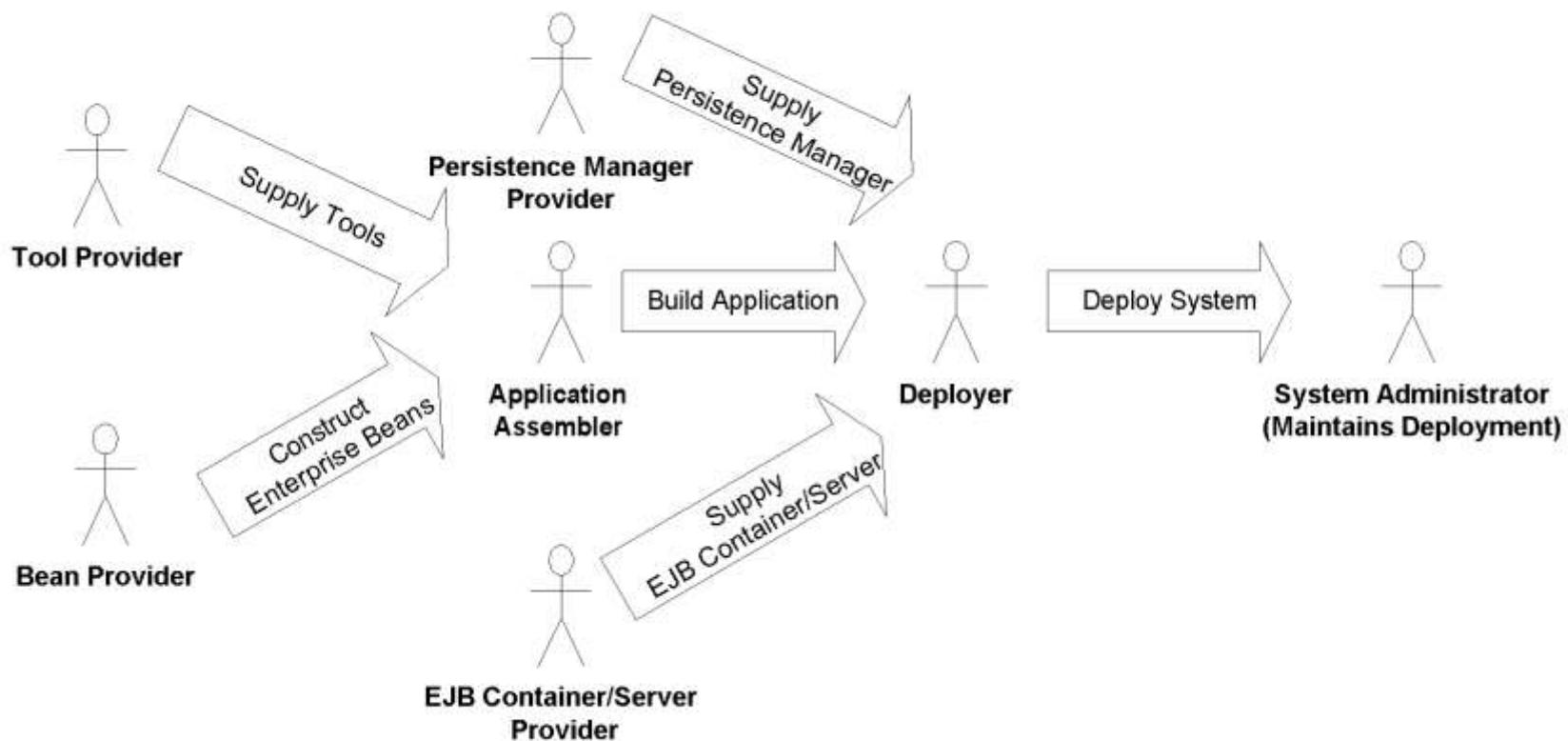
- EJB container = environnement au sein du serveur dans lequel les EJB vivent.
- Le container EJB fournit les services middleware et manage les EJBs.
- Exemples : Weblogic, Websphere, BEA, Oracle Orion Server, JRun, JBoss...
- Il existe d'autres containers spécialisés (container Web comme Tomcat, Resin...)
- Le fournisseur du serveur d'application est le même que le fournisseur de containers EJB.
- On confond en général container et serveur d'application.

L'écosystème EJB

6 - Les vendeurs d'outils

- Développer une application à base d'EJB est assez lourd. Pourtant la manière de développer, construire, maintenir, déployer les EJBs est standard.
- Il est très pratique d'utiliser un IDE pour simplifier les tâches répétitives comme le déploiement, etc...
- Principaux IDEs : JBuilder, Visual Age, Visual Cafe.
- Autres outils : les outils UML comme Together/J, Rational Rose
- Outil de test (JUnit), de stress (LodeRunner), etc...

Les différents métiers...



Les différents métiers...

- Bientôt un nouveau métier : le "persistence manager"
 - Développe des outils qui se "branchent" sur le serveur d'application et implémentent les mécanismes de persistance.
 - Mapping BD relationnelles/Objets
 - Mapping BD objet/Objets
 - Etc...
- Pas encore standardisé dans la spécification EJB 2.0

La plate-forme Java J2EE

- EJB = la clé de voûte d'une architecture distribuée java appelée J2EE
 - Pour le développement d'application serveur
 - Ensemble de spécifications et d'APIs
 - Contient deux autres architectures
 1. J2ME (Java 2 Micro Edition) : pour les mobiles
 2. J2SE : pour applications et applets standards
 - Non attaché à un vendeur particulier.
 - Quiconque respecte les spécification est "J2EE compliant"
 - Applications développées indépendantes des vendeurs d'outils.

La plate-forme Java J2EE

- Chaque API dans J2EE à sa propre spécification (PDF)
- Batterie de logiciel pour valider les implémentations (test suites)
- Implémentation de référence
 - J2EE est livrée avec un serveur d'application par exemple...
- Ensemble de "blueprints" : définit précisément le rôle de chaque API (PDF)

J2EE : les APIs

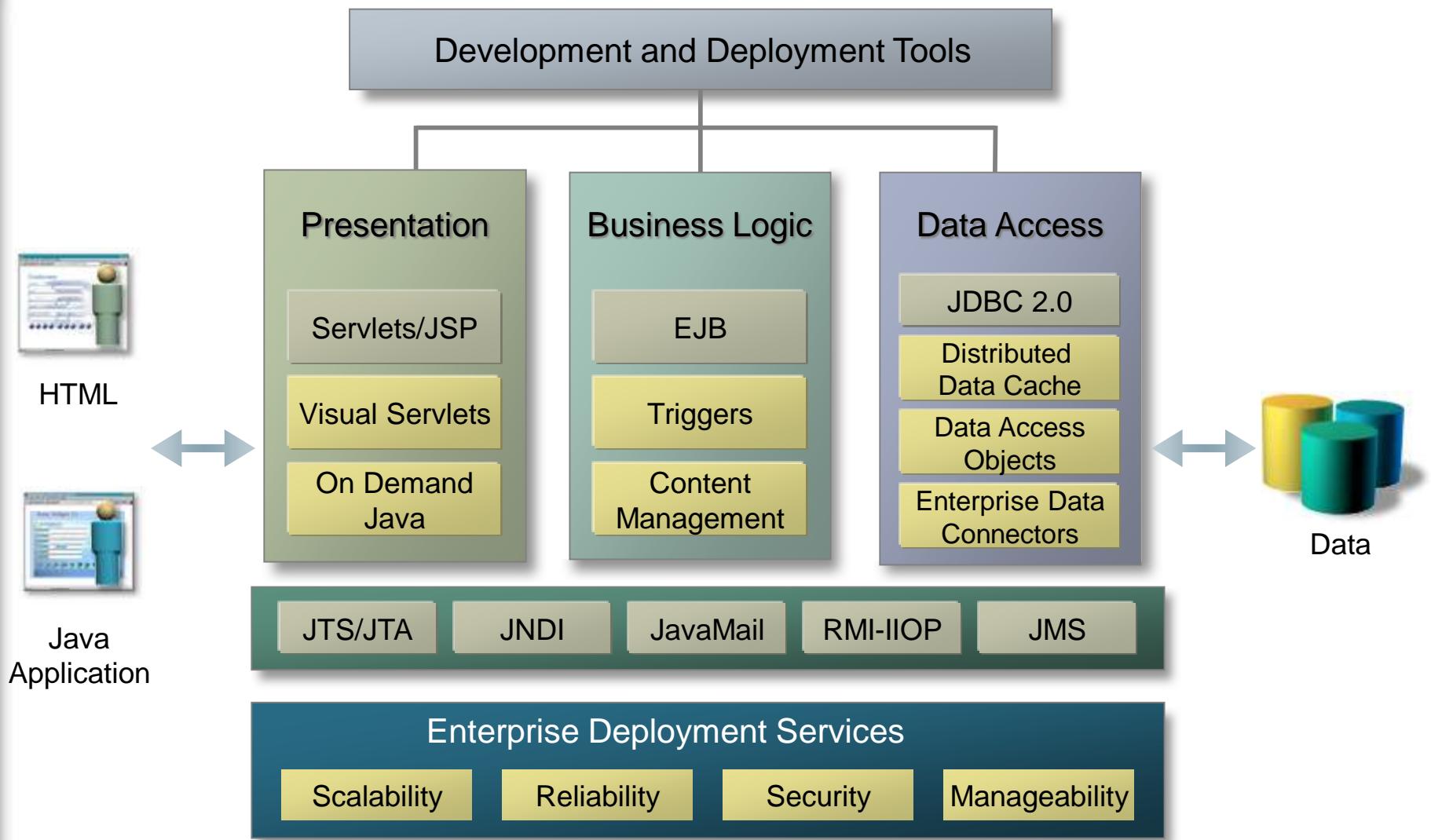
■ J2EE comprend en plus de J2ME et J2SE

- EJB : standard de définition de composants
- Java 2 RMI et RMI-IIOP : objets distribués
- JNDI (Java Naming and Directory Interface)
- JDBC (Java Data Base Connectivity)
- JTA (Java Transaction API)
- JMS (Java Messaging Service)
- Java Servlets and Java Pages (JSP)
- Java IDL (Corba)
- JavaMail
- JCA (J2EE Connector Architecture) : ponts vers SAP/3, CICS...
- JAXP (Java API for XML Parsing)
- JAAS (Java Authentification and Authorization Service)

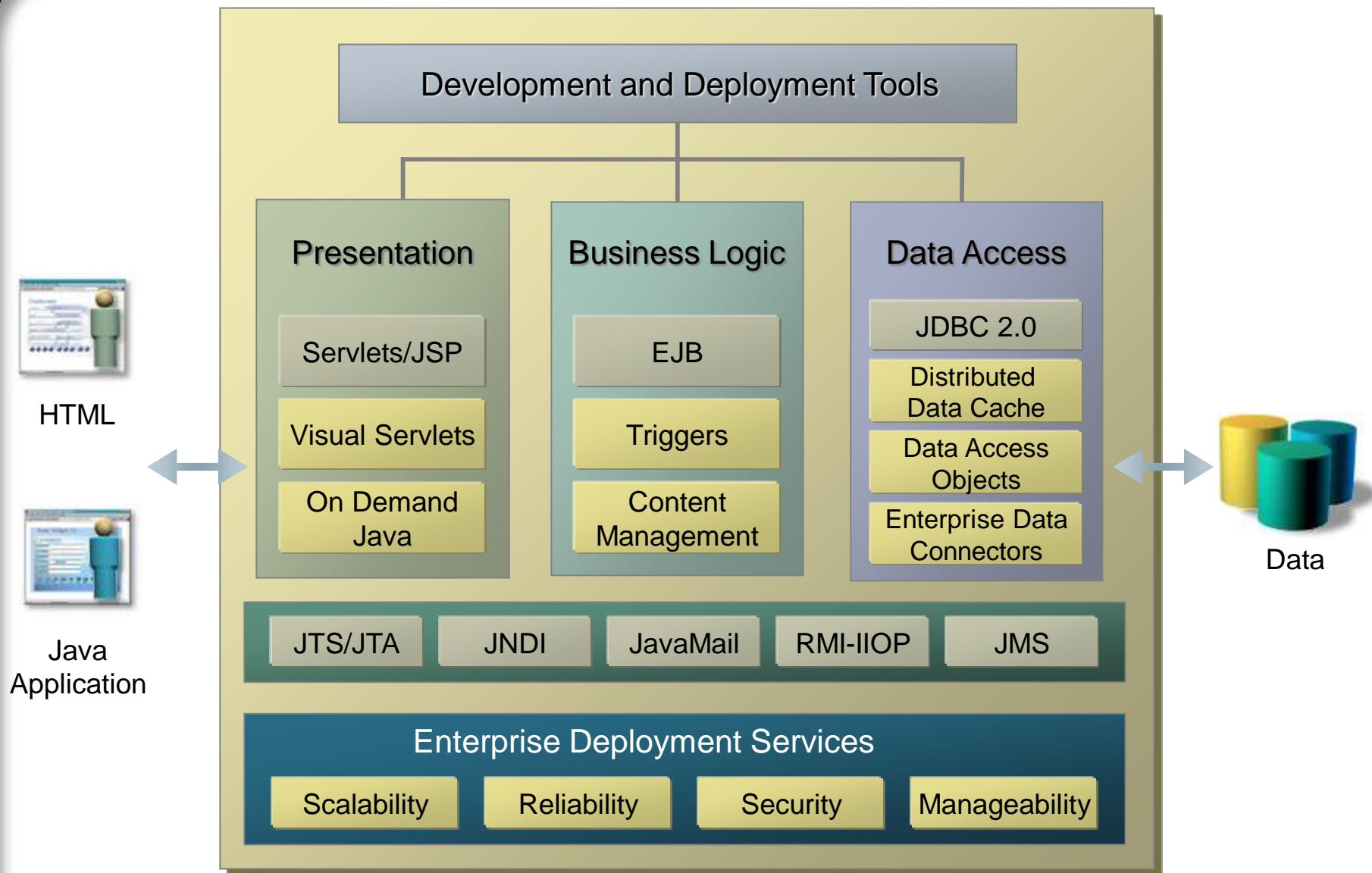
- J2EE standardizes interfaces for key Application Server components

**"J2EE will be
like SQL for
Application Servers"**

J2EE for the Real World



Consistent, Integrated Architecture



EJB : les fondamentaux

Enterprise Bean

- Composant serveur qui peut être *déployé*
- Composé de *un ou plusieurs objets*
- Les clients d'un Bean lui parlent *au travers d'une interface*
- Cette interface, de même que le Bean, suivent la spécification EJB
- Cette spécification requiert que le Bean expose *certaines méthodes*

Enterprise Bean

- Le client d'un Bean peut être
 - Une servlet
 - Une applet
 - Une application classique
 - Un autre Bean
- On peut décomposer une application en un graphe de tâches/sous-tâches
- Exemple : achat d'un CD à partir du code-barre
 - Scanner (qui a une JVM embarquée) client d'un Bean sur le Serveur
 - Ce Bean client d'autres Beans : gestion de catalogue, de commandes, de gestion de transaction VISA, etc...
- Modèle flexible, extensible...

3 types de Beans : Session Bean

■ Session Beans

- Modèlissent un traitement (business process)
- Correspondent à des *verbes*, à des *actions*
- Ex : gestion de compte bancaire, affichage de catalogue de produit, vérifieur de données bancaires, gestionnaire de prix...
- Les actions impliquent des calculs, des accès à une base de données, consulter un service externe (appel téléphonique, etc.)

■ Souvent clients d'autres Beans

3 types de Beans : Entity Bean

■ Entity beans

- Modèlissent des données
- Correspondent à des *noms*
- Ce sont des objets java qui *cachent* des données d'une base de données
- Ce sont des objets persistants
- Ex : un Bean Personne, un Bean compte bancaire, un Bean produit, un Bean commande.
- Serveurs pour des Beans Session le plus souvent
- Servent de proxy entre la logique métier et les base de données
- Mapping base de donnée relationnelle/Objet facilité par EJB 2.0

Exemple de Session/Entity bean

Session Bean	Entity Bean
Gestion de compte	Compte bancaire
Vérificateur de CB	Carte de crédit
Système d'entrée gestion de commandes	Commande, ligne de commande
Gestion de catalogue	Produits
Gestionnaire d'enchères	Enchère, Produit
Gestion d'achats	Commande, Produit, ligne de commande

3 types de Beans : Message-Driven Bean

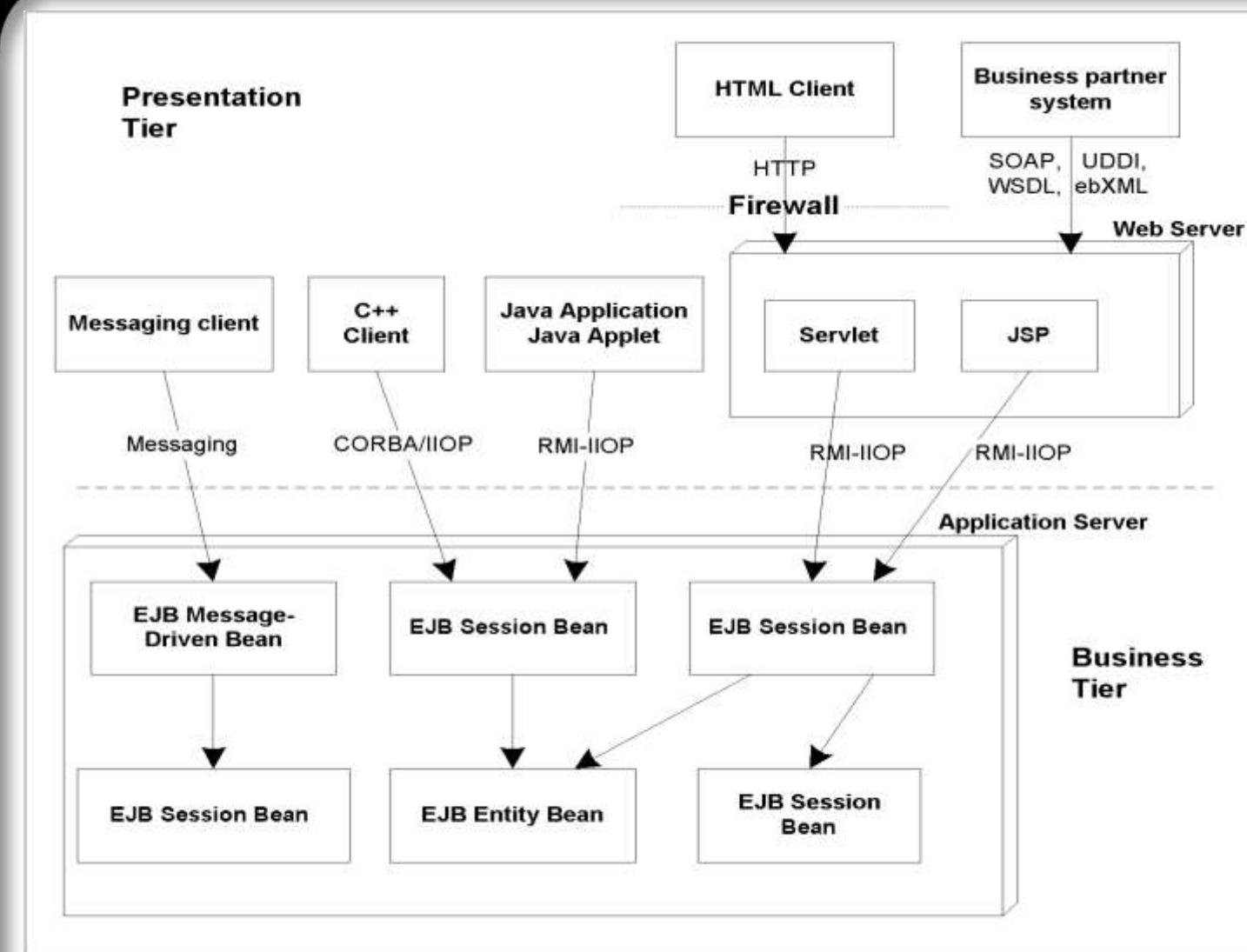
■ Message-Driven Beans

- Introduits à partir de la norme EJB 2.0, nous sommes aujourd'hui en 3.0
- Similaire aux Session bean : représentent des verbes ou des actions,
- On les invoque en leur envoyant des messages,
- Ex : message pour déclencher des transactions boursières, des autorisations d'achat par CB,
- Souvent clients d'autres beans...

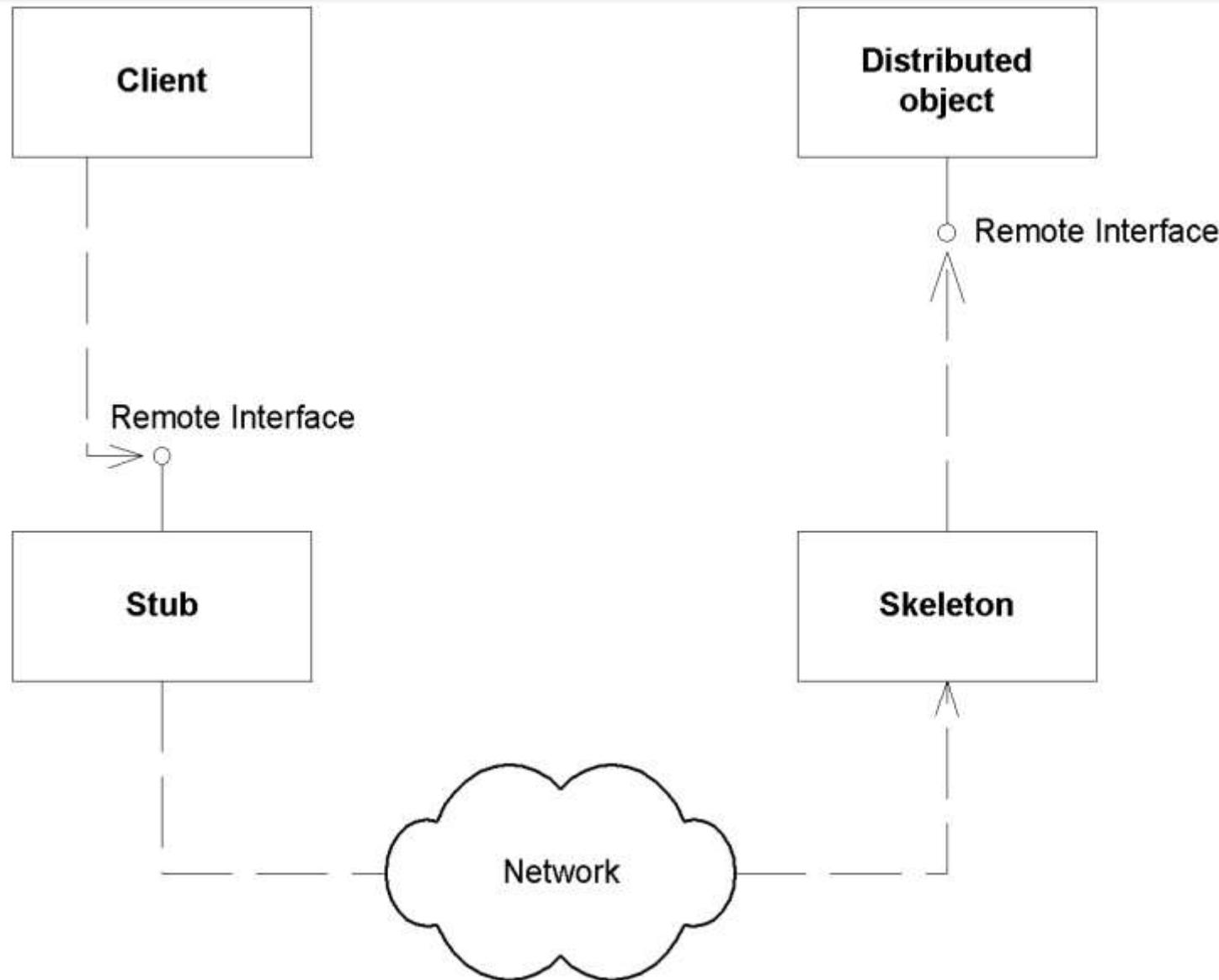
3 types de Beans : pourquoi ?

- Pas d'Entity Beans dans les solutions Microsoft par exemple...
- Nombreuses compagnies impliquées dans les standards EJB/J2EE
 - Leurs clients ont des besoins variés,
 - Solution proposée flexible mais plus complexe,
 - Standard EJB plus difficile à apprendre,
 - Risque de mauvaise utilisation mais...
 - On est gagnant sur le long terme.

Clients interagissant avec un serveur à base d'EJBs



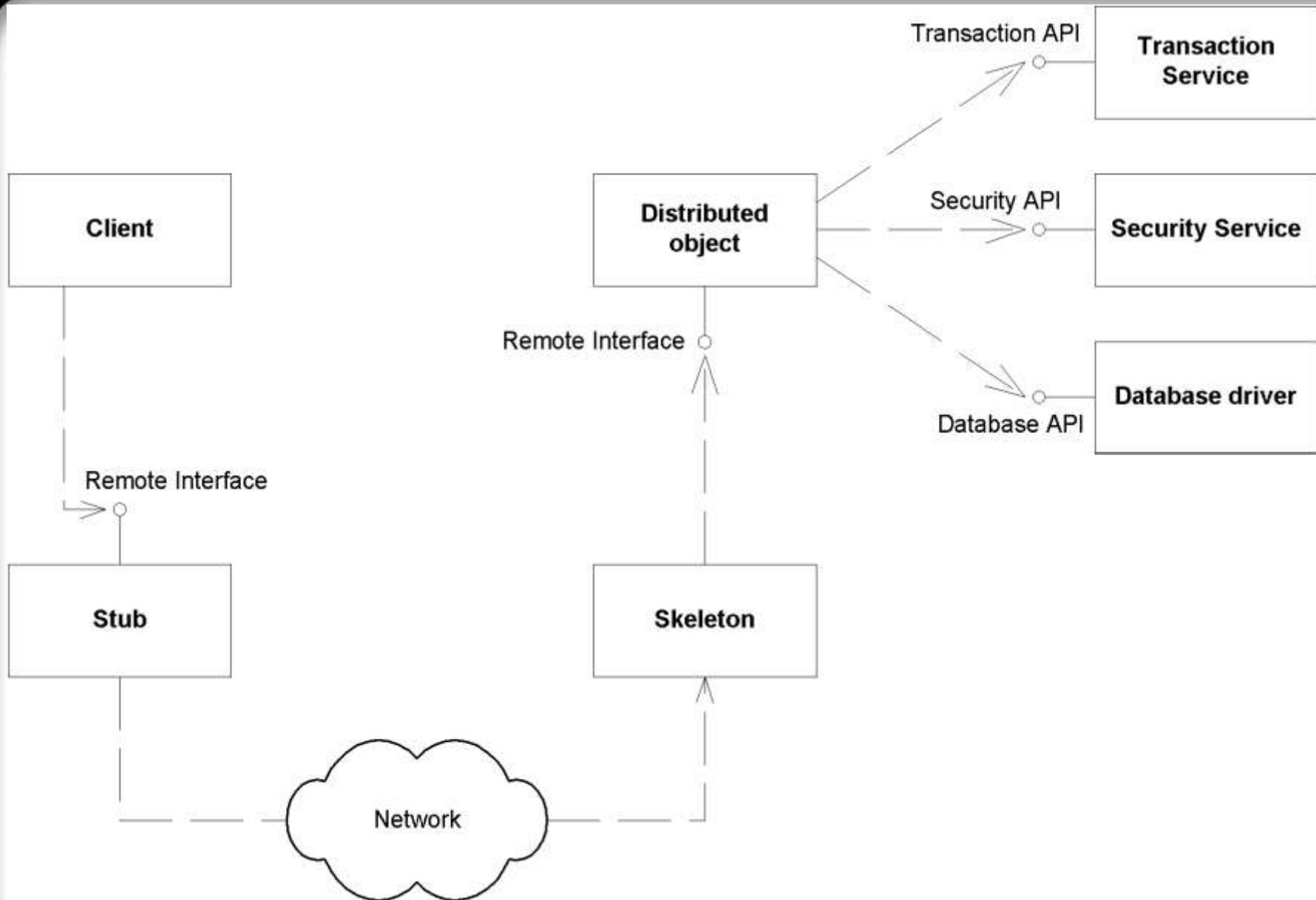
Les objets distribués au cœur des EJBs



Les objets distribués et le middleware

- Lorsqu'une application devient importante, des besoins récurrents apparaissent : sécurité, transactions,etc...
- C'est là qu'intervient le *middleware!*
- Deux approches
 1. Middleware explicite,
 2. Middleware implicite

Middleware explicite



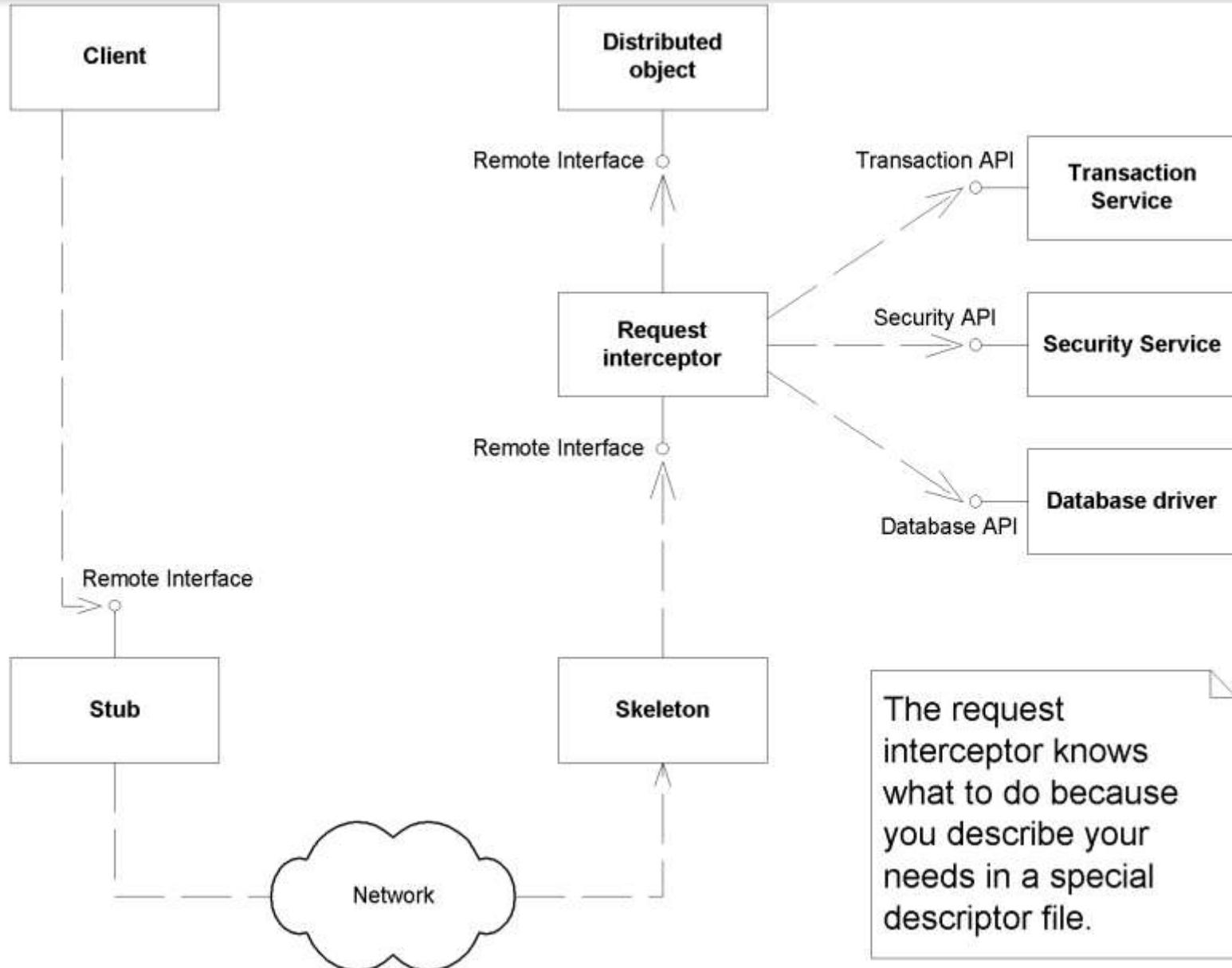
Middleware explicite

- Exemple : transfert d'un compte bancaire vers un autre :
- **transfert(Compte c1, Compte c2, long montant)**
 1. Appel vers l'API middleware qui fait une vérification de sécurité,
 2. Appel vers l'API de transaction pour démarrer une transaction,
 3. Appel vers l'API pour lire des lignes dans des tables d'une BD,
 4. Faire le calcul : enlever de l'argent d'un compte pour le mettre dans l'autre
 5. Appeler l'API pour mettre à jour les lignes dans les tables,
 6. Appeler l'API de transaction pour terminer la transaction.

Middleware explicite

- Difficile à écrire,
- Difficile à maintenir,
- Votre code est dépendant des API du vendeur de middleware que vous utilisez.

Middleware implicite



Les EJB : middleware implicite mais API pour descendre au bas niveau, Explicite

- La plupart du temps le développeur demeure au niveau implicite,
- Mais il peut, même si le travail est plus complexe, utiliser des APIs de J2EE pour contrôler «manuellement » les transactions, la sécurité, etc.

EJB et SOA (Service Oriented Architecture)

- Une application = ensemble de services,
- Un service = ensemble de composants,
- Un composant = des classes,
- Les services peuvent tourner sur des « nœuds » différents (des cpus différents) et donc former une architecture distribuée
- Avantage : souplesse, mise à jour, etc... Le code ne change pas qu'on soit en mono-cpu ou en distribué.

EJB et SOA (Service Oriented Architecture)

- SOA = un paradigme de conception
 - Service = indépendant, faiblement couplé aux autres
- Les Web Service = un exemple de SOA, mais il y en a d'autres.
- Par exemple des services fait d'EJBs, les services du Robotic Studio de Microsoft, etc.

**Avant les EJB 3.0 étaient les EJB 2.x
et... c'était puissant mais trop
compliqué !**

EJB 2.0 : constitution d'un bean, les principes sont les mêmes en 3.0 sauf que l'on a pas à écrire autant de code

Constitution d'un EJB : *Enterprise Bean class*

■ La classe du Bean (Enterprise Bean class)

- Une classe qui implémente une interface précise et qui respecte certaines règles,
- Il s'agit de l'implémentation du bean lui-même,
 - **Session Bean** : logique métier, calculs, transfert de compte bancaire, saisie de commandes, etc...
 - **Entity Bean** : logique orientée donnée, par exemple comment changer le nom d'un client, diminuer un compte bancaire...
 - **Message-Driven Bean** : logique orientée message, traitement après réception d'un ordre d'achat d'actions boursières...

Constitution d'un EJB : *EJB Object*

- Les clients n'invoquent jamais directement les méthodes de la classe du Bean
- Les appels de méthodes (requests) sont interceptés par le Container, afin d'assurer le traitement middleware implicite,
- Une fois ce traitement effectué, le container appelle les méthodes de la classe du Bean
- Le développeur de composant se concentre sur la logique, ignore la partie middleware.

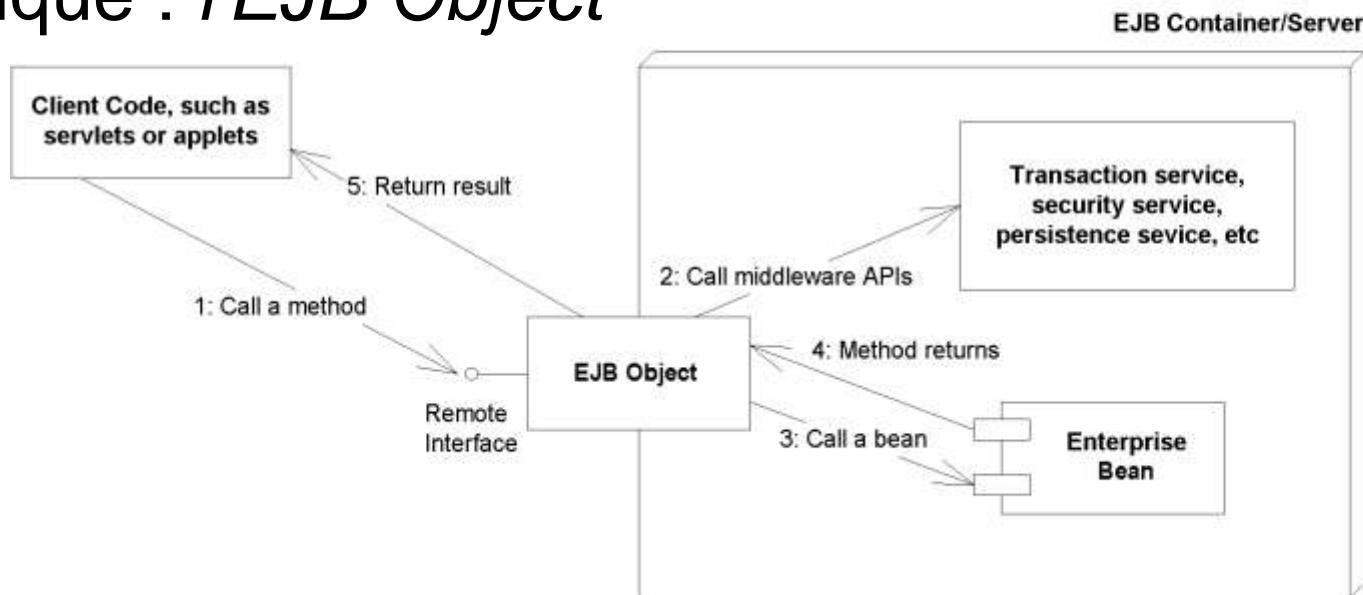
Constitution d'un EJB : *EJB Object*

■ Que se passe-t-il lors de l'interception ?

- Prise en compte des transactions,
- Sécurité : le client est-il autorisé ?
- Gestion des ressources + cycle de vie des composants : threads, sockets, connexions DB, pooling des instances (mémoire),
- Persistance,
- Accès distant aux objets,
- Threading des clients en attente,
- Clustering,
- Monitoring : statistiques, graphiques temps réel du comportement du système...
- ...

Constitution d'un EJB : *EJB Object*

- Container = couche d'indirection entre le client et le bean
- Cette couche est matérialisée par un objet unique : *l'EJB Object*



Constitution d'un EJB : *EJB Object*

- *L'EJB Object* contient du code spécifique au container (vendeur-dépendant)
- Il appelle les méthodes de la classe du Bean,
- Il est généré par le container !
- Chaque container est livré avec des outils pour générer les *EJB Object* pour chaque Bean.

EJB : classe du Bean et EJB Object

EJ Bean

- Code simple

Container
EJB

- Génération du code à partir du Bean
- Le code généré fournit Transactions, Sécurité, Persistance, Accès Distant, gestion des ressources, etc.

Serveur
EJB

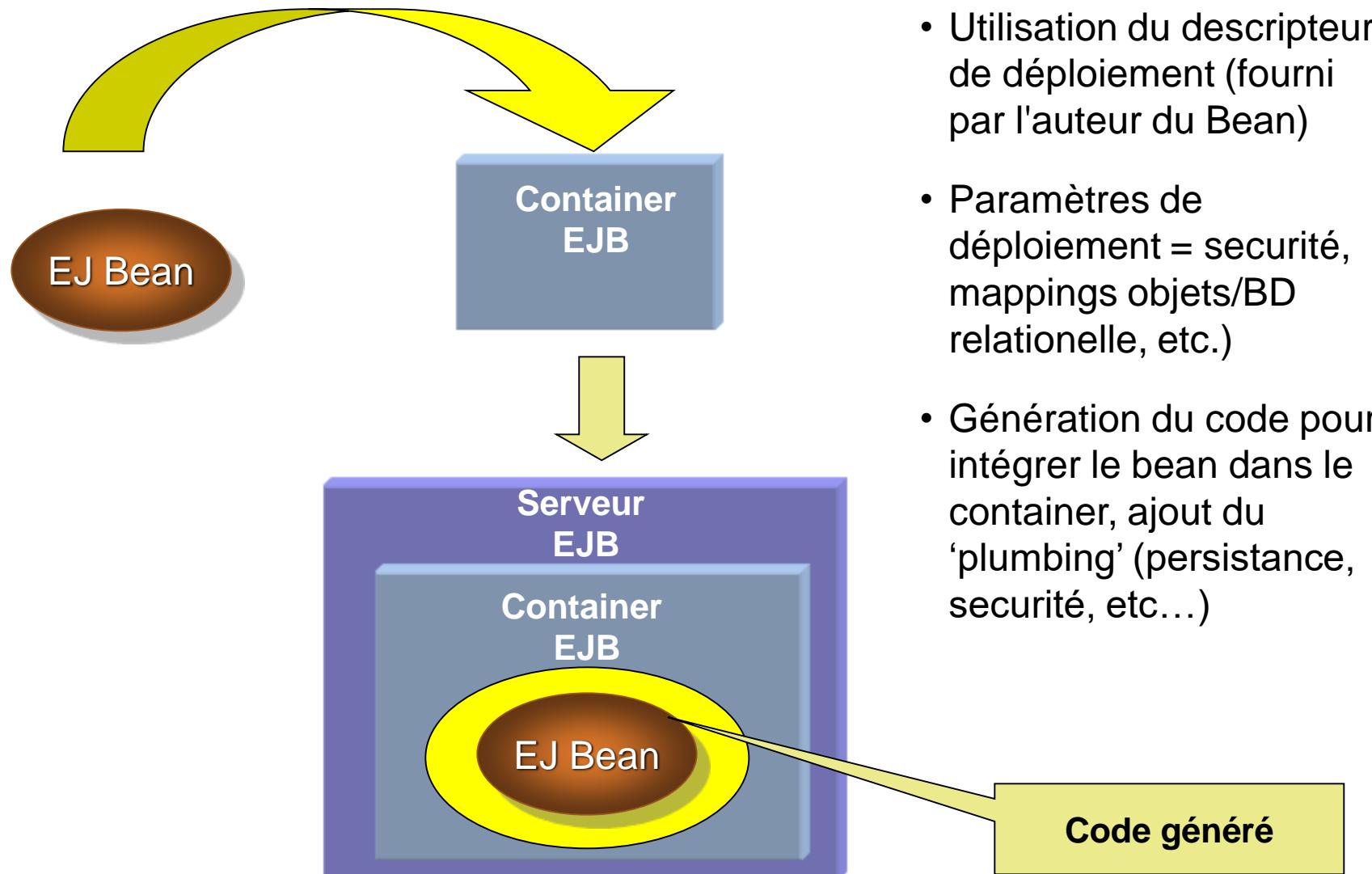
- Fournit les services au container

EJB Server

EJB Container

EJ Bean

EJB Object : génération du code



Constitution d'un EJB : *l'interface distante*

- Les clients invoquent les méthodes des EJB Objects,
- Ces EJB Objects doivent *cloner* toutes les méthodes que le bean expose,
- Comment l'outil qui génère l'EJB Object peut-il connaître la liste des méthodes à cloner ?
- Réponse : il utilise une interface fournie par le programmeur du bean, *l'interface distante*

Constitution d'un EJB : *l'interface distante*

```
public interface javax.ejb.EJBObject extends java.rmi.Remote {  
    public abstract javax.ejb.EJBHome getEJBHome()  
        throws java.rmi.RemoteException;  
  
    public abstract java.lang.Object getPrimaryKey()  
        throws java.rmi.RemoteException;  
  
    public abstract void remove()  
        throws java.rmi.RemoteException,  
            javax.ejb.RemoveException;  
  
    public abstract javax.ejb.Handle getHandle()  
        throws java.rmi.RemoteException;  
  
    public abstract boolean isIdentical(javax.ejb.EJBObject)  
        throws java.rmi.RemoteException;  
}
```

Constitution d'un EJB : *l'interface distante*

- Le programmeur du Bean dérivera son interface distante de *javax.ejb.EJBObject*,
- Rajoutera les signatures des méthodes qu'il souhaite exposer,
- Qui implémentera cette interface ?
- L'EJB Object généré par le container !
- Presque rien à faire pour le développeur de bean !

Java RMI-IIOP et EJB Objects

- `Javax.ejb.EJBObject` dérive de `java.rmi.Remote`,
- Quiconque implémente `Remote` est appelable à distance depuis une autre JVM,
- EJB Objects = RMI-IIOP + EJB compatibles
- RMI-IIOP = convention de passage de paramètres + valeurs de retour lors d'appels de méthode distante, entre autres...

Constitution d'un EJB : *Home Object*

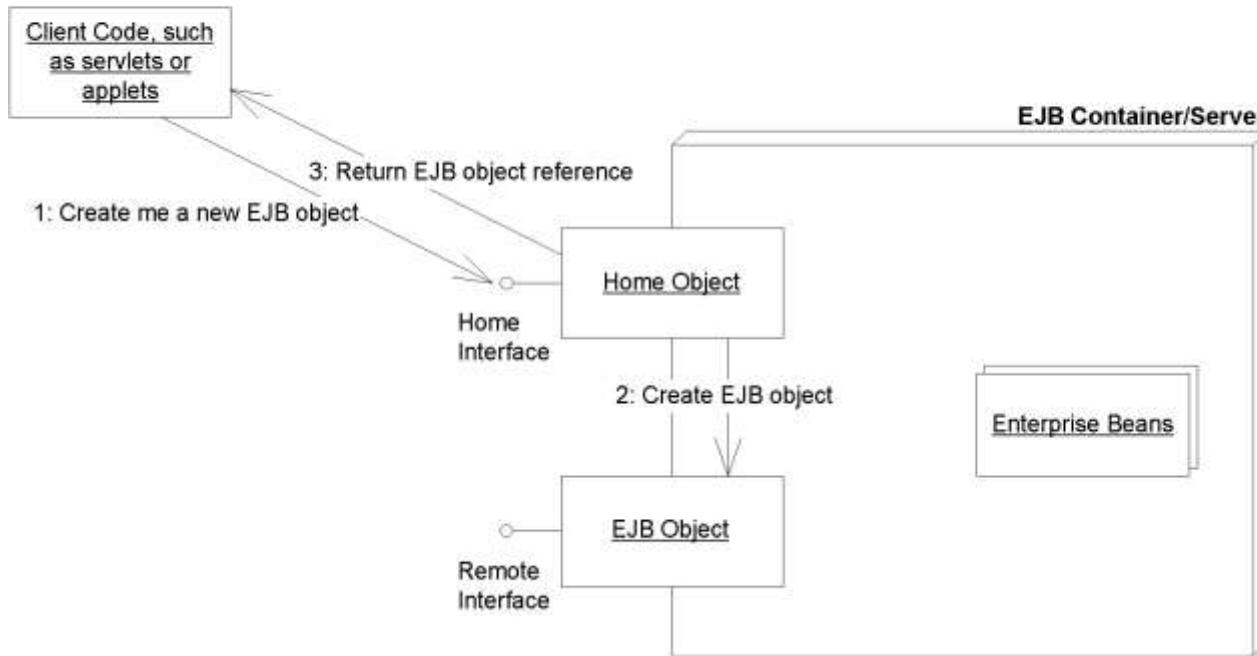
- Nous avons vu comment les clients appelaient les méthodes d'un Bean : via l'EJB Object.
- Mais comment obtiennent-ils *une référence* sur l'EJB Object ?
- En effet : pas d'instanciations lorsque on travaille avec des objets distants !
- Solution : le client demande la référence à une fabrique d'EJB Objects (*EJB Object Factory*)
 - Design pattern!

Constitution d'un EJB : *Home Object*

- L'EJB factory est responsable de l'instanciation et de la destruction des EJB Objects.
- La spécification EJB appelle cette factory un *Home Object*.
- Responsabilités du *Home Object*
 - Créer les EJB Objects,
 - Trouver les EJB Objects existants (Entity beans seulement)
 - Supprimer les EJB Objects.

Constitution d'un EJB : *Home Object*

- Comme les EJB Objects, les Home Objects sont générés par le container
 - Contiennent du code spécifique,
 - Assurent le load-balancing, etc...



Constitution d'un EJB : *l'interface Home*

- Comment un *Home object* sait de quelle manière le client veut initialiser l'EJB Object ?
 - Rappel : au lieu d'appeler un constructeur, on demande au Home object de retourner une référence.
 - Comme pour les constructeurs, il est possible de passer des paramètres d'initialisation.
- Comme pour les EJB Objects, le développeur du bean doit spécifier une *interface Home*

Constitution d'un EJB : *l'interface Home*

■ *L'interface Home* définit

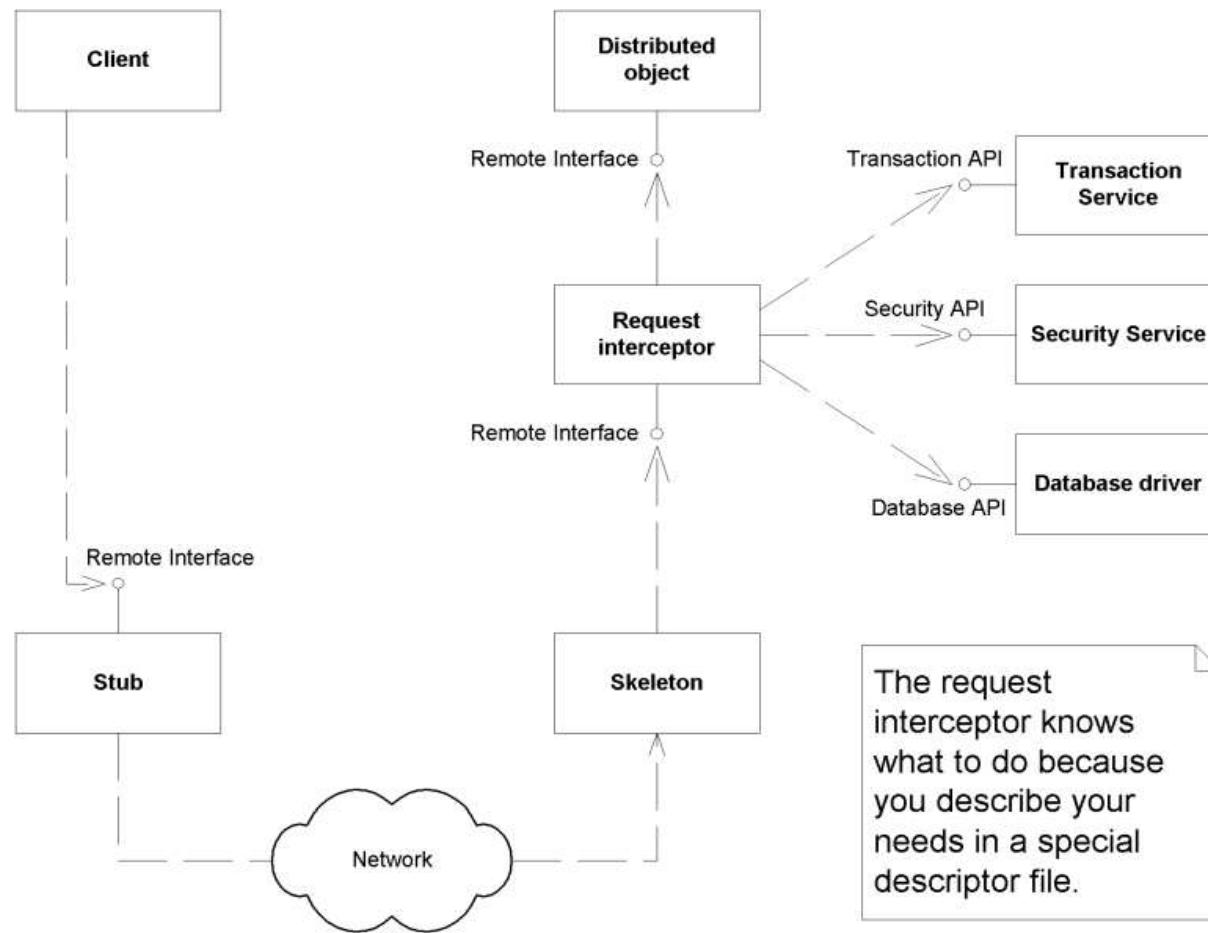
- Les méthodes pour créer, détruire et localiser les EJB Objects
- Le Home Object (généré par le container) implémente cette interface.
- L'interface fournie par le développeur dérive de *javax.ejb.EJBHome*
- *Javax.ejb.EJBHome* dérive de *java.rmi.Remote*
- Les *Home objects* sont aussi des objets distants, compatibles RMI-IIOP

Constitution d'un EJB : *l'interface Home*

```
public interface javax.ejb.EJBHome extends java.rmi.Remote {  
    public EJBMetaData getEJBMetaData()  
        throws java.rmi.RemoteException;  
  
    public javax.ejb.HomeHandle getHomeHandle()  
        throws java.rmi.RemoteException;  
  
    public void remove(Handle handle)  
        throws java.rmi.RemoteException  
            javax.ejb.RemoveException;  
  
    public void remove(Object primaryKey)  
        throws java.rmi.RemoteException,  
            javax.ejb.RemoveException;  
}
```

Constitution d'un EJB : les *interfaces locales*

- Problème : la création de bean et l'appel de méthode distante coûtent cher !



Constitution d'un EJB : *les interfaces locales*

■ Commentaires sur la figure précédente

1. Le client appelle un *stub* (*souche*),
2. Le *stub* encode les paramètres dans un format capable de voyager sur le réseau,
3. Le *stub* ouvre une connexion sur le *skeleton* (*squelette*),
4. Le *skeleton* décode les paramètres,
5. Le *skeleton* appelle *l'EJB Object*,
6. *L'EJB Object* effectue les appels *middleware*,
7. *L'EJB Object* appelle la méthode du bean,
8. Le Bean fait son travail,
9. On fait le chemin inverse pour retourner la valeur de retour vers le client !
10. ...Sans compter le chargement dynamique des classes nécessaires !

Constitution d'un EJB : *les interfaces locales*

- Nouveau dans EJB 2.0 : les interfaces locales.
- Introduit la notion de *Local Object*, en remplacement de *EJB Object*
- Les *Local Objects* implémentent une *interface locale* au lieu d'une *interface distante*
- Exemple d'appel de méthode distante
 1. Le client appelle le *Local Object*,
 2. Le *Local Object* appelle le *middleware* puis la méthode du bean
 3. La valeur de retour est renvoyée au *Local Object*, puis au client

Constitution d'un EJB : les *interfaces locales*

- Pour l'appel distant de méthodes, le développeur peut fournir une *interface locale*, qui sera implémentée par le container en tant que *Local Object*
 - A distinguer du cas "normal" (EJB 1.1) où on a *interface distante* implémentée par *EJB Object*
- Pour la création/localisation de beans, le développeur peut fournir une interface *home interface locale*, qui sera implémentée par le container en tant que *Home Object local*
 - A comparer avec *Home Interface* implémentée par le container en tant que *Home Object*

Constitution d'un EJB : *les interfaces locales*

- Les interfaces locales ne tournent que si les EJB sont dans le même processus (même container),
- Attention, paramètres et valeurs de retour (lors des appels de méthodes) se passent maintenant *par référence*
 - *Toujours par valeur dans le cas d'appels distants!*
 - *Plus performant mais sémantique différente.*
- Difficile de passer d'une implémentation locale à une implémentation classique
 - *Aurait pu être fait dans les descripteurs (weblogic)*

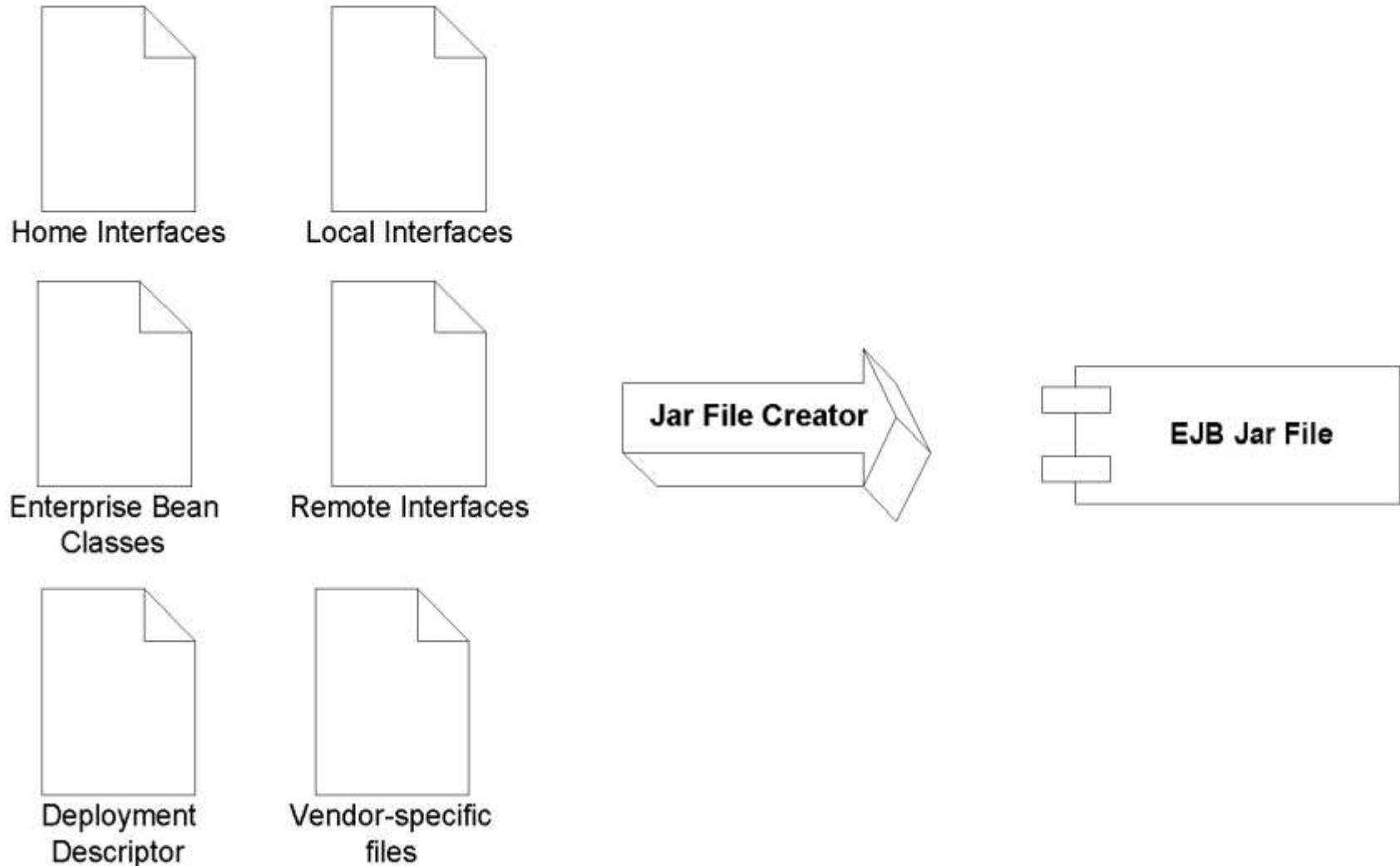
Constitution d'un EJB : *les descripteurs de déploiement*

- Pour informer le container des besoins middleware, on utilise *un descripteur de déploiement (XML)*
 - Standardisé,
 - A l'extérieur de l'implémentation du bean.
 - Attention si on les écrit à la main!
 - Outils d'aide au déploiement : IDEs (Jbuilder, Visual Cafe), outils spécifiques (Borland Application Server, Weblogic 6.1)
 - Descripteurs peuvent être modifiés après le déploiement.

Constitution d'un EJB : *les descripteurs de déploiement*

- Descripteurs spécifiques au serveur d'application
 - Chaque vendeur ajoute des trucs en plus : load-balancing, persistance complexe, clustering, monitoring...
 - Dans des fichiers spécifiques (inprise-ejb.xml avec Borland)

Déploiement : un fichier .jar



Résumé

- Enterprise Bean class
- Interface distante (remote interface)/Interface locale
- EJB Object/Local Object
- Home interface/Local Home interface
- Home Object/Local Home Object
- Descripteur de déploiement standard
- Descripteurs spécifiques au vendeur
- Fichier .jar de l'EJB

OUF !

Le modèle EJB 3.0 a fait le ménage !

Qu'est-ce qui a changé ?

- Pas grand-chose sur le fond, les EJBs adressent toujours le même problème,
- Beaucoup sur la forme.
 - Pour les Session Beans et pour les Message Driven Beans : modèle POJO (Plain Old Java Object)
 - Les développeurs veulent écrire du Java, pas du XML et pas 50 fichiers pour faire HelloWorld !
 - Pour les Entity Beans, on utilisera les « annotations de code » pour indiquer le mapping entre la classe java et les données dans la BD
- Le modèle EJB 3.0 utilise beaucoup les « annotations de code » introduites avec java 1.5

Introduction aux Session Beans

Session Bean : rappel

■ Un Session Bean représente

- une action, un verbe,
- une logique métier, un algorithme,
- Un enchaînement de tâches...

■ Exemples

- Saisie d'une commande,
- Compression vidéo,
- Gestion d'un caddy, d'un catalogue de produits,
- Transactions bancaires...

Durée de vie d'un Session Bean

- Durée de vie = *la session*
 - En gros, le temps qu'un client reste connecté sur le bean.
 - Dans une logique e-commerce, le temps qu'un utilisateur est connecté sur le site.
- Le container crée une instance lorsqu'un client se connecte sur le Session Bean.
- Il peut la détruire lorsque le client se déconnecte.
- Les Session Beans ne résistent pas à des crashes du serveur.
 - Ce sont des objets en mémoire, non persistants.
 - Le contraire des Entity Beans que nous verrons plus tard.

Types de Session Beans

- Chaque EJB a un moment donné entretient une *conversation* avec un client.
 - Conversation = suites d'appels de méthodes.
- Il existe deux types de Session Beans
 1. *Stateful Session Beans*,
 2. *Stateless Session Beans*.
- Chacun modélisant un type particulier de *conversation*.

Stateful Session Beans

- Certaines conversations se déroulent sous forme de *requêtes* successives.
 - Exemple : un client surfe sur un site de e-commerce, sélectionne des produits, remplit son caddy...
- D'une requête HTTP à l'autre, il faut un moyen de conserver un *état* (le caddy par ex.)
 - Autre exemple : une application bancaire. Le client effectue plusieurs opérations. On en va pas à chaque fois lui redemander son No de compte...

Stateful Session Beans

- En résumé, un Stateful Session Bean est utile pour maintenir un état pendant la durée de vie du client
 - au cours d'appels de méthodes successifs.
 - Au cours de *transactions* successives.
 - Si un appel de méthode change l'état du Bean, lors d'un autre appel de méthode l'état sera disponible.
- Conséquence : une instance de Stateful Session Bean par client.
- Avec certains containers, les Stateful Session Beans peuvent être persistants (BAS/BES...) par sérialisation.

Stateless Session Beans

- Certaines conversations peuvent se résumer à un appel de méthode, sans besoin de connaître l'état courant du Bean
 - Ex : simple traitement, simple calcul (validation de No de CB), compression...
 - Le client passe toutes les données nécessaires au traitement lors de l'appel de méthode.
- Le container est responsable de la création et de la destruction du Bean
 - Il peut le détruire *juste après un appel de méthode*, ou le garder en mémoire pendant un certain temps pour *réutilisation*.
- Une instance de Stateless Session Bean n'est pas propre à un client donné, elle peut être partagée entre chaque appel de méthode.

Pooling des Stateful Session Beans

- Le pooling des instances de Stateful Session Beans n'est pas aussi simple...
- Le client entretient une conversation avec le bean, dont l'état doit être disponible lorsque ce même client appelle une autre méthode.
- Problème si trop de clients utilisent ce type de Bean en même temps.
 - Ressources limitées (connexions, mémoire, sockets...)
 - Mauvaise scalabilité du système,
 - L'état peut occuper pas mal de mémoire...
- Problème similaire à la gestion des tâches dans un OS...

Pooling des Stateful Session Beans

- Avec un OS : on utilise le concept de *mémoire virtuelle*...
- Lorsqu'un processus ne fait plus rien (Idle), on swappe son état mémoire sur disque dur, libérant de la place.
- Lorsqu'on a de nouveau besoin de ce processus, on fait l'inverse.
- Ceci arrive souvent lorsqu'on passe d'une application à l'autre...

Pooling des Stateful Session Beans

■ Avec les Stateful Session Beans on fait pareil !

- Entre chaque appel de méthode, un client ne fait rien (Idle),
- Un utilisateur d'un site de e-commerce lit les infos sur la page www, réfléchit... de temps en temps il clique sur un bouton...
- Pendant qu'il ne fait rien, l'état du bean est swappé mais les ressources telles que les connexions BD, sockets, la mémoire intrinsèque qu'il occupe, peuvent être utilisées par un autre client.
- Etc...

Pooling des Stateful Session Beans

- Ceci a un coût : l'activation/passivation génère des E/S
- Choix du bean à swapper par LRU le plus souvent (Least Recent Used)
- Choix du bean à activer : lorsqu'on le demande (*just in time*)

En quoi consiste l'état d'un Bean Stateful?

- L'état conversationnel d'un bean suit les règles de la *sérialisation d'objets* en java.
 - En effet, la *passivation* (swap de la mémoire vers le HD) et l'activation (l'inverse) sont réalisées par sérialisation.
- Rappelez-vous que *javax.ejb.EnterpriseBean* implémente *java.io.Serializable*
- Tous les attributs du Bean *non transients* sont donc concernés.

En quoi consiste l'état d'un Bean Stateful?

- Ce sont tous les attributs de la classe + d'autres... ceux du code généré par le container.
- Peut-on effectuer des actions avant la passivation ou après l'activation (libérer des ressources, les re-ouvrir...)
- Réponse: oui!

Activation/Passivation callbacks

- Lorsqu'un bean va être mis en passivation, le container peut l'avertir (@PrePassivate)
 - Il peut libérer des ressources (connexions...)
- Idem lorsque le bean vient d'être activé (@PostActivate)

```
@Stateful  
public class MyBean {  
    @PrePassivate  
    public void passivate() {  
        <close socket connections, etc...>  
    }  
    ...  
}
```

```
@Stateful  
public class MyBean {  
    @PostActivate  
    public void activate() {  
        <open socket connections, etc...>  
    }  
    ...  
}
```

Introduction aux Entity Beans

Entity Bean, introduction

- Un Entity Bean représente

- Des objets persistants stockés dans une base de donnée,
- Des noms, des données

- Dans ce chapitre on étudiera

- Le concept de persistance,
- Ce qu'est un entity bean, du point de vue du programmeur,
- Les caractéristiques des entity beans,
- Les concepts de programmation des entity beans.

La persistance par sérialisation

- Sérialisation = sauvegarde de l'état d'un objet sous forme d'octets.
 - Rappel : l'état d'un objet peut être quelque chose de très compliqué.
 - Etat d'un objet = ses attributs, y compris les attributs hérités.
 - Si les attributs sont eux-même des instances d'une classe, il faut sauvegarder aussi les attributs de ces instances, etc...
- A partir d'un état sérialisé, on peut reconstruire l'objet
- En java, au travers de l'interface
java.io.Serializable, des méthodes de
java.io.ObjectInputStream et
java.io.ObjectOutputStream

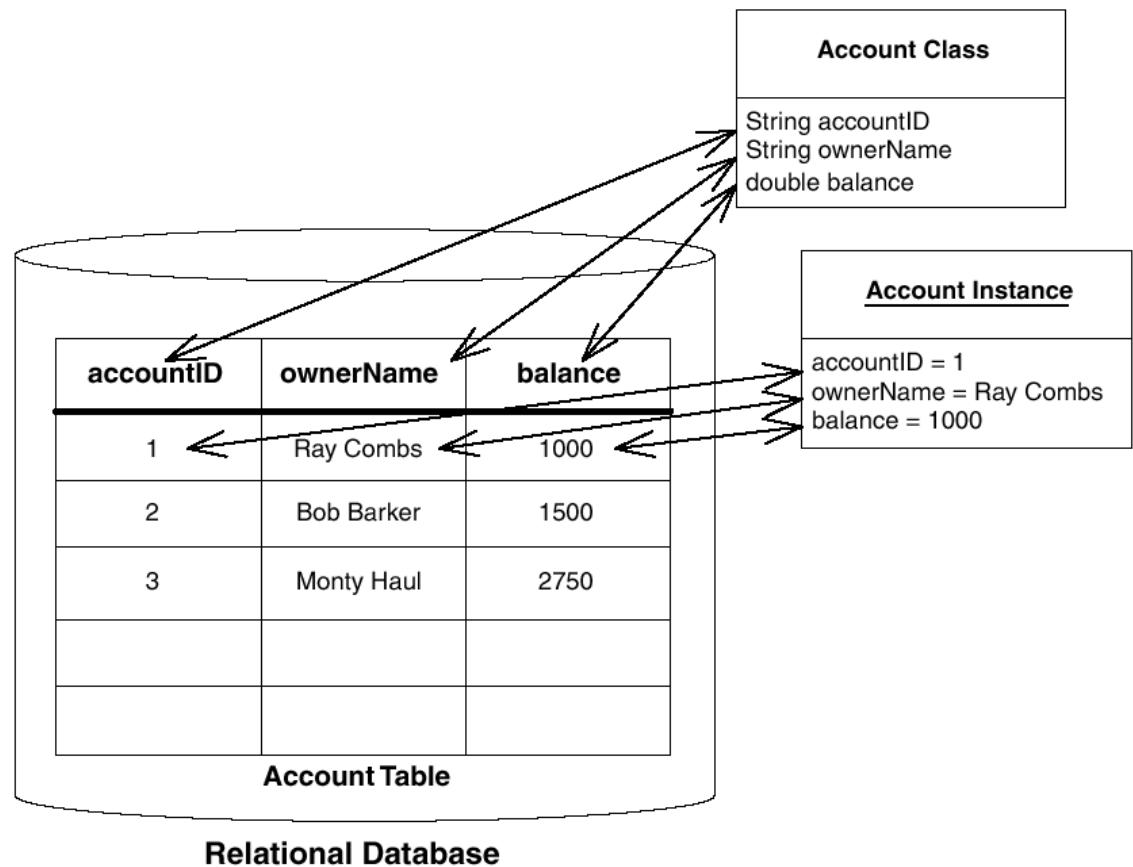
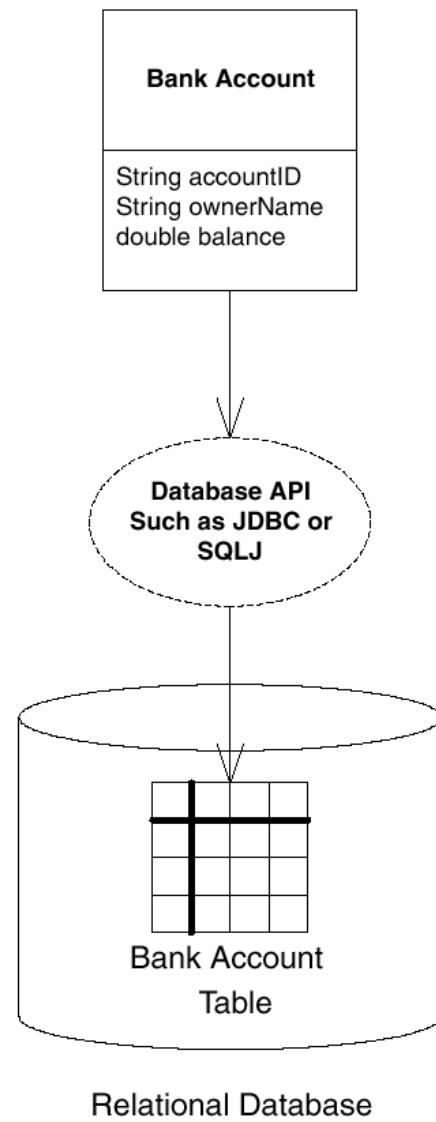
La persistance par sérialisation

- Défauts nombreux...
- Gestion des versions, maintenance...
- Pas de requêtes complexes...
 - Ex : on sérialise mille comptes bancaires. Comment retrouver ceux qui ont un solde négatif ?
- Solution : stocker les objets dans une base de donnée!

La persistance par mapping objet/BD relationelle

- On stocke l'état d'un objet dans une base de donnée.
- Ex : la classe Personne possède deux attributs **nom** et **prenom**, on associe cette classe à *une table* qui possède deux colonnes : **nom** et **prenom**.
- On décompose chaque objet en une suite de variables dont on stockera la valeur dans une ou plusieurs tables.
- Permet des requêtes complexes.

La persistance par mapping objet/BD relationnelle



La persistance par mapping objet/BD relationelle

■ Pas si simple...

- Détermination de l'état d'un objet parfois difficile, tout un art...
- Il existe des produits pour nous y aider... TopLink (WebGain), JavaBlend (Sun),
- Aujourd'hui la plupart des gens font ça à la main avec JDBC ou SQL/J.
- Mais SQL dur à tester/debugger... source de

La persistance à l'aide d'une BD Objet

- Les Base de données objet stockent *directement* des objets.
- Plus de mapping !
- Object Query Language (OQL) permet de manipuler les objets...
- Relations entre les objets évidentes (plus de join...)
- Bonnes performances mais mauvaise *scalabilité*.

Le modèle de persistence EJB 3.0

- EJB 3.0 propose un modèle standard de persistence à l'aide des Entity beans
- Les outils qui assureront la persistence (Toplink, Hibernate, etc.), intégrés au serveur d'application, devront être compatibles avec la norme EJB 3.0

Qu'est-ce qu'un Entity Bean

- Ce sont des objets qui savent se *mapper* dans une base de donnée.
- Ils utilisent un mécanisme de persistance (parmi ceux présentés)
- Ils servent à représenter sous forme d'objets des données situées dans une base de donnée
 - Le plus souvent un objet = une ou plusieurs ligne(s) dans une ou plusieurs table(s)

Qu'est-ce qu'un Entity Bean

- Exemples
 - Compte bancaire (No, solde),
 - Employé, service, entreprises, livre, produit,
 - Cours, élève, examen, note,
- Mais au fait, pourquoi nous embêter à passer par des objets ?
 - Plus facile à manipuler par programme,
 - Vue plus compacte, on regroupe les données dans un objet.
 - On peut associer des méthodes simples pour manipuler ces données...
 - On va gagner la couche middleware !

Exemple avec un compte bancaire

- On lit les informations d'un compte bancaire en mémoire, dans une instance d'un entity bean,
- On manipule ces données, on les modifie en changeant les valeurs des attributs d'instance,
- Les données seront mises à jour dans la base de données automatiquement !
- Instance d'un entity bean = *une vue* en mémoire des données physiques

Fichiers composant un entity bean

- Idem session beans mais quelques particularités cependant:
 - Interface locale uniquement (performances),
 - La classe du bean se mappe dans une base de données.
 - C'est une classe java « normale » (POJO) avec des attributs, des accesseurs, des modifieurs, etc.
 - On utilisera les méta-données ou « attributs de code » pour indiquer le mapping, la clé primaire, etc.
 - Clé primaire = un objet sérializable, unique pour chaque instance. C'est la clé primaire au sens SQL.
 - On manipulera les données de la BD à l'aide des EntityBeans + à l'aide d'un PERSISTENT MANAGER.
 - Le PM s'occupera de tous les accès disque, du cache, etc.
 - Lui seul contrôle quand et comment on va accéder à la BD, c'est lui qui génère le SQL, etc.

Un exemple d'entity bean

```
package examples.entity.intro;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.Id;

/**
 * This demo entity represents a Bank Account.
 * <p>
 * The entity is not a remote object and can only be accessed locally by
 * clients. However, it is made serializable so that instances can be
 * passed by value to remote clients for local inspection.
 * <p>
 * Access to persistent state is by direct field access.
 */

@Entity
public class Account implements Serializable {

    // The account number is the primary key
    @Id
    public int accountNumber;
    public String ownerName;
    public int balance;

    /**
     * Entities must have a public no-arg constructor
     */
    public Account() {
        // our own simple primary key generation
        accountNumber = (int) System.nanoTime();
    }

    /**
     * Deposit a given amount
     * @param amount
     */
    public void deposit(int amount) {
        balance += amount;
    }

    /**
     * Withdraw a given amount, or 0 if it is larger than the balance
     */
```

```
     * @param amount
     * @return The amount that was withdrawn
     */
    public int withdraw(int amount) {
        if (amount > balance) {
            return 0;
        } else {
            balance -= amount;
            return amount;
        }
    }
}
```

- La classe = POJO,
- Sérifiable,
- Un attribut = la clé primaire
- C'est tout !

Client de l'entity bean précédent : un session bean (servant de façade)

```
package examples.entity.intro;

import java.util.List;
import javax.ejb.Stateless;
import javax.ejb.Remote;
import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;
import javax.persistence.Query;

/**
 * Stateless session bean facade for account entities,
 * remotely accessible
 */
@Stateless
```

- Ce session bean est stateless,
- Utilise un EntityManager,
 - Sert à envoyer des requêtes EJB QL,
 - Méthode persist(entity) pour créer une nouvelle entrée (insert)
 - Le reste passe par des appels de méthodes classiques de l'entity bean.

```
@Remote(Bank.class)
public class BankBean implements Bank {

    /** the entity manager object, injected by the container */
    @PersistenceContext
    private EntityManager manager;

    public List<Account> listAccounts() {
        Query query = manager.createQuery("SELECT a FROM Account a");
        return query.getResultList();
    }

    public Account openAccount(String主人名) {
        Account account = new Account();
        account.主人名 = 主人名;
        manager.persist(account);
        return account;
    }

    public int getBalance(int 账户号) {
        Account account = manager.find(Account.class, 账户号);
        return account.balance;
    }

    public void deposit(int 账户号, int amount) {
        Account account = manager.find(Account.class, 账户号);
        account.deposit(amount);
    }

    public int withdraw(int 账户号, int amount) {
        Account account = manager.find(Account.class, 账户号);
        return account.withdraw(amount);
    }

    public void close(int 账户号) {
        Account account = manager.find(Account.class, 账户号);
        manager.remove(account);
    }
}
```

Quand se font les E/S ? Les transactions ?

- Par défaut, une transaction est créée dès qu'on entre dans un session bean, et se termine soit en cas d'exception (rollback), soit en cas de succès (sortie du session bean, commit),
- Les E/S s'effectuent par défaut en début et fin de transaction.
- On doit cependant faire un find chaque fois que l'on cherche le solde d'un compte...

Autre version : on garde dans le session bean la « mémoire » de l'entity bean

```
package examples.entity.intro;

import javax.ejb.Remote;
import javax.ejb.Stateful;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.PersistenceContextType;

/**
 * Stateful session bean facade for account entities, remotely
 * accessible
 */

@Stateful
@Remote(AccountInterface.class)
public class AccountBean implements AccountInterface {

    /** The entity manager, injected by the container */
    @PersistenceContext(type=PersistenceContextType.EXTENDED,
        unitName="intro")
    private EntityManager manager;

    private Account account = null;

    public void open(int accountNumber) {
```

- Le session bean est stateful,
- Il garde la référence de l'entity bean,
- On a du « étendre » la portée du Persistence Manager

Suite de l'exemple

```
account = manager.find(Account.class, accountNumber);
if (account == null) {
    account = new Account();
    account.ownerName = "anonymous";
    account.accountNumber = accountNumber;
    manager.persist(account);
}

public int getBalance() {
    if(account==null)
        throw new IllegalStateException();
    return account.balance;
}

public void deposit(int amount) {
    if(account==null)
        throw new IllegalStateException();
    account.deposit(amount);
}

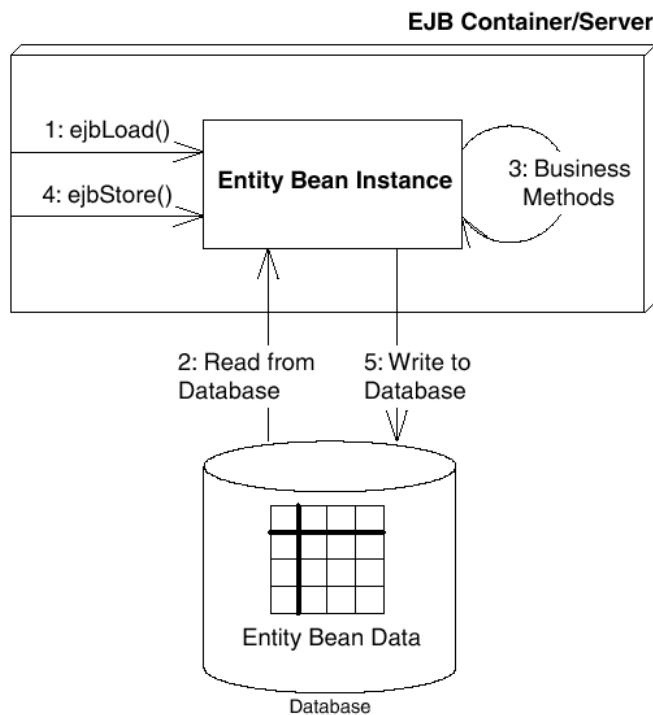
public int withdraw(int amount) {
    if(account==null)
        throw new IllegalStateException();
    return account.withdraw(amount);
}
```

- Dans getBalance() on utilise plus de find,
- On utilise les Exceptions

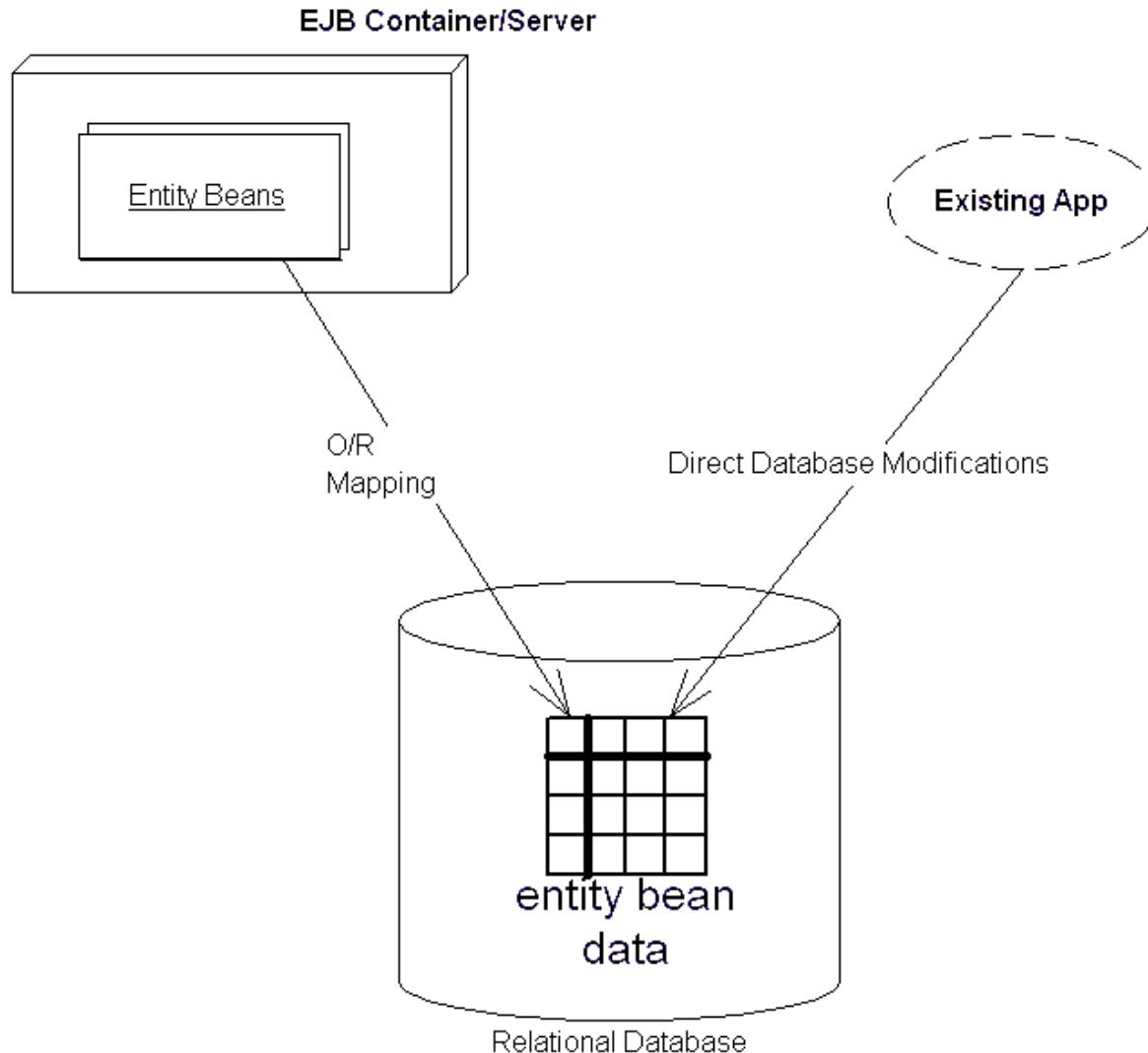
Caractéristiques des entity beans

- Survivent aux crashes du serveur, du SGBD
- Ce sont des vues sur des données dans un SGBD

This ejbLoad()-business method-ejbStore() cycle may be repeated many times.



Modifier les données sans passer par le bean



Packager et déployer un Entity Bean

- Les EB sont déployés dans des « persistence Units »,
 - Spécifié dans le fichier « persistence.xml » qui est dans le jar contenant les EJBs.
 - Exemple le plus simple :

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence">
    <persistence-unit name="intro"/>
</persistence>
```

- Mais on peut ajouter de nombreux paramètres :
 - <description>, <provider>, <transaction type>, <mapping file> etc.

Que faire avec un entity manager ?

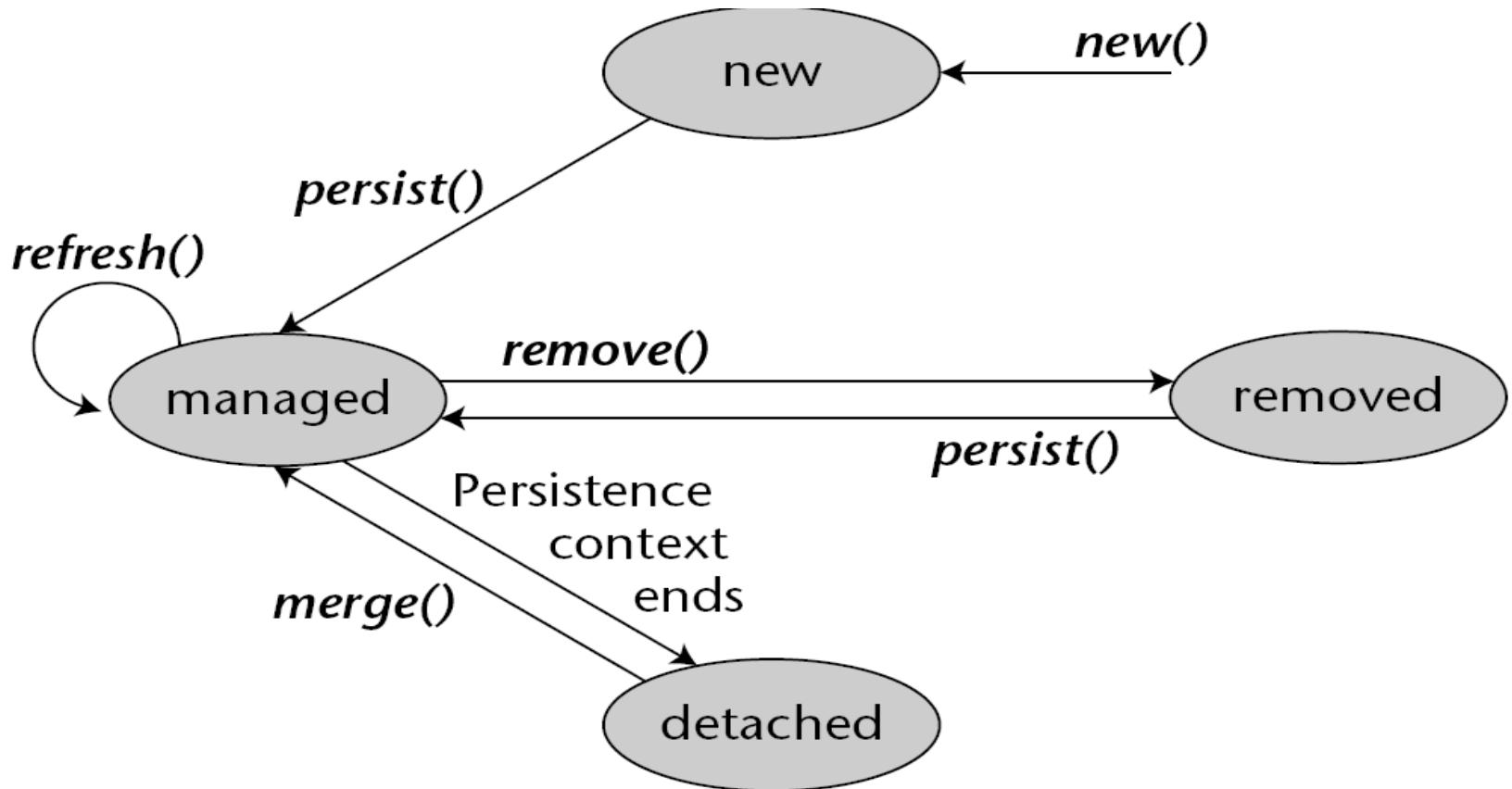
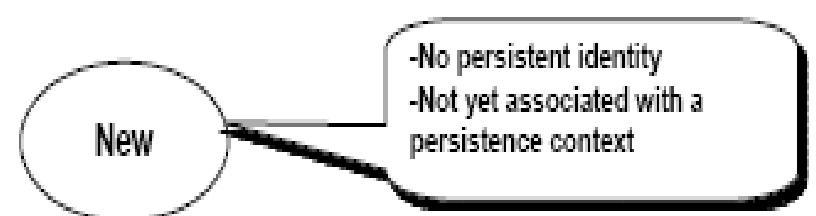


Figure 6.3 Entity life cycle.

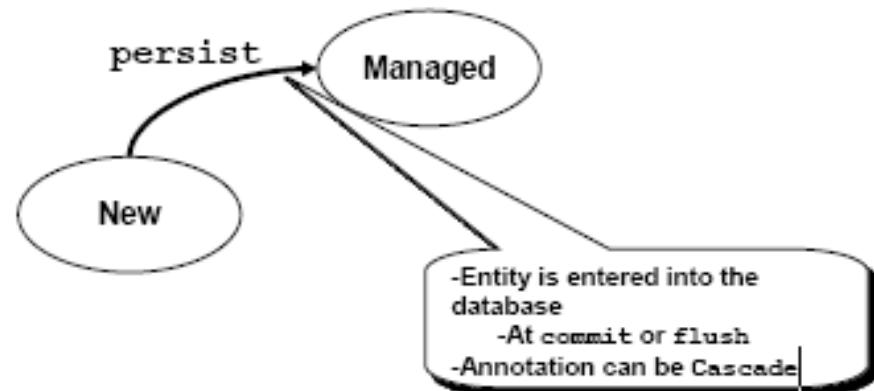
POJO: cycle de vie

- Pas d 'identité de persistance
- Pas associé au conteneur



- Identité de persistance
- Géré par le conteneur
- Synchronisation SGBDR
- Gestion des relations

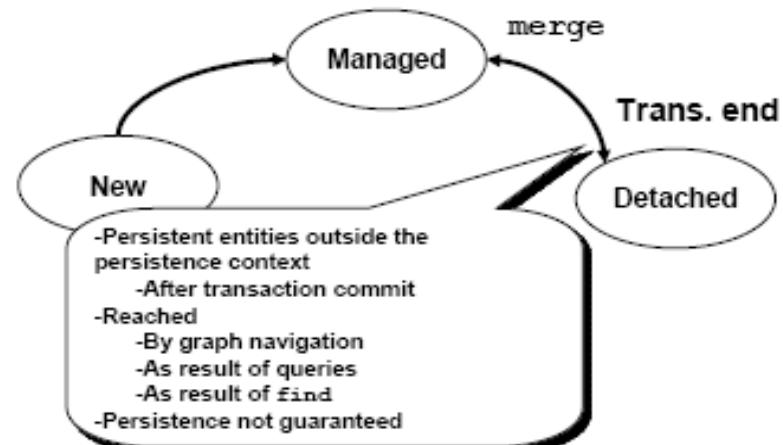
Réalisé par: MEDINI



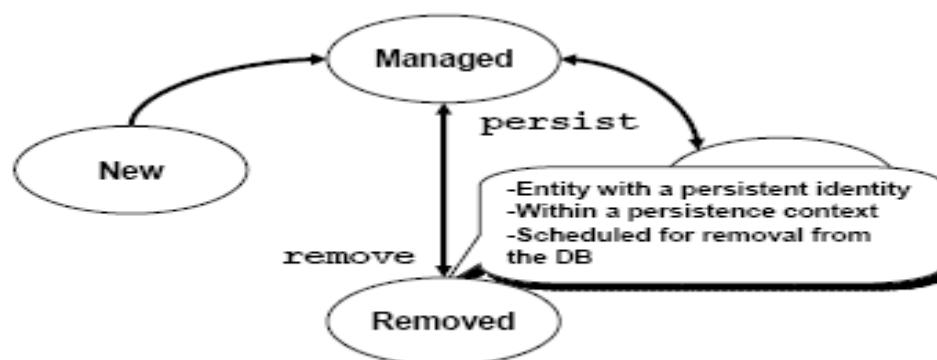
POJO: cycle de vie



- Identité de persistance
- Non géré par le conteneur
- Nécessité de la synchronisation



- Identity de persistance
- Suppression du bean



Etats d'un Entity Bean

■ Un EB peut avoir 4 états

1. **New**: le bean existe en mémoire mais n'est pas encore associé à une BD, il n'est pas encore associé à un contexte de persistence (via l'entity manager)
2. **Managed** : après le persist() par exemple. Le bean est associé avec les données dans la BD. Les changements seront répercutés (transaction terminées ou appel a flush())
3. **Detached** : le bean est n'est plus associé au contexte de persistence
4. **Removed** : le bean est associé à la BD, au contexte, et est programmé pour être supprimé (les données seront supprimées aussi).

Utilisation du persistent manager

- Remove() pour supprimer des données,
- Set(), Get(), appel de méthodes de l'entity bean pour modifier les données, mais le bean doit être dans un état « managed »,
- Persist() pour créer des données, le bean devient managé,
- Merge pour faire passer un bean « detached » dans l'état « managed ».

Exemple de merge() avec le bean stateless

- Note : avec le PM en mode « étendu » ceci aurait été inutile

```
public Account openAccount(String ownerName) {  
    Account account = new Account();  
    account.ownerName = ownerName;  
    manager.persist(account);  
    return account;  
}
```

```
public void update(Account detachedAccount) {  
    Account managedAccount = manager.merge(detachedAccount);  
}
```

Callbacks

- Comme pour les session beans, on peut définir des méthodes de l'entity bean qui seront appelées à divers moments de son cycle de vie.
- Pour cela on utilise les attributs de code:
 - PrePersist
 - PostPersist
 - PreRemove
 - PostRemove
 - PreUpdate
 - PostUpdate
 - PostLoad

Recherche d'entity beans

- Les entity beans correspondant à des lignes dans une BD, on peut avoir besoin de faire des recherches.
- Similaire à un SELECT
- Plusieurs fonctions sont proposées par l'entity manager

Recherche d'entity beans

■ Recherche par clé primaire :

```
/** Find by primary key. */
public <T> T find(Class<T> entityClass, Object primaryKey);
```

■ Exécution de requêtes EJB-QL

```
public List<Account> listAccounts() {
    Query query = manager.createQuery("SELECT a FROM Account a");
    return query.getResultList();
}
```

Recherche d'entity beans

■ Requêtes SQL:

```
public Query createNativeQuery(String sqlString, Class resultClass);  
  
public Query createNativeQuery(String sqlString,  
    String resultSetMapping);
```

■ Requêtes nommées:

```
public List<Account> listAccounts() {  
    Query query = manager.createNamedQuery("findThem");  
    return query.getResultList();  
}
```

```
@Entity  
@NamedQuery(name="findThem", queryString="SELECT a FROM Account a")  
public class Account implements Serializable {...}
```

Message-Driven Beans

Message-Driven Beans

- Nouveauté apparue avec EJB 2.0,
- *Messaging* = moyen de communication léger, comparé à RMI-IIOP,
- Pratique dans de nombreux cas,
- Message-Driven beans = beans accessibles par *messaging* asynchrone.

Message-Driven Beans : motivation

■ Performance

- Un client RMI-IIOP *attend* pendant que le serveur effectue le traitement d'une requête,

■ Fiabilité

- Lorsqu'un client RMI-IIOP parle avec un serveur, ce dernier doit être en train de fonctionner. S'il crashe, ou si le réseau crashe, le client est coincé.

■ Pas de broadcasting !

- RMI-IIOP limite les liaisons 1 client vers 1 serveur

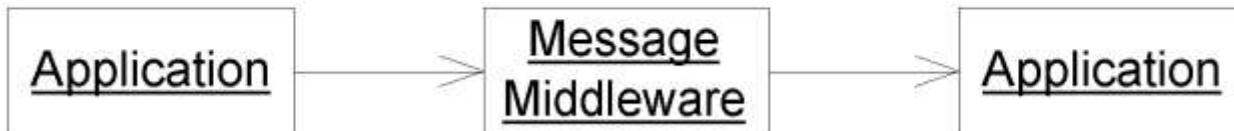
Messaging

- C'est comme le mail ! Ou comme si on avait une troisième personne entre le client et le serveur !

Remote method invocations:



Messaging:



Messaging

- A cause de ce "troisième homme" les performances ne sont pas toujours au rendez-vous !
- Message Oriented Middleware (MOM) est le nom donné aux middlewares qui supportent le *messaging*.
 - Tibco Rendezvous, IBM MQSeries, BEA Tuxedo/Q, Microsoft MSMQ, Talarian SmartSockets, Progress SonicMQ, Fiorano FioranoMQ, ...
 - Ces produits fournissent : messages avec garantie de livraison, tolérance aux fautes, load-balancing des destinations, etc...

The Java Message Service (JMS)

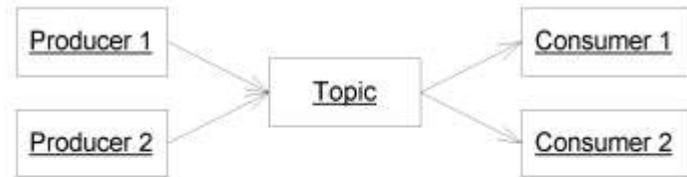
- Les serveurs MOM sont pour la plupart propriétaires : pas de portabilité des applications !
- JMS = un standard pour normaliser les échanges entre composant et serveur MOM,
 - Une API pour le développeur,
 - Un Service Provider Interface (SPI), pour connecter l'API et les serveurs MOM, via les drivers JMS

JMS : Messaging Domains

- Avant de faire du messaging, il faut choisir un *domaine*
 - Domaine = type de messaging

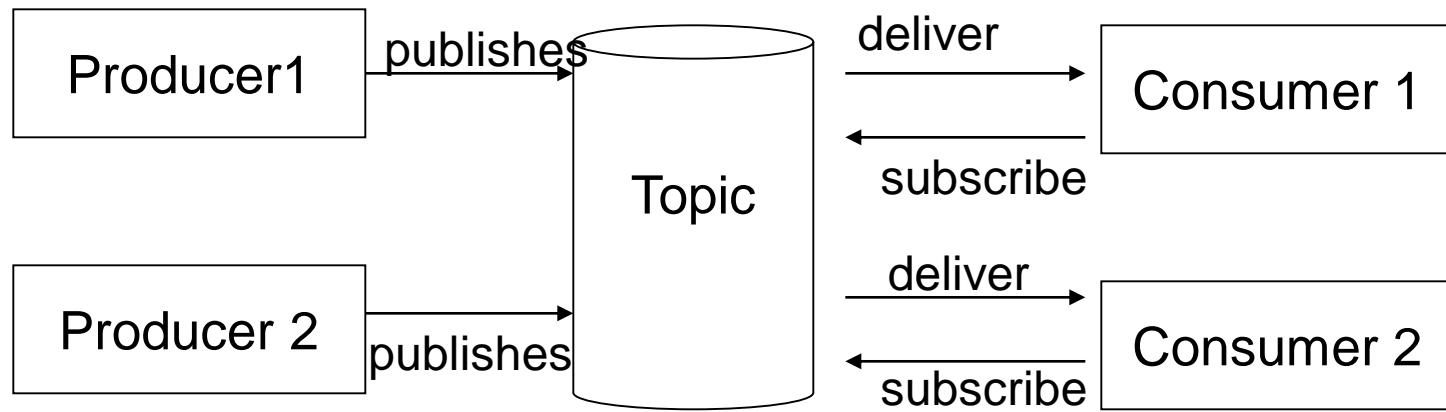
- Domaines possibles
 - Publish/Subscribe (pub/sub) : n producteurs, n consommateurs (tv)
 - Point To Point (PTP) : n producteurs, 1 consommateur

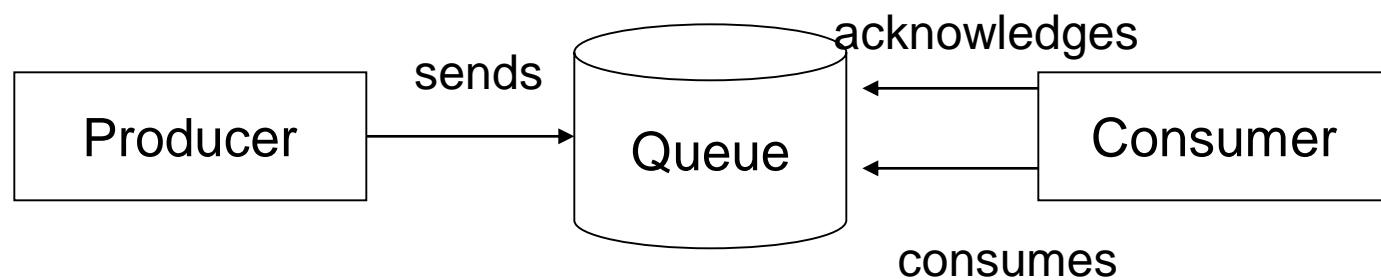
Publish/subscribe:



Point-to-point:







■ Point-to-point model

- In the **point-to-point model**, a *sender* posts messages to a particular queue and a *receiver* reads messages from the queue. Here, the sender knows the destination of the message and posts the message directly to the receiver's queue.

- This model is characterized by the following:
 - Only one consumer gets the message.
 - The producer does not have to be running at the time the consumer consumes the message, nor does the consumer need to be running at the time the message is sent.
 - Every message successfully processed is acknowledged by the consumer.

- **Publish/subscribe model**
- The **publish/subscribe model** supports publishing messages to a particular message topic. *Subscribers* may register interest in receiving messages on a particular message topic. In this model, neither the *publisher* nor the subscriber knows about each other. A good analogy for this is an anonymous bulletin board.

Characteristics of this model:

- Multiple consumers (or none) will receive the message.
- There is a timing dependency between publishers and subscribers. The publisher has to create a message topic for clients to subscribe. The subscriber has to remain continuously active to receive messages, unless it has established a durable subscription. In that case, messages published while the subscriber is not connected will be redistributed whenever it reconnects.

JMS : les étapes

1. Localiser le driver JMS

- lookup JNDI. Le driver est une *connection factory*

2. Créer une connection JMS

- obtenir une *connection* à partir de la *connection factory*

3. Créer une session JMS

- Il s'agit d'un objet qui va servir à recevoir et envoyer des messages.
On l'obtient à partir de la *connection*.

4. Localiser la destination JMS

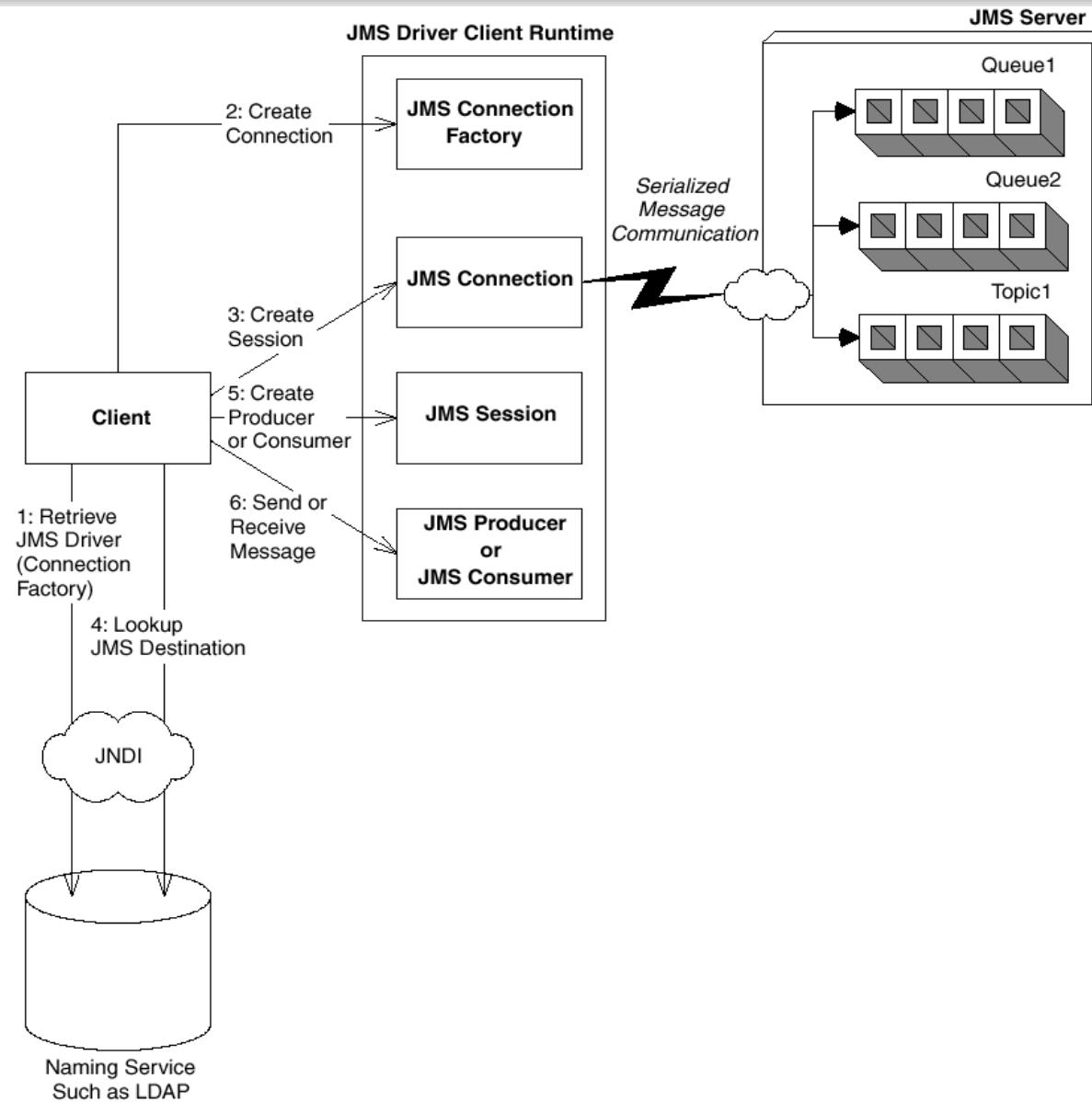
- Il s'agit du canal, de la chaîne télé ! Normalement, c'est réglé par le déployeur. On obtient la *destination* via JNDI.

5. Créer un producteur ou un consommateur JMS

- Utilisés pour écrire ou lire un message. On les obtient à partir de la *destination* ou de la *session*.

6. Envoyer ou recevoir un message

JMS : les étapes



JMS : les interfaces

PARENT INTERFACE	POINT-TO-POINT	PUB/SUB
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver, QueueBrowser	TopicSubscriber

JMS : exemple de code (1)

```
package examples.messaging;

import javax.jms.*;
import javax.naming.InitialContext;

public class LogClient {

    public static void main(String[] args) throws Exception {
        // Initialize JNDI
        InitialContext ctx = new InitialContext(System.getProperties());

        // 1: Lookup connection factory
        TopicConnectionFactory factory =
            (TopicConnectionFactory) ctx.lookup
                ("jms/TopicConnectionFactory");

        // 2: Use connection factory to create JMS connection
        TopicConnection connection = factory.createTopicConnection();

        // 3: Use connection to create a session
        TopicSession session =
            connection.createTopicSession
                (false,Session.AUTO_ACKNOWLEDGE);

        // 4: Lookup destination
        Topic topic = (Topic)ctx.lookup("jms/Topic");

        // 5: Create a message publisher
```

JMS : exemple de code (2)

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

    // 6: Create and publish a message
    TextMessage msg = session.createTextMessage();
    msg.setText("This is a test message.");
    publisher.send(msg);

    // finish
    publisher.close();
    System.out.println("Message published. Please check application
server's console to see the response from MDB.");
}

}
```

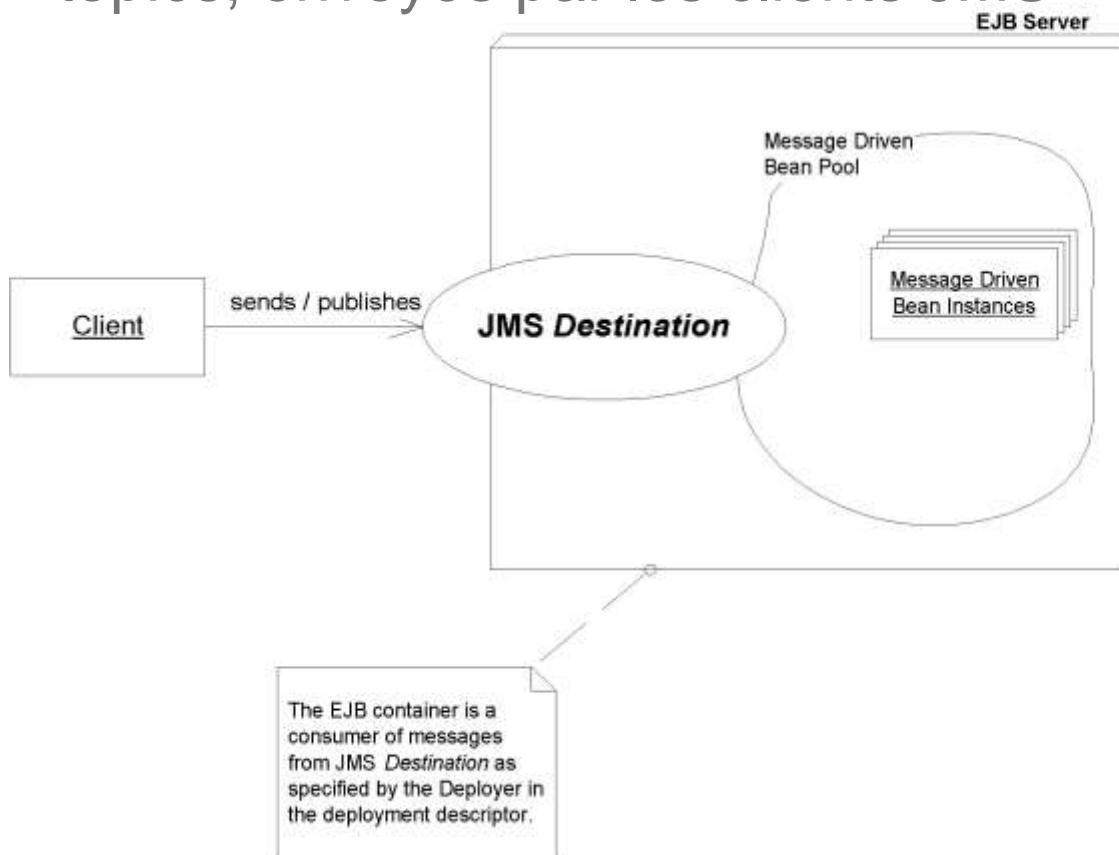
Note : Dans 3) false = pas de transactions, AUTO_ACKNOWLEDGE = inutile ici puisqu'on envoie des messages.

Intégrer JMS et les EJB

- Pourquoi créer un nouveau type d'EJB ?
- Pourquoi ne pas avoir délégué le travail à un objet spécialisé ?
- Pourquoi ne pas avoir augmenté les caractéristiques des session beans ?
- Parce que ainsi on peut bénéficier de tous les avantages déjà rencontrés : cycle de vie, *pooling*, descripteurs spécialisés, code simple...

Qu'est-ce qu'un Message-Driven Bean ?

- Un EJB qui peut recevoir des messages
 - Il consomme des messages depuis les queues ou topics, envoyés par les clients JMS



Qu'est-ce qu'un Message-Driven Bean ?

- Un client n'accède pas à un MDB via une interface, il utilise l'API JMS,
- Un MDB n'a pas d'interface Home, Local Home, Remote ou Local,
- Les MDB possèdent une seule méthode, faiblement typée : **onMessage ()**
 - Elle accepte un message JMS (**BytesMessage**, **ObjectMessage**, **TextMessage**, **StreamMessage** ou **MapMessage**)
 - Pas de vérification de types à la compilation.
 - Utiliser `instanceof` au run-time pour connaître le type du message.
- **Les MDB n'ont pas de valeur de retour**
 - Ils sont déconnectés des *producteurs* de messages.¹⁵¹

Qu'est-ce qu'un Message-Driven Bean ?

- Pour envoyer une réponse à l'expéditeur : plusieurs *design patterns*...
- Les MDB ne renvoient pas d'exceptions au client (mais au container),
- Les MDB sont stateless...
- Les MDB peuvent être des abonnés durables ou non-durables (*durable or nondurable subscribers*) à un *topic*
 - *Durable* = *reçoit tous les messages, même si l'abonné est inactif,*
 - *Dans ce cas, le message est rendu persistant et sera délivré lorsque l'abonné sera de nouveau actif.*
 - *Nondurable* = *messages perdus lorsque abonné inactif.*

Qu'est-ce qu'un Message-Driven Bean ?

- Le consommateur (celui qui peut les détruire) des messages est en général le Container
 - C'est lui qui choisit d'être durable ou non-durable,
 - S'il est durable, les messages résistent au crash du serveur d'application.

Développer un Message-Driven Bean

- Les MDBs doivent implémenter

```
public interface javax.jms.MessageListener {  
    public void onMessage(Message message);  
}  
  
public interface javax.ejb.MessageDrivenBean  
extends EnterpriseBean {  
  
    public void ejbRemove()  
        throws EJBException;  
  
    public void setMessageDrivenContext(MessageDrivenContext ctx)  
        throws EJBException;  
}
```

- La classe d'implémentation doit fournir une méthode **ejbCreate()** qui renvoie **void** et qui n'a pas d'arguments.

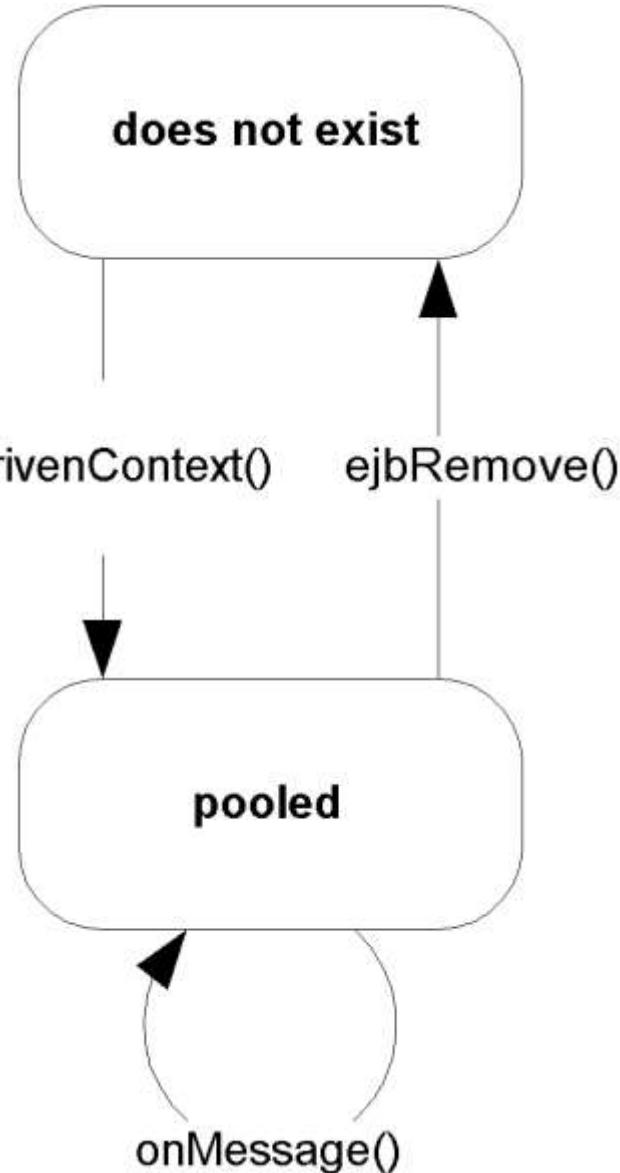
Développer un Message-Driven Bean

- Méthodes qui doivent être implémentées
 - **onMessage (Message)**
 - Invoquée à chaque consommation de message
 - Un message par instance de MBD, pooling assuré par le container
 - **setMessageDrivenContext (MessageDrivenContext)**
 - Appelée avant ejbCreate, sert à récupérer le contexte.
 - Ne contient que des méthodes liées aux transactions...

Développer un Message-Driven Bean

The lifecycle of an message driven bean.
Each method call shown is an invocation from the container to the bean instance.

1: newInstance()
2: setMessageDrivenContext()
3: ejbCreate()



Un exemple simple

- Un MDB qui fait du logging, c'est à dire affiche des messages de textes à l'écran chaque fois qu'il consomme un message.
 - Utile pour débugger....
- Rappel : pas d'interfaces !

La classe du bean

```
package examples.messaging;

import javax.jms.*;
import javax.ejb.*;
import javax.annotation.*;

@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationType",
        propertyValue = "javax.jms.Topic")
})
public class LogBean implements MessageListener {

    public LogBean() {
        System.out.println("LogBean created");
    }

    public void onMessage(Message msg) {
        if (msg instanceof TextMessage) {
            TextMessage tm = (TextMessage) msg;
            try {
                String text = tm.getText();
                System.out.println("Received new message : " + text);
            } catch (JMSEException e) {
                e.printStackTrace();
            }
        }
    }
}
```

La classe du bean (suite)

```
}

@PreDestroy
public void remove() {
    System.out.println("LogBean destroyed.");
}
}
```

Question ?

- Comment sait-on quelle queue ou quel topic de messages le bean consomme ?
 - Cela n'apparaît pas dans le descripteur !
- C'est fait exprès pour rendre les MDB portables et réutilisables.
- L'information se trouve dans l'`@ActivationConfigProperty` au début du code

Exemple de descripteur spécifique, tiré d'un autre exemple (Borland)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Borland Software Corporation//DTD Enterprise JavaBeans 2.0//EN"
 "http://www.borland.com/devsupport/appserver/dtds/ejb-jar_2_0-borland.dtd">
<ejb-jar>
  <enterprise-beans>
    <message-driven>
      <ejb-name>HelloEJBQueue</ejb-name>
      <message-driven-destination-name>serial://jms/q</message-driven-destination-name>
      <connection-factory-name>serial://jms/xaqcf</connection-factory-name>
      <pool>
        <max-size>20</max-size>
        <init-size>2</init-size>
      </pool>
    </message-driven>
    <message-driven>
      <ejb-name>HelloEJBTTopic</ejb-name>
      <message-driven-destination-name>serial://jms/t</message-driven-destination-name>
      <connection-factory-name>serial://jms/tcf</connection-factory-name>
      <pool>
        <max-size>20</max-size>
        <init-size>2</init-size>
      </pool>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

Le client (1)

```
import javax.naming.*;  
  
import javax.jms.*;  
import java.util.*;  
  
public class Client {  
    public static void main (String[] args) throws Exception {  
        // Initialize JNDI  
        Context ctx = new InitialContext(System.getProperties());  
  
        // 1: Lookup ConnectionFactory via JNDI  
        TopicConnectionFactory factory =  
            (TopicConnectionFactory)  
            ctx.lookup("javax.jms.TopicConnectionFactory");  
  
        // 2: Use ConnectionFactory to create JMS connection  
        TopicConnection connection =  
            factory.createTopicConnection();
```

Le client (2)

```
// 3: Use Connection to create session
TopicSession session = connection.createTopicSession(
    false, Session.AUTO_ACKNOWLEDGE);

// 4: Lookup Destination (topic) via JNDI
Topic topic = (Topic) ctx.lookup("testtopic");

// 5: Create a Message Producer
TopicPublisher publisher = session.createPublisher(topic);

// 6: Create a text message, and publish it
TextMessage msg = session.createTextMessage();
msg.setText("This is a test message.");
publisher.publish(msg);

}
```

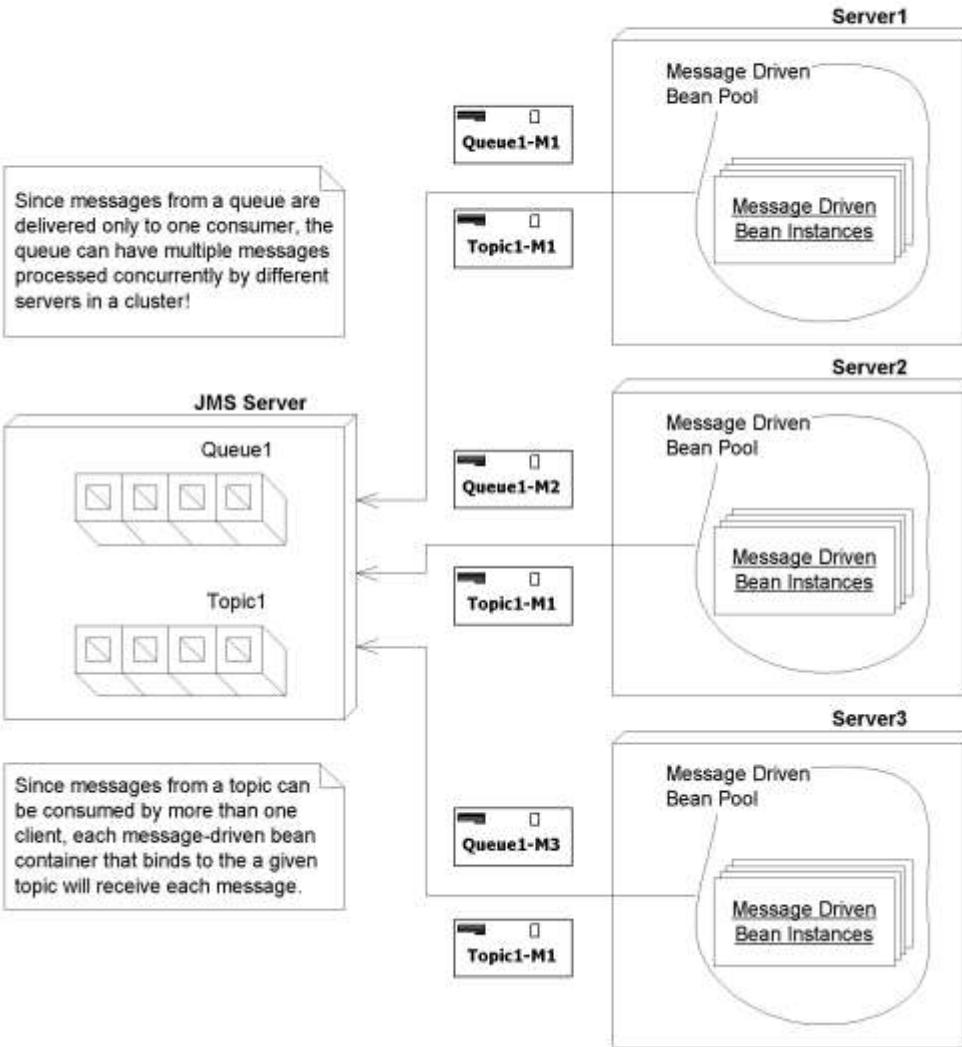
Concepts avancés

- Transactions et MBD,
 - La production et la consommation du message sont dans deux transactions séparées...
- Sécurité,
 - Les MDB ne reçoivent pas les informations de sécurité du producteur avec le message. On ne peut pas effectuer les opérations classiques de sécurité sur les EJB.
- Load-Balancing,
 - Modèle idéal : les messages sont dans une queue et ce sont les MDB qui consomment, d'où qu'ils proviennent.
 - Comparer avec les appels RMI-IIOP pour les session et entity beans, où on ne peut que faire des statistiques...

Concepts avancés

- Consommation dupliquée dans les architectures en clusters : utiliser une queue au lieu d'un topic si on veut que le message ne soit consommé qu'une fois !
- Chaque container est un consommateur !

Concepts avancés



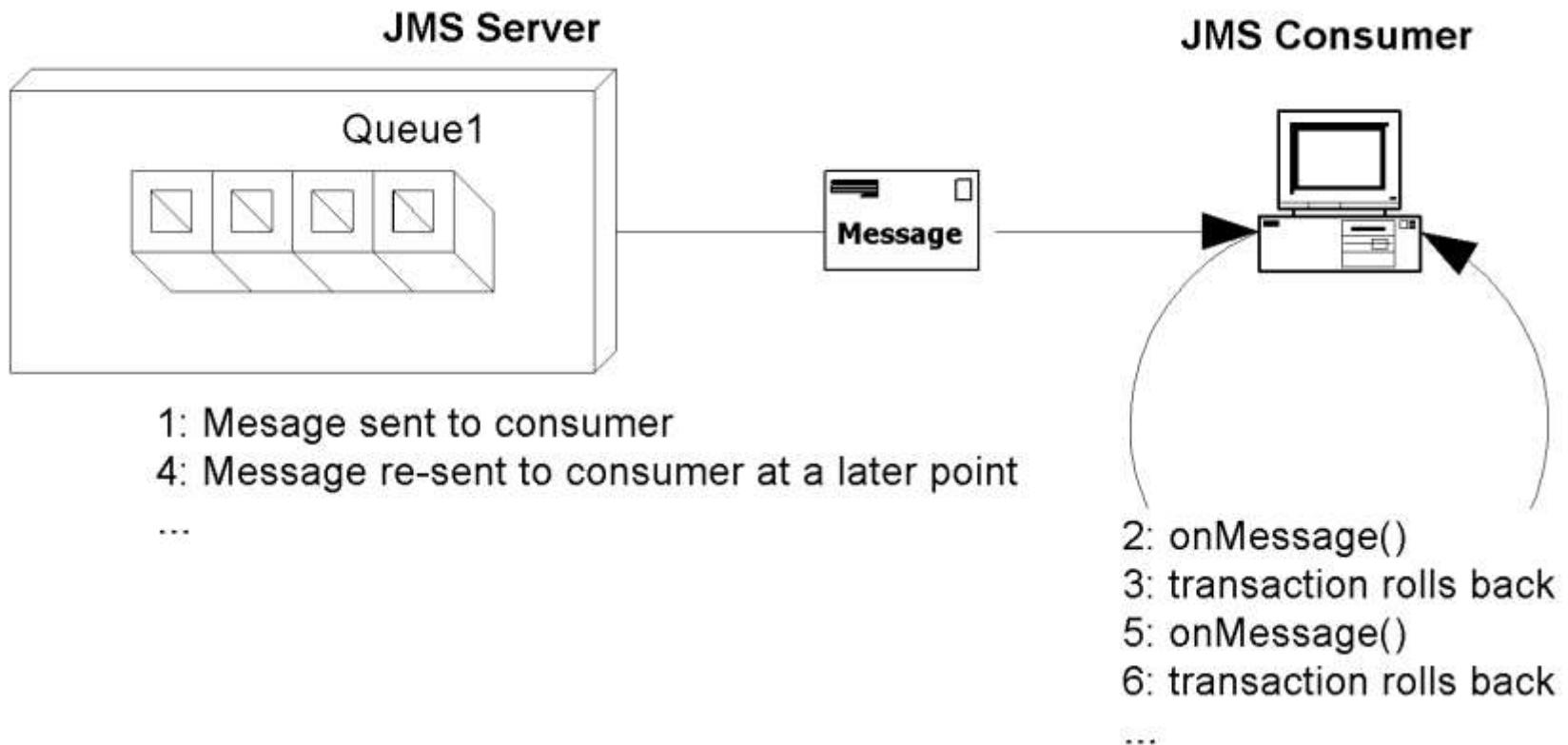
Pièges !

- Ordre des messages
 - Le serveur JMS ne garantit pas l'ordre de livraison des messages.
- L'appel à ejbRemove() n'est pas garanti, comme pour les session beans stateless...
 - A cause du pooling,
 - En cas de crash.
- Messages empoisonnés (*poison messages*)
 - A cause des transactions un message peut ne jamais être consommé

Pièges !

■ Messages empoisonnés (*poison messages*)

- A cause des transactions un message peut ne jamais être consommé



MDB empoisonné !

```
package examples;

import javax.ejb.*;
import javax.jms.*;

public class PoisonBean
    implements MessageDrivenBean, MessageListener {

    private MessageDrivenContext ctx;

    public void setMessageDrivenContext(MessageDrivenContext ctx) {
        this.ctx = ctx;
    }

    public void ejbCreate() {}

    public void ejbRemove() {}

    ...
}
```

MDB empoisonné !

```
...  
  
public void onMessage(Message msg)  {  
    try {  
  
        System.out.println("Received msg " + msg.getJMSMessageID());  
  
        // Let's sleep a little bit so that we don't see rapid fire re-sends of the message.  
        Thread.sleep(3000);  
  
        // We could either throw a system exception here or  
        // manually force a rollback of the transaction.  
        ctx.setRollbackOnly();  
    }  
  
    catch (Exception e) {  
        e.printStackTrace();  
    }  
}  
}
```

MDB empoisonné !

■ Solutions

- Ne pas lever d'exception,
- Utiliser des transactions gérées par le bean, non par le container,
- Certains serveurs peuvent configurer une "poison message queue" ou posséder un paramètre "nb max retries"
- ...

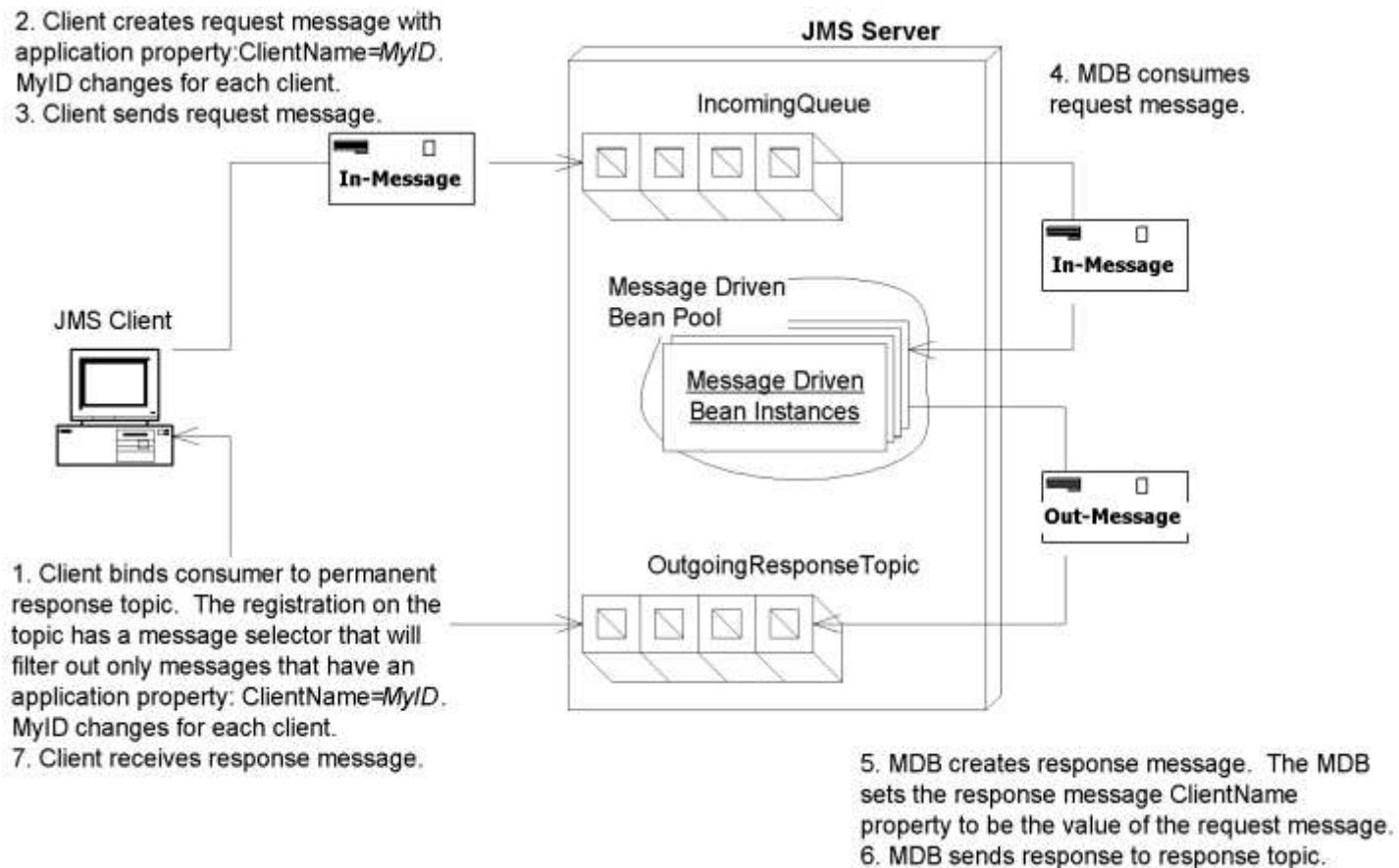
Comment renvoyer des résultats à l'expéditeur du message ?

- A faire à la main ! Rien n'est prévu !

Comment renvoyer des résultats à l'expéditeur du message ?

- Néanmoins, des problèmes se posent si le client est lui-même un EJB de type stateful session bean
 - Que se passe-t-il en cas de *passivation* ?
 - Perte de la connexion à la destination temporaire!
- Solution : ne pas utiliser d'EJB SSB comme client! Utiliser une Servlet ou un JSP
- Autre solution : configurer un topic permanent pour les réponses, au niveau du serveur JMS.

Comment renvoyer des résultats à l'expéditeur du message ?



Comment renvoyer des résultats à l'expéditeur du message ?

- D'autres solutions existent...
- JMS propose deux classes `javax.jms.QueueRequestor` et `javax.jms.TopicRequestor` qui implémentent une pattern simple question/réponse
- Solution bloquante, pas de gestion de transactions...
- Le futur : invocation de méthode asynchrone

Ajouter de la fonctionnalité aux beans

Dans ce chapitre, nous allons voir

- Comment appeler un bean depuis un autre bean,
- Utiliser les propriétés d'environnement du bean au runtime,
- Comment utiliser les *resources factories* (JDBC, JMS),
- Utiliser le modèle de sécurité EJB,
- Comment utiliser les *EJB object handles* et *EJB home handles*.

Appeler un bean depuis un autre bean

- La conception de la moindre application implique des beans qui sont clients d'autres beans...
 - Un bean gestionnaire de compte (session) utilise des beans compte bancaires (entity), etc...
- On utilise JNDI comme pour n'importe quel client
 1. *Lookup* du *home object* de l'EJB cible via JNDI,
 2. Appel de **create ()** sur le *home object*,
 3. Appel de méthodes business,
 4. Appel de **remove** sur le *ejb object*.

Appeler un bean depuis un autre bean

- Comment indiquer les paramètres d'initialisation JNDI (driver, etc...) ?
- Pas besoin, le container les a déjà initialisés
 - On les récupères via un InitialContext

```
//Obtain the DEFAULT JNDI initial context by calling the  
//no-argument constructor  
Context ctx =new InitialContext();  
//Look up the home interface  
Object result =ctx.lookup("java:comp/env/ejb/CatalogHome");  
//Convert the result to the proper type,RMI-IIOP style  
CatalogHome home =(CatalogHome)  
javax.rmi.PortableRemoteObject.narrow(  
result,CatalogHome.class);  
//Create a bean  
Catalog c =home.create(...);
```

Comprendre les références EJB

- Dans le code précédent, la localisation JNDI du bean recherché est `java:comp/env/ejb`.
 - Recommandé par la spécification EJB.
 - Peut néanmoins changer si le serveur JNDI a une configuration bizarre.
 - Le déployeur ne peut pourtant pas changer votre code !
- EJB références = dans le descripteur
 - Permet au déployeur de vérifier que tous les beans référencés par un autre bean sont bien déployés et enregistrés avec les noms JNDI corrects.

Comprendre les références EJB

```
...
<enterprise-beans>
<!--
Here, we define our Catalog bean. We use the "Catalog"ejb-name.We will use
this below.
-->
<session>
<ejb-name>Catalog</ejb-name>
<home>examples.CatalogHome</home>
...
</session>

<session>
<ejb-name>Pricer</ejb-name>
<home>examples.PricerHome</home>
<ejb-ref>
<description>
  This EJB reference says that the Pricing Engine session bean (Pricer) uses the Catalog
  Engine session bean (Catalog)
</description>
```

Comprendre les références EJB

```
<!--  
The nickname that Pricer uses to look up Catalog.We declare it so the deployer  
knows to bind the Catalog home in java:comp/env/ejb/CatalogHome.This may not  
correspond to the actual location to which the deployer binds the object via the  
container tools.The deployer may set up some kind of symbolic link to have the  
nickname point to the real JNDI location.  
-->  
<ejb-ref-name>ejb/CatalogHome</ejb-ref-name>  
<!--Catalog is a Session bean -->  
<ejb-ref-type>Session</ejb-ref-type>  
<!--The Catalog home interface class -->  
<home>examples.CatalogHome</home>  
<!--The Catalog remote interface class -->  
<remote>examples.Catalog</remote>  
<!-- (Optional) the Catalog ejb-name -->  
<ejb-link>Catalog</ejb-link>  
</ejb-ref>  
</session>  
</enterprise-beans>
```

Resources factories (JDBC, JMS...)

- Comment accéder à des ressources extérieures à l'EJB ?
 - Sources de données ?
 - JMS driver ?
- Pour utiliser une resource factory, il faut d'abord la localiser

```
//Obtain the initial JNDI context
Context initCtx =new InitialContext();
//Perform JNDI lookup to obtain resource factory
javax.sql.DataSource ds =(javax.sql.DataSource)
initCtx.lookup("java:comp/env/jdbc/ejbPool");
```

Resources factories (JDBC, JMS...)

- La localisation `java:comp/env/jdbc/ejbPool`
 - Les informations *réelles* doivent se trouver dans le descripteur (driver JDBC, URL de la *datasource*, etc...)

Resources factories (JDBC, JMS...)

```
<enterprise-beans>
<session>
<ejb-name>Catalog</ejb-name>
<home>examples.CatalogHome</home>
<!-- This element indicates a resource factory reference -->
<resource-ref>
  <description>
    This is a reference to a JDBC driver used within the Catalog bean.
  </description>
  <!-- The JNDI location that Catalog uses to look up the JDBC driver. We declare it
      so the deployer knows to bind the JDBC driver in java:comp/env/jdbc/ejbPool.-->
  <res-ref-name>jdbc/ejbPool</res-ref-name>
  <!-- The resource factory class -->
  <res-type>javax.sql.DataSource</res-type>
  <!-- Security for accessing the resource factory. Can either be "Container" or
      Application". -->
  <res-auth>Container</res-auth>
  <!-- Whether connections should be shared with other clients in the different
      transactions -->
  <res-sharing-scope>Sharable</res-sharing-scope>
</resource-ref>
```

Resources factories (JDBC, JMS...)

- Remarque sur <res-auth>..</res-auth>
- L'accès à la *datasource* peut nécessiter des autorisations (login, password),
 - On peut passer ces paramètres dans l'application,
 - Ou dans le descripteur (c'est donc le container qui va gérer l'authentification)
 - Cas le plus courant !

Le modèle de sécurité EJB

- Deux mesures de sécurité que les clients doivent satisfaire lorsqu'on ajoute la sécurité à un système EJB
 - 1. Le client doit *s'authentifier*
 - 2. Le client doit être *autorisé*

Le modèle de sécurité EJB

1. Le client doit s'*authentifier*

- Le mécanisme d'authentification vérifie que le client est bien celui qu'il prétend être,
- A l'aide de login/password validés auprès d'une base de données, de LDAP, etc...
- Une fois authentifié, on lui associe une *security identity* pour le reste de la session.

2. Le client doit être *autorisé*

- Une fois authentifié, il doit avoir la permission d'effectuer les opérations désirées.
- Par exemple tout le monde peut faire des offres de prêts, seul le banquier peut les accepter.

Etape 1 : l'authentification

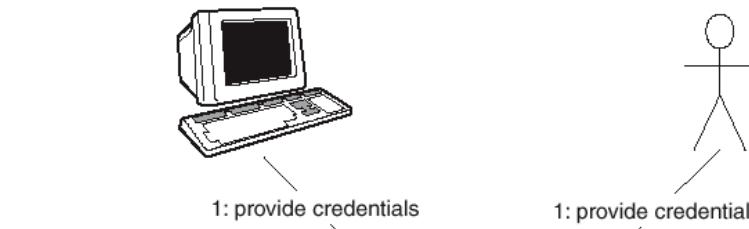
- Avec EJB 1.0 et 1.1, pas de mécanisme d'authentification portable
- EJB 2.0 propose un modèle portable et robuste
 - A l'aide de Java Authentication and Authorization Service (JAAS) : une API J2EE standard.

Présentation de JAAS

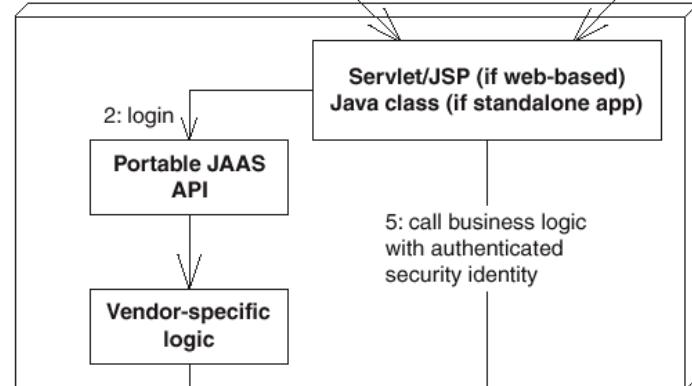
- JAAS est implémentée par le serveur d'application
- JAAS permet d'utiliser de manière transparente pour le développeur de bean n'importe quel système d'authentification sous-jacent.
 - Login/password, LDAP, certificats, simple lookup dans une BD, etc...

Présentation de JAAS

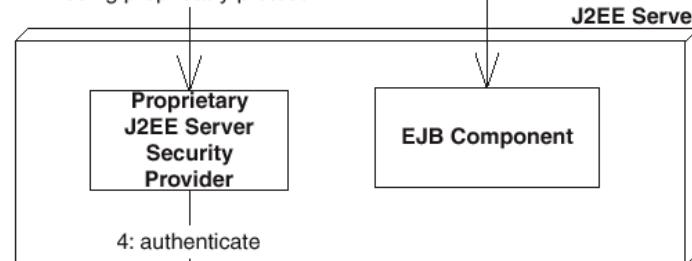
Web Browser (if Web-based application) User (if standalone app)



Client Machine



3: call J2EE server
using proprietary protocol

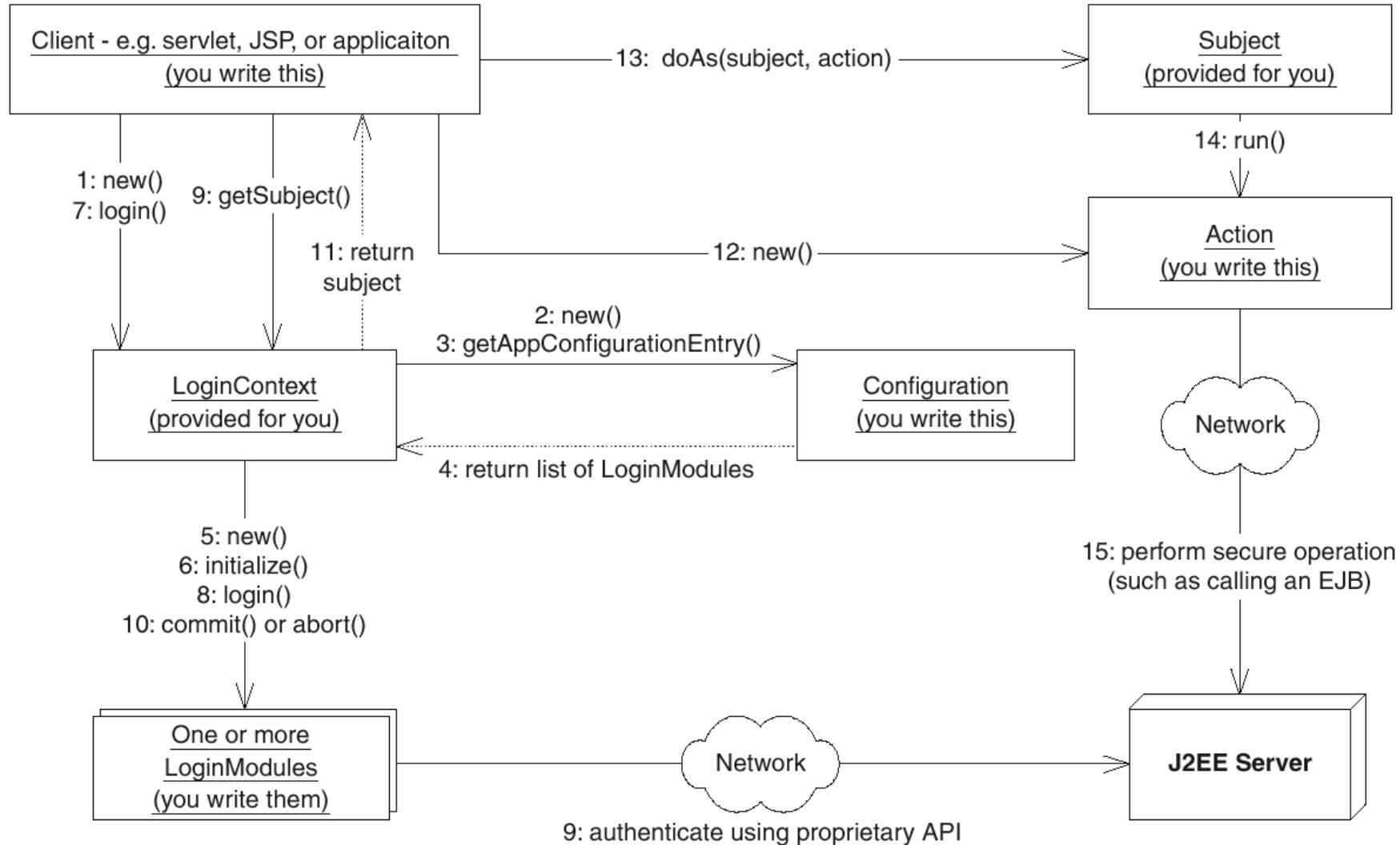


LDAP, RDBMS, home
grown, or other
existing security
system

Quelques commentaires

- Les informations fournies par le client, dans le cas d'un client web peuvent être de 4 types
 1. **Basic authentication** : login password en texte (idem .htaccess), certains serveurs permettent SSL
 2. **Form-based authentication** : idem sauf que login et password sont saisis dans un formulaire web
 3. **Digest authentication** : hascode calculé à partir de login+password+message HTTP lui-même. Le serveur web fait la vérif à partir d'un password qu'il à lui-même de son côté.
 4. **Certificate authentication** : protocole X509...

Architecture JAAS



Exemple de code JAAS

- Cet exemple montre un client qui s'authentifie avant d'appeler la méthode "hello world" d'un bean.
- Si le password n'est pas correct, une exception est levée

HelloClient.java

```
package examples;

import javax.naming.*;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.rmi.PortableRemoteObject;
public class HelloClient {

    public static void main(String [] args))throws Exception {
        /* Authenticate via JAAS */
        LoginContext loginContext =new LoginContext("Hello Client");
        loginContext.login();
        /* Retrieve the logged-in subject */
        Subject subject =loginContext.getSubject();
        /* Perform business logic while impersonating the authenticated subject */
        CallHelloWorld action =new CallHelloWorld();
        String result =(String)Subject.doAs(subject,action);
        /* Print the return result from the business logic */
        System.out.println(result);
    }
}
```

PasswordConfig.java (1)

```
package examples;

import java.util.Hashtable;

import javax.security.auth.login.*;

/** Sample configuration class for JAAS user authentication. This class is
 * useful because it can be rewritten to use different login modules without
 * affecting client code.

For example, we could have a login module that did username/password
authentication, and another that did public/private key certificate
authentication.*/

public class PasswordConfig extends Configuration {

    /** A configuration class must have a no-argument constructor */

    public PasswordConfig() {}
```

PasswordConfig.java (2)

```
/** This method chooses the proper login module. */

public AppConfigurationEntry [] getAppConfigurationEntry(String applicationName) {

    /* Return the one login module we 've written,which uses username/password
     authentication.

    -The "REQUIRED" flag says that we require that this login module succeed for
     authentication.

    -The new hashtable is a hashtable of options that our login module will
     receive. For example,we might define an option that turns debugging on.Our login
     module would inspect this hashtable and start logging output.*/

    AppConfigurationEntry []loginModules = new AppConfigurationEntry [1];
    loginModules[0] = new AppConfigurationEntry("examples.PasswordLoginModule",
                                                AppConfigurationEntry.LoginModuleControlFlag.REQUIRED,
                                                new Hashtable());
    return loginModules;
}

/** Refresh and reload the Configuration object by readingall of the login
 configurations again.*/

public void refresh(){}
}
```

PasswordLoginModule.java (1)

```
package examples;

import java.util.*;

import javax.naming.Context;

import javax.security.auth.*;

import javax.security.auth.callback.*;

import javax.security.auth.login.*;

import javax.security.auth.spi.*;

/** Sample login module that performs password authentication. The purpose of this
class is to actually go out and perform the authentication. */

public class PasswordLoginModule implements LoginModule {

    private Subject subject = null;

    /** Initializes us. We set ourselves to the particular subject which we will later
authenticate. */

    public void initialize(Subject subject, CallbackHandler callbackHandler, Map
                           sharedState, Map options) {
        this.subject =subject;
    }
}
```

PasswordLoginModule.java (2)

```
/** This method authenticates the user. It is called when the client tries to login in.  
Our method implementation contains the vendor-specific way to access our permanent  
storage of usernames and passwords.
```

Note that while this code is not portable, it is 100% hidden from your application
code behind the LoginModule.

The intention is that you develop a different LoginModule for each J2EE server.

In this case, BEA has provided us with a helper class that talks JNDI to the
Weblogic server, and the server then goes to whatever the currently configured
security realm is, such as a file, RDBMS, or LDAP server.*

```
public boolean login() throws LoginException {  
    try {  
        /* Authenticate the user 's credentials, populating Subject  
         Note: In a real application, we would not hardcode the username and ^p  
         password. Rather, we would write a reusable LoginModule that would work with any  
         username and password. We would then write a special callback handler that knows  
         how to interact with the user, such as prompting the user for a password. We would  
         then call that callback handler here. */  
        weblogic.jndi.Environment env = new  
            weblogic.jndi.Environment(System.getProperties());  
    }  
}
```

PasswordLoginModule.java (3)

```
env.setSecurityPrincipal("guest");
env.setSecurityCredentials("guest");
weblogic.security.auth.Authenticate.authenticate(env,subject);
/* Return that we have successfully authenticated the subject* /
return true;
} catch (Exception e){
    throw new LoginException(e.toString());
}
}

/** This method is called if the overall authentication succeeded (even if this
particular login module failed). This could happen if there are other login
modules involved with the authentication process. This is our chance to
perform additional operations, but since we are so simple,we don 't do
anything.
@return true if this method executes properly */
public boolean commit()throws LoginException {
    return true;
}
```

PasswordLoginModule.java (4)

```
/** This method is called if the overall authentication failed (even if
this particular login module succeeded).This could happen if there
are other login modules involved with the authentication
process.

This is our chance to perform additional operations, but since we are so
simple,we don 't do anything.

@return true if this method executes properly */

public boolean abort()throws LoginException {
    return true;
}

/** Logout the user.

@return true if this method executes properly */

public boolean logout()throws LoginException {
    return true;
}
```

CallHelloWorld.java (1)

```
package examples;

import java.security.*;
import javax.naming.*;
import java.util.Hashtable;
import javax.rmi.PortableRemoteObject;

/** This is a helper class that knows how to call a "Hello,World!" bean. It does so
 * in a secure manner, automatically propagating the logged in security context
 * to the J2EE server. */

public class CallHelloWorld implements PrivilegedAction {

    /* This is our one business method. It performs an action securely, and
     * returns application-specific results.*/
    public Object run(){

        String result ="Error";

        try {
            /* Make a bean */

            Context ctx =new InitialContext(System.getProperties());
            Object obj =ctx.lookup("HelloHome");

```

CallHelloWorld.java (2)

```
    HelloHome home =(HelloHome)
        PortableRemoteObject.narrow(obj,HelloHome.class);
    Hello hello =home.create();
    /* Call a business method,propagating the security context */
    result =hello.hello();
} catch (Exception e) {
    e.printStackTrace();
}
/* Return the result to the client */
return result;
}
```

Etape 2 : l'autorisation

- Une fois identifié, le client doit passer un test d'autorisation
- On force cette étape en définissant la police de sécurité pour le bean (*security policies*)
- Deux manière de faire
 1. Par programmation
 - On effectue les tests dans le code du bean...
 2. Par autorisation déclarative
 - Le container effectue les tests...

Rôles de sécurité (*security roles*)

- Rôle de sécurité = une collection d'identités pour le client
- Un client est autorisé à effectuer une opération
 - Son identité doit être dans le bon rôle de sécurité pour cette opération,
 - Le déployeur de beans a la responsabilité d'associer les identités et les rôles de sécurité après que vous ayez écrit le bean,
 - Grande portabilité et flexibilité.

SECURITY ROLE	VALID IDENTITIES
employees	EmployeeA, EmployeeB
managers	ManagerA
administrators	AdminA

Autorisation par programmation

■ Étape 1 : écrire la logique de sécurité

- Savoir qui appelle le bean,
- On obtient l'information via l'objet EJBContext

```
public interface javax.ejb.EJBContext {  
    ...  
    public java.security.Principal getCallerPrincipal();  
    public boolean isCallerInRole(String roleName);  
    ...  
}
```

Autorisation par programmation

■ Exemple d'utilisation de `isCallerInRole()`

```
public class EmployeeManagementBean implements SessionBean {  
    private SessionContext ctx;  
    ...  
    public void modifyEmployee(String employeeID) throws SecurityException {  
        /* If the caller is not in the 'administrators' security role, throw an  
         * exception.*/  
        if (!ctx.isCallerInRole("administrators")) {  
            throw new SecurityException(...);  
        }  
        // else, allow the administrator to modify the  
        // employee records  
        //...  
    }  
}
```

Autorisation par programmation

Exemple d'utilisation de `getCallerPrincipal()`

```
import java.security.Principal;  
...  
public class EmployeeManagementBean implements SessionBean {  
    private SessionContext ctx;  
    ...  
    public void modifyEmployee(){  
        Principal id = ctx.getCallerIdentity();  
        String name = id.getName();  
        // Query a database based on the name to determine if the user is  
        // authorized  
    }  
}
```

Autorisation par programmation

- Étape 2 : décrire les rôles de sécurité abstraits que le bean va utiliser
 - Par exemple indiquer que le bean a un rôle "administrator",
 - On indique ceci dans le descripteur de déploiement.

Autorisation par programmation

```
<enterprise-beans>
<session>
    <ejb-name>EmployeeManagement</ejb-name>
    <home>examples.EmployeeManagementHome</home>
    ...
    <!--
        This declares that our bean code relies on
        the administrators role; we must declare it here
        to inform the application assembler and deployer.
    -->
    <security-role-ref>
        <description>
            This security role should be assigned to the administrators who are
            responsible for
            modifying employees.
        </description>
        <role-name>administrators</role-name>
    </security-role-ref>
    ...
</session>
...
</enterprise-beans>
```

Autorisation par programmation

- Étape 3 : associer les rôles de sécurité abstraits avec des rôles concrets (*actual*)
 - Autre niveau d'indirection, permet au déployeur de beans de renommer les rôles...

Autorisation par programmation

```
...
<enterprise-beans>
<session>
<ejb-name>EmployeeManagement</ejb-name>
<home>examples.EmployeeManagementHome</home>
...
<security-role-ref>
<description>
This security role should be assigned to the administrators who are responsible for
modifying employees.
</description>
<role-name>administrators</role-name>
<!-- Here we link what we call "administrators" above, to a real security-role, called
    "admins", defined below
-->
<role-link>admins</role-link>
</security-role-ref>
...
</session>
<assembly-descriptor>
...

```

Autorisation par programmation

```
<!--  
This is an example of a real security role.  
-->  
  
<security-role>  
    <description>  
        This role is for personnel authorized to perform  
        employee administration.  
    </description>  
    <role-name>admins</role-name>  
  
</security-role>  
...  
  
</assembly-descriptor>  
  
</enterprise-beans>  
...
```

Autorisation déclarative

- Tout se passe dans le descripteur, au déploiement.
 - Le container fait tout le travail !
- Étape 1 : déclarer les permissions associées aux méthodes du bean que l'on désire sécuriser
 - Le container générera le code qui fera les vérifications nécessaires dans l'EJB home et dans l'EJB object

Exemple de déclaration d'autorisation (1)

```
<assembly-descriptor>
```

```
<!-- You can set permissions on the entire bean.
```

Example: Allow role "administrators" to call every method on the bean class.

```
-->
```

```
<method-permission>
```

```
    <role-name>administrators</role-name>
```

```
    <method>
```

```
        <ejb-name>EmployeeManagement</ejb-name>
```

```
        <method-name>*</method-name>
```

```
    </method>
```

```
</method-permission>
```

```
...
```

Exemple de déclaration d'autorisation (2)

```
<!--
```

You can set permissions on a method level. Example: Allow role "managers" to call method "modifySubordinate()" and "modifySelf()". -->

```
<method-permission>

    <role-name>managers</role-name>

    <method>

        <ejb-name>EmployeeManagement</ejb-name>

        <method-name>modifySubordinate</method-name>

    </method>

    <method>

        <ejb-name>EmployeeManagement</ejb-name>

        <method-name>modifySelf</method-name>

    </method>

</method-permission>
```

Exemple de déclaration d'autorisation (3)

```
<!-- If you have multiple methods with the same name but that take  
different parameters, you can even set permissions that distinguish  
between the two. Example:allow role "employees" to call method  
"modifySelf(String)" but not modifySelf(Int) -->  
<method-permission>  
<role-name>employees</role-name>  
<method>  
    <ejb-name>EmployeeManagement</ejb-name>  
    <method-name>modifySelf</method-name>  
    <method-params>String</method-params>  
  </method>  
</method-permission>
```

Exemple de déclaration d'autorisation (4)

```
<!--
```

This is the list of methods that we don't want ANYONE to call. Useful if you receive a bean from someone with methods that you don't need. -->

```
<exclude-list>
```

```
    <description>
```

We don't have a 401k plan, so we don't support this method.

```
    </description>
```

```
    <method>
```

```
        <ejb-name>EmployeeManagement</ejb-name>
```

```
        <method-name>modify401kPlan</method-name>
```

```
        <method-params>String</method-params>
```

```
    </method>
```

```
</exclude-list>
```

```
...
```

```
</assembly-descriptor>
```

Autorisation déclarative

■ Étape 2 : déclarer les rôles de sécurité

- Même méthode que pour les autorisation par programmation...
- On les déclare et on donne une description (optionnelle) pour le déployeur.

Déclarer les rôles de sécurité

```
<assembly-descriptor>
  <security-role>
    <description> System administrators </description>
    <role-name>administrators</role-name>
  </security-role>
  <security-role>
    <description> Employees that manage a group </description>
    <role-name>managers</role-name>
  </security-role>
  <security-role>
    <description> Employees that don 't manage anyone
    </description>
    <role-name>employees</role-name>
  </security-role>
  ...
</assembly-descriptor>
```

Que choisir ? Déclarative ou par programmation ?

- Dans un monde idéal, on utiliserait tout le temps la méthode déclarative
 - Externalisation de la politique de sécurité,
 - Plus facile à maintenir et à ajuster,
 - Le travail du déployeur, pas du développeur !
- Malheureusement, dans la spécification EJB 2.0 il manque encore
 - La gestion de la sécurité par instance,
 - Ex : un bean banquier ne peut modifier que des instances de comptes bancaires dont le solde est inférieur à 50.000 francs. (seul le chef peut manipuler les comptes bien fournis!)
 - Dans ce cas, recourir à la gestion par programmation.

Propagation de la sécurité

- Un client a passé tous les tests de sécurité pour appeler le bean A.
- La méthode de A qu'il appelle appelle une méthode d'un bean B,
- Le client doit-il encore passer les tests de sécurité pour B ?
- En coulisse, le container gère un contexte de sécurité (*Security Context*), qu'il passe de stub et skeleton, d'appel de méthode en appel de méthode...

Propagation de la sécurité

```
<enterprise-beans>
  <session>
    <ejb-name>EmployeeManagement</ejb-name>
    <home>examples.EmployeeManagementHome</home>
    <security-identity>
      <run-as>
        <role-name>admins</role-name>
      </run-as>
    </security-identity>
  </session>
  <assembly-descriptor>
    ...
    <security-role>
      <description> This role is for personnel authorized to perform employee
                    Administration.
      </description>
      <role-name>admins</role-name>
    </security-role>
    ...
  </assembly-descriptor>
</enterprise-beans>
```

Un mot sur les Message-Driven Beans

- Pas de gestion d'autorisation pour les MDB, ni déclarative, ni par programmation
- Les MDB ne reçoivent pas d'appel de méthode par RMI-IIOP, ils reçoivent des *messages*,
- Ces messages ne contiennent pas d'information liées à la sécurité (*credentials*)
- Un MDB ne peut pas non plus propager un contexte de sécurité qui n'existe pas,
- Ils peuvent néanmoins avoir une *identité* lorsqu'ils appellent d'autres beans.

Comprendre les *EJB Object Handles*

- Dans certaines applications, le client d'un bean peut se déconnecter, puis se reconnecter plus tard *en retrouvant le bean dans l'état !*
- Exemple : un caddy représenté par un session bean stateful...
 - le client décide de se déconnecter... lorsqu'il se reconnecte il veut retrouver son caddy dans l'état !

Comprendre les *EJB Object Handles*

- La spécification EJB fournit un outil particulier pour gérer ce genre de situations : *l'EJB object handles*
 - C'est un proxy vrs un EJB object,
 - Longue durée de vie,

Exemple d'utilisation d'un *EJB Object Handle*

```
//First, get the EJB object handle from the EJB object.

javax.ejb.Handle myHandle =myEJBObject.getHandle();

// Next, serialize myHandle, and then save it in permanent storage.

ObjectOutputStream stream =...;

stream.writeObject(myHandle);

// time passes... When we want to use the EJB object again, deserialize
// the EJB object handle

ObjectInputStream stream =...;

Handle myHandle =(Handle) stream.readObject();

// Convert the EJB object handle into an EJB object

MyRemoteInterface myEJBObject =(MyRemoteInterface)

javax.rmi.PortableRemoteObject.narrow(myHandle.getEJBObject(),
MyRemoteInterface.class);

// Resume calling methods again

myEJBObject.callMethod();
```

Variante : les *EJB Home Handles*

- Idem, mais pour rendre persistant une référence vers le *home object* d'un bean,
- Peu utilisés...

Exemple d'utilisation d'un *EJB Home Handle*

```
//First, get the EJB home handle from the home object.  
  
javax.ejb.HomeHandle homeHandle =myHomeObject.getHomeHandle();  
  
//Next, serialize the home handle, and then save it in permanent storage.  
  
ObjectOutputStream stream =...;  
  
stream.writeObject(homeHandle);  
  
// time passes... When we want to use the home object again, deserialize the home  
// handle  
  
ObjectInputStream stream =...;  
  
javax.ejb.HomeHandle homeHandle =(HomeHandle) stream.readObject();  
  
// Convert the home object handle into a home object  
  
MyHomeInterface myHomeObject =(MyHomeInterface)  
  
        javax.rmi.PortableRemoteObject.narrow(homeHandle.getHomeObject(),  
                                         MyHomeInterface.class);  
  
// Resume using the home object  
  
myHomeObject.create();
```

Gestion des transactions

Gestion de transactions

- Service clé pour le développement côté serveur,
- Permet à des applications de fonctionner de manière robuste,
- Les transactions sont très utiles lorsqu'on effectue des opérations de persistance, comme des mises-à-jours dans une base de données...
- Dans ce chapitre nous présentons le concept de transaction et sa mise en application au sein des EJB.

Motivation pour les transactions

- Opérations atomiques, bien que composées de plusieurs petites opérations
 - Ex : transfert de compte à compte bancaire : on enlève sur un compte, on dépose sur l'autre...
 - Si 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* !

Traitemen~~t~~ par exceptions

■ On peut s'en sortir en gérant des exceptions

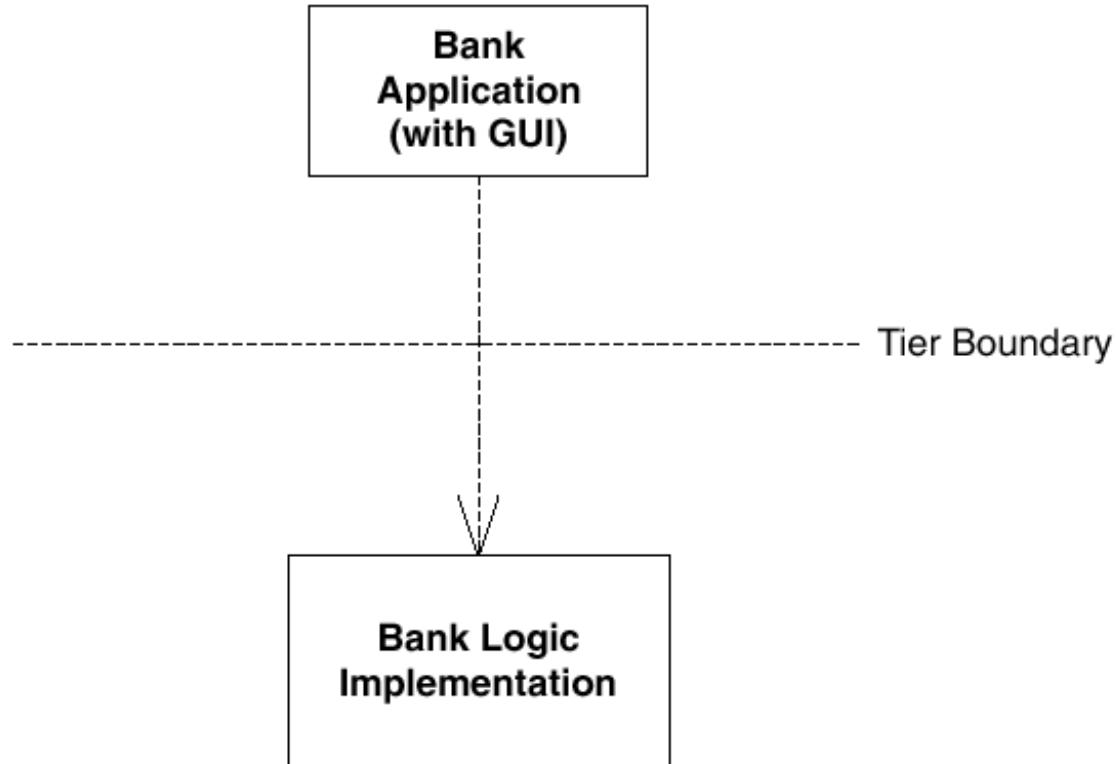
```
try {  
    // retirer de l'argent du compte 1  
} catch (Exception e) {  
    // Si une erreur est apparue, on arrête.  
    return;  
}  
  
try {  
    // Sinon, on dépose l'argent sur le compte 2  
} catch (Exception e) {  
    // Si une erreur est apparue, on s'arrête, mais avant, on redépose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

■ Qu'en pensez-vous ?

Traitements par exceptions

- Et si on échoue la dernière opération et qu'on arrive pas à remettre l'argent sur le compte 1 ?
- 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 ?

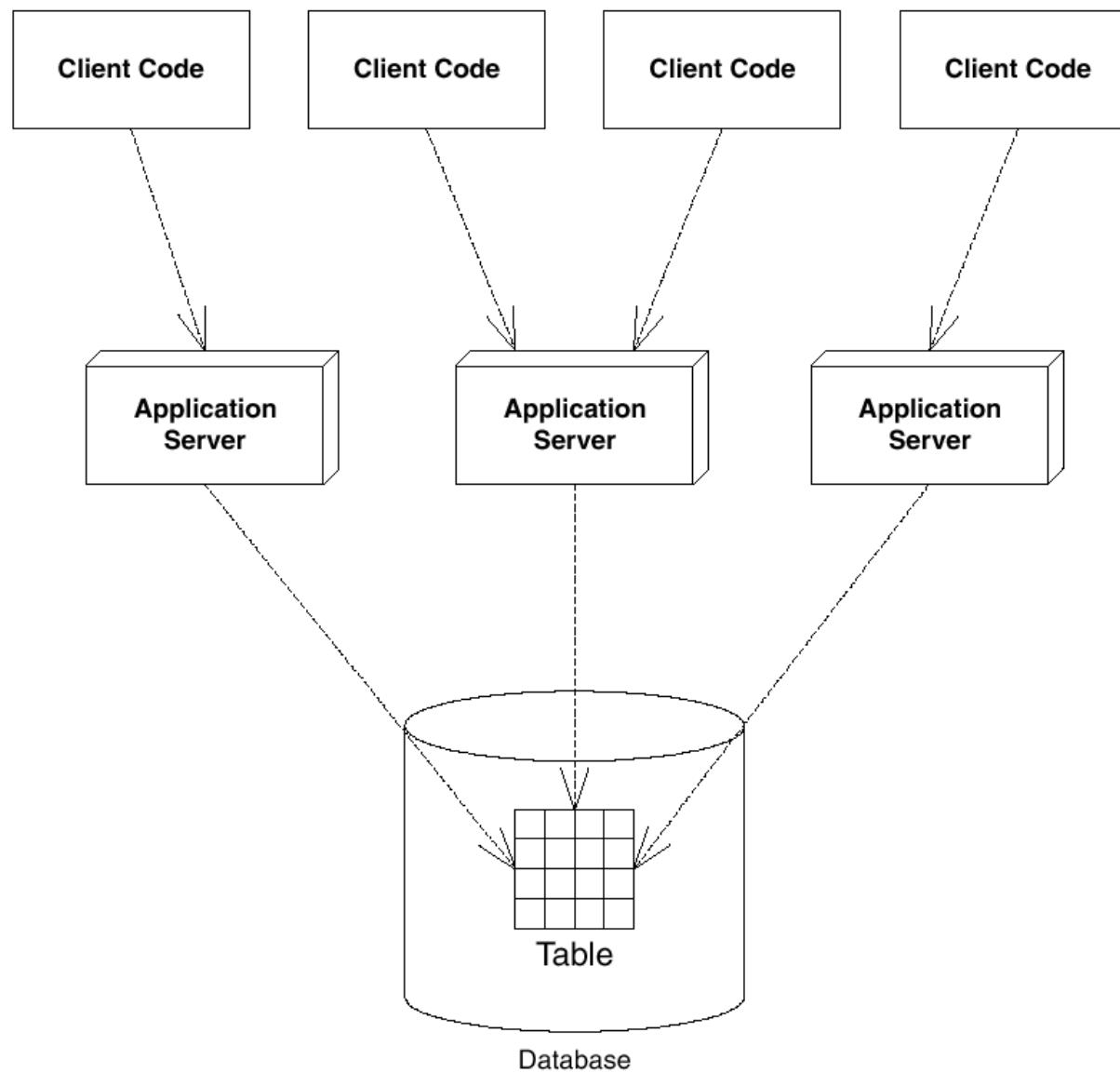
Panne réseau ou panne machine



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 !

Problèmes résolus par les *transactions* !

- Un transaction est une série d'opérations qui apparaissent sous la forme d'une large opération atomique.
- Soit la *transaction* réussit, soit elle échoue.
 - Traduire par "toutes les opérations qui la composent..."
- Les transactions s'accordent avec les pannes machines ou réseau,
- Répondent au problèmes du partage de données.

Un peu de vocabulaire

- **Objet ou composant transactionel**
 - Un composant bancaire impliqué dans une transaction. Un EJB compte bancaire, un composant .NET, CORBA...
- **Gestionnaire de transaction**
 - Celui qui en coulisse gère l'état de la transaction
- **Ressource**
 - L'endroit où on lit et écrit les données : un DB, une queue de messages, autre...
- **Gestionnaire de ressource**
 - Driver d'accès à une BD, à une queue de message...
Implémentent l'interface X/Open XA, standard *de facto* pour la gestion de transactions...

Les propriété ACID

■ Atomicité

- Nombreux acteurs : servlet, corba, rmi-iiop, ejbs, DB... Tous votent pour indiquer si la transaction s'est bien passée...

■ Consistance

- Le système demeure consistent après l'exécution d'une transaction (comptes bancaires ok!)

■ Isolation

- Empêche les transactions concurrentes de voir des résultats partiels. Chaque transaction est *isolée* des autres.
- Implémenté par des protocoles de synchronisation bas-niveau sur les BDs...

■ Durabilité

- Garantit que les mises à jour sur une BD peuvent survivre à un crash (BD, machine, réseau)
- En général, on utilise un fichier de log qui permet de faire des *undos* pour revenir dans l'état avant le crash.

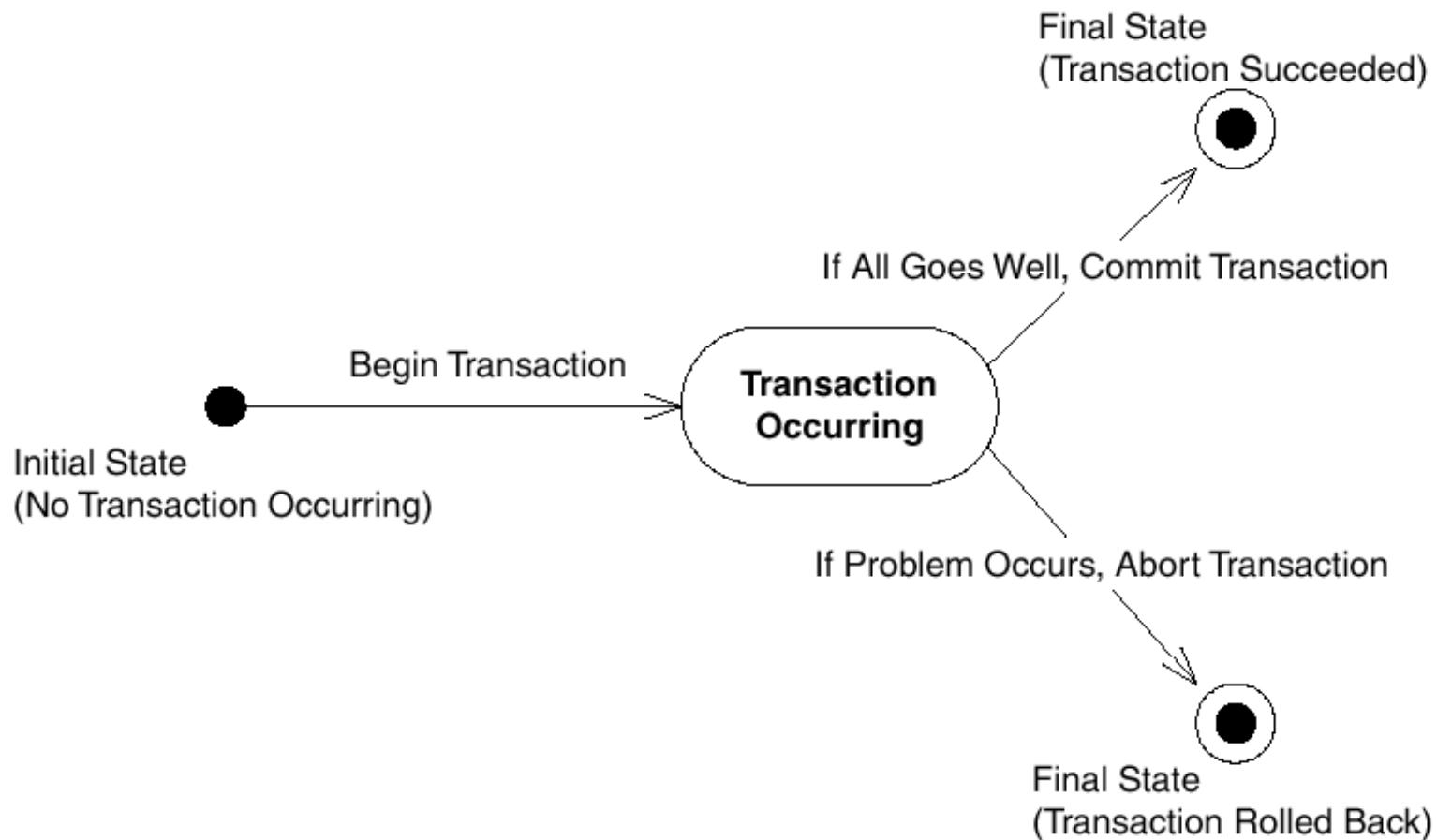
Modèles de transactions

- 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 commitée (*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...

Flat transactions



Comment s'effectue le *rollback* ?

- Tant qu'il n'y a pas de commit, les modifications à une BD ne sont pas permanentes...
- Dans un contexte de composants, *tout est fait en coulisse*. Vous n'avez pas à vous préoccuper de l'état de votre bean... Il sera restauré en cas de rollback.

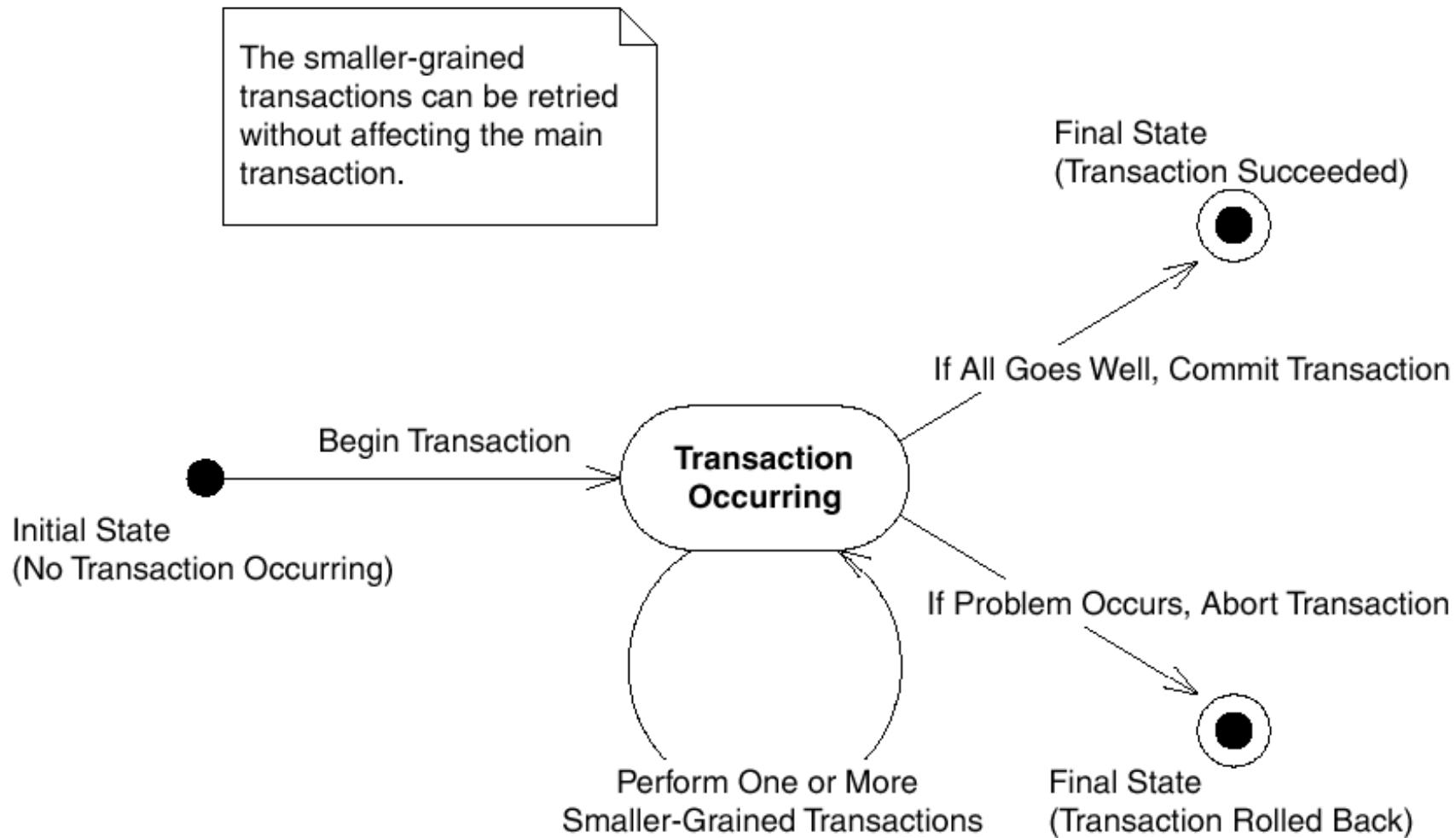
Transactions imbriquées

- Cas d'école : on veut faire un tour du monde
 - 1. Notre application achète un billet de train de Nice 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...

Transactions imbriquées

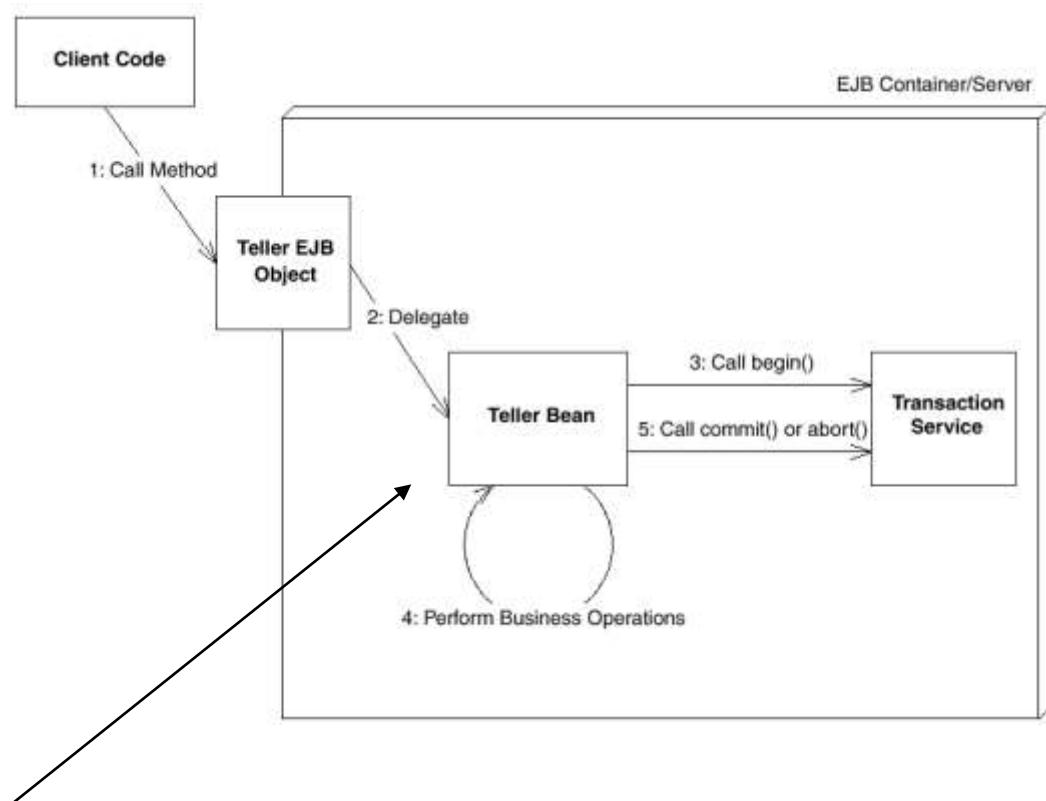


Gestion des transactions avec les EJBs

- 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,
 2. De manière déclarative,
 3. De manière initiée par le client.

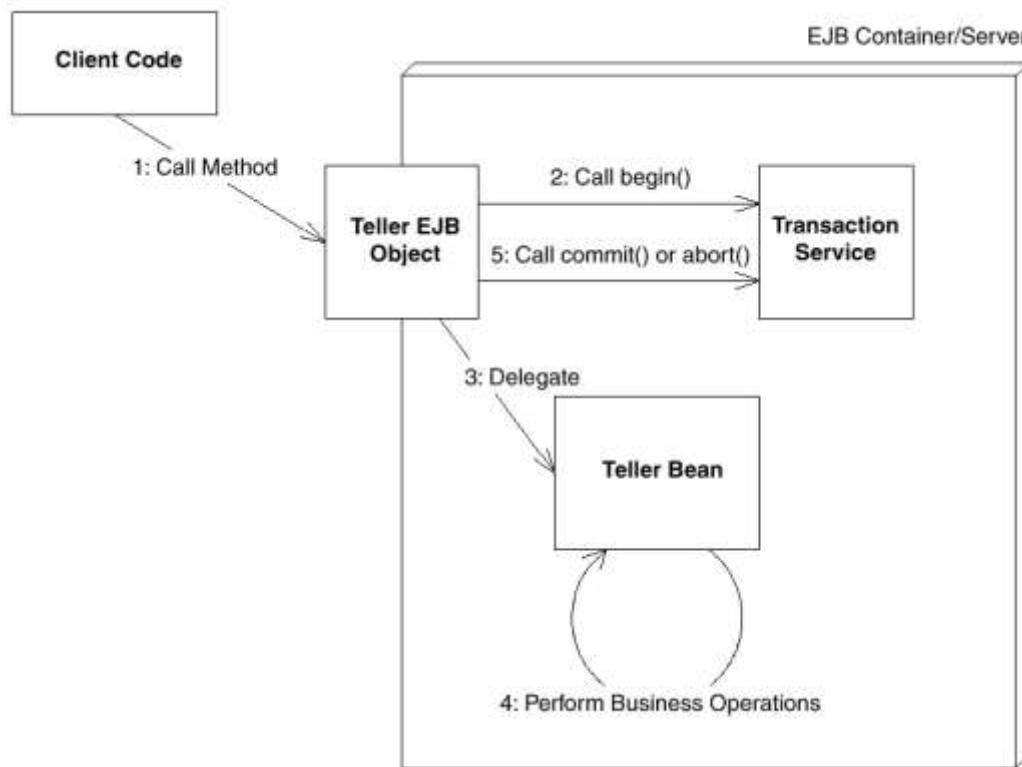
Gestion des transactions par programmation

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



Gestion des transactions déclarative

- Le bean est automatiquement enrôlé (*enrolled*) dans une transaction...
- Le container fait le travail...



Gestion des transactions déclarative

■ Génial pour le développeur!

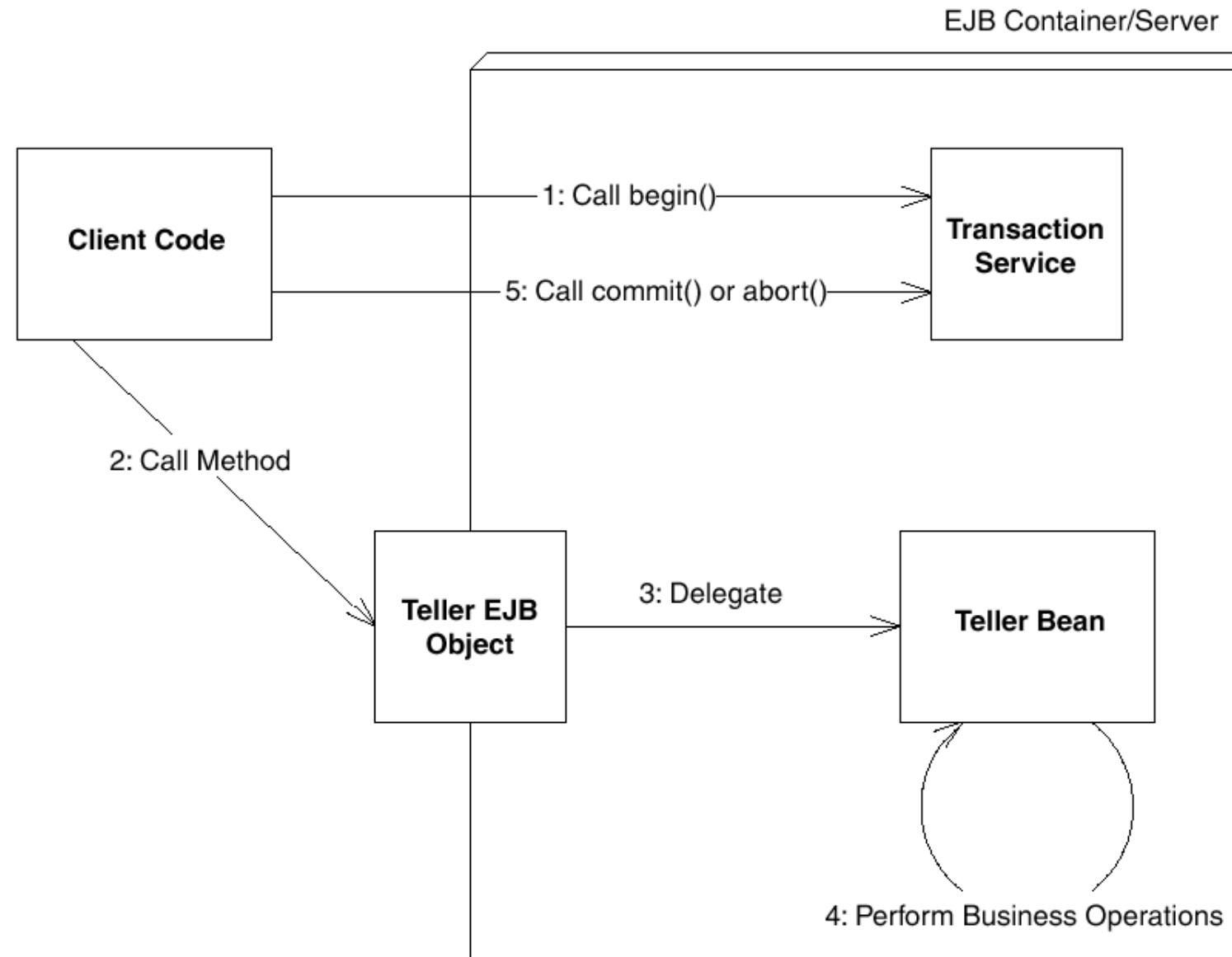
```
import javax.ejb.*;
import javax.annotation.Resource;
import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;

@Stateless()
@TransactionalManagement(javax.ejb.TransactionManagementType.CONTAINER)
public class TellerBean implements Teller {
    @PersistenceContext private EntityManager em;
    @Resource private SessionContext ctx;

    @TransactionAttribute(javax.ejb.TransactionAttributeType.REQUIRED)
    public void transferFunds(float amount, String fromAccount, String
                               toAccount)
    {
        // Lookup for accts with the provided account IDs

        BankAccount acct1 = em.find(BankAccount.class, fromAccount);
```

Transactions initiées par le client



Que choisir ?

- Par programmation : contrôle très fin possible...
- Déclaratif : super, 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 déetecter les crashes machine, réseau, etc...

Transactions et entity beans

- Dans une transaction, on accède à un entity bean :
 - À un moment donné : chargement des données de la DB dans le bean,
 - Un verrou (*lock*) est acquis sur la DB, assurant la consistance,
 - Juste après l'appel du commit de la transaction, **les données sont sauvegardées**, la DB est mise à jour et libère le verrou.
 - On a donc le code d'accès à la BD, le code des méthodes métiers et l'appel à **la sauvegarde dans la BD** qui se trouvent dans la transaction.

Transactions et entity beans

- Si on contrôlait la transaction dans le bean
 - Démarrer la transaction avant l'accès à la BD,
 - Faire le commit à la fin, après la sauvegarde,
- Problème : on a pas le contrôle sur ce code...
- *Bean-Managed transactions illégales pour les entity beans. Seules les transactions gérées par le container sont autorisées !*

Transactions et entity beans

- Conséquence : un entity bean n'accède pas à la BD à chaque appel de méthode, mais à chaque transaction.
- Si un entity s'exécute trop lentement, il se peut qu'une transaction intervient pour chaque appel de méthode, impliquant des accès BD.
- Solution : inclure plusieurs appels de méthodes de l'entity dans une même transaction.
- Se fait en précisant *les attributs de transaction du bean*.

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...
258

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.

Attributs de transactions gérées par le container

- Pour chaque bean, on précise des *attributs de transaction*
 - On peut spécifier des attributs pour le bean entier ou méthode par méthode,
 - On peut préciser les deux... Le plus restrictif gagne.
 - Chaque méthode métier doit être traitée (globalement ou individuellement)

Par défaut : attribut = REQUIRED

■ Génial pour le développeur!

```
import javax.ejb.*;
import javax.annotation.Resource;
import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;

@Stateless()
@TransactionalManagement(javax.ejb.TransactionManagementType.CONTAINER)
public class TellerBean implements Teller {
    @PersistenceContext private EntityManager em;
    @Resource private SessionContext ctx;

    @TransactionAttribute(javax.ejb.TransactionAttributeType.REQUIRED)
    public void transferFunds(float amount, String fromAccount, String
                               toAccount)
    {
        // Lookup for accts with the provided account IDs

        BankAccount acct1 = em.find(BankAccount.class, fromAccount);
```

Valeur des attributs de transaction

■ **Required**

- Le bean est toujours dans une transaction.
- Si une transaction pour ce bean existe, alors le bean la rejoint (*join*), sinon, le container crée une nouvelle transaction.
- La plupart du temps proposé comme valeur par défaut par les IDEs et leurs Wizards/éditeurs de descripteurs...

Attribut de transaction : Required

■ Exemple : passage d'une commande

- Un session bean utilise deux entity beans : un bean carte de crédit et bean commande,
- Lors d'un passage de commande, on envoie la commande, puis on débite la carte de crédit,
- Si le bean session a comme attribut de transaction *Required*, une transaction est créée dès que le passage de commande démarre,
- Lors de l'appel au bean Commande, si celui-ci est également en mode *Required*, il rejoindra la transaction en cours. Idem lors de l'appel au bean Carte de Crédit,
- Si un problème se pose, la carte de crédit ne sera pas débitée et la commande ne sera pas passée.

Attribut de transaction : RequiresNew

■ **RequiresNew**

- Le bean est toujours dans une nouvelle transaction.
 - Si une transaction existe, elle est suspendue,
 - Lorsque la nouvelle transaction se termine (*abort* ou *commit*), l'ancienne transaction est résumée.
-
- Utile si on veut respecter les propriétés ACID dans l'unité du bean, sans qu'une logique externe intervienne.

Attribut de transaction : Supports

■ **Supports**

- Semblable à *Required* sauf que si une transaction n'existe pas, elle n'est pas créée.
- Si l'exécution du bean intervient dans une transaction existante, il la rejoint néanmoins.

■ A éviter pour les applications *mission-critical* !

- Dans ce cas, choisir *Required*.

Attribut de transaction : Mandatory

■ Mandatory

- Une transaction doit exister lorsque le bean est exécuté.
- Si ce n'est pas le cas,
`javax.ejb.TransactionRequiredException` est levée et renvoyée au client.
- Si le client est local, c'est
`javax.ejb.TransactionRequiredLocalException` qui est levée...
- Attribut sûr, qui oblige le client à inscrire sa logique dans une transaction avant d'appeler le bean.

Attribut de transaction : NotSupported

■ **NotSupported**

- Le Bean ne supporte pas les transactions,
- Si une transaction existe lors de l'appel du bean, la transaction est suspendue, le bean s'exécute, puis la transaction est résumée.
- Utiliser cet attribut lorsque les propriétés ACID ne sont pas importantes...
 - Exemple : un bean qui fait des statistiques toutes les dix minutes en parcourant une BD. On tolère que les données lues ne sont peut-être pas à jour...
- Gain en performance évident.

Attribut de transaction : Never

■ **Never**

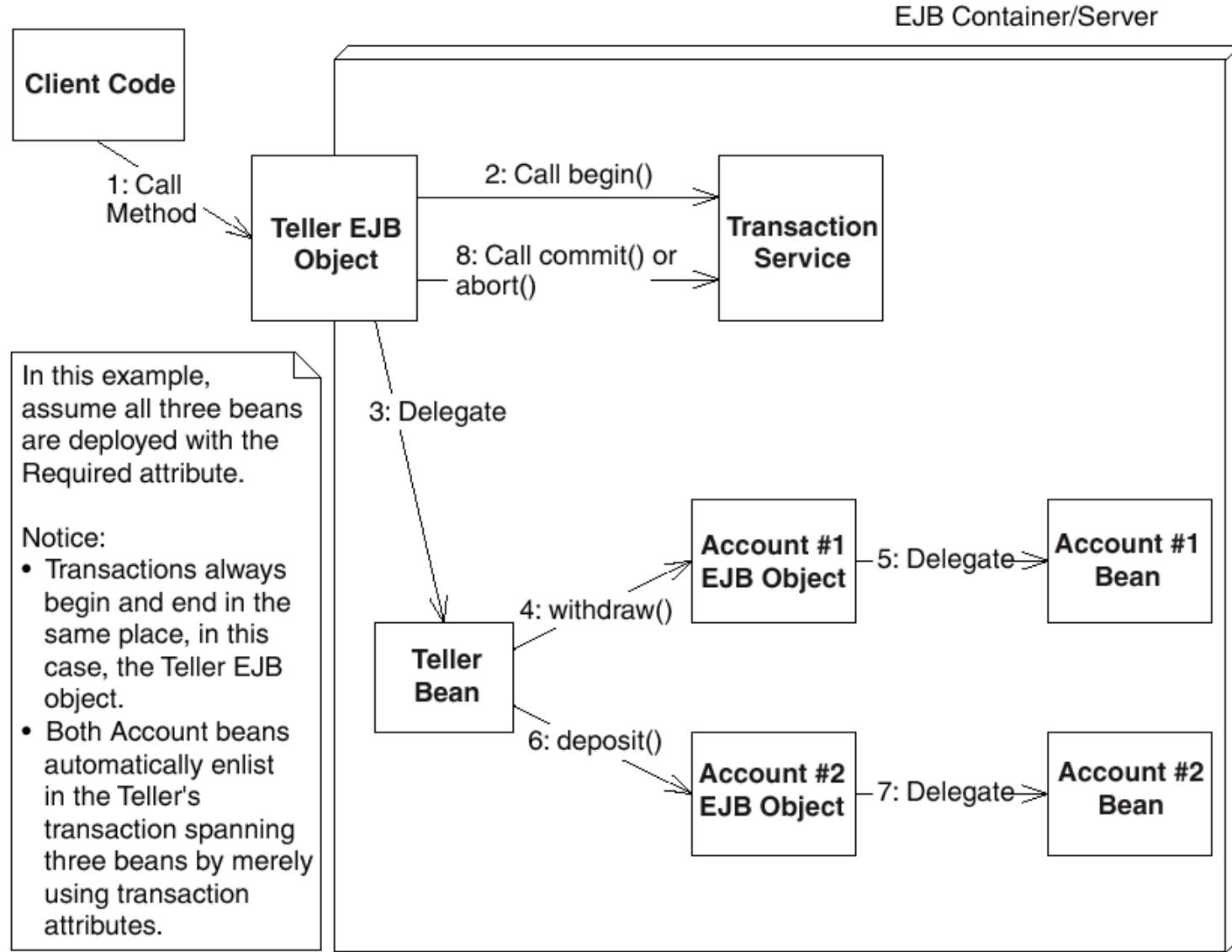
- Le Bean ne supporte pas les transactions,
- Une exception `javax.rmi.RemoteException` ou `javax.ejb.EJBException` est envoyée au client si une transaction existe au moment de l'appel du bean.
- A utiliser lors du développement d'une logique non transactionnelle.

Résumé sur les attributs de transaction

- Soit T1 une transaction initiée par le client, T2 la nouvelle transaction initiée par le bean appelé.

TRANSACTION ATTRIBUTE	CLIENT'S TRANSACTION	BEAN'S TRANSACTION
Required	none	T2
	T1	T1
RequiresNew	none	T2
	T1	T2
Supports	none	none
	T1	T1
Mandatory	none	error
	T1	T1
NotSupported	none	none
	T1	none
Never	none	none
	T1	error

Début et fin d'une transaction



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

TRANSACTION ATTRIBUTE	STATELESS SESSION 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 nombreux partis sont impliqués : 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...
- Mortel à utiliser !

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 javax.transaction.UserTransaction

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

L'interface javax.transaction.UserTransaction

METHOD	DESCRIPTION
begin()	Begins a new transaction. This transaction becomes associated with the current thread.
commit()	Runs the two-phase commit protocol on an existing transaction associated with the current thread. Each resource manager will make its updates durable.
getStatus()	Retrieves the status of the transaction associated with this thread.
rollback()	Forces a rollback of the transaction associated with the current thread.
setRollbackOnly()	Calls this to force the current transaction to roll back. This will eventually force the transaction to abort.
setTransactionTimeout(int)	The <i>transaction timeout</i> is the maximum amount of time that a transaction can run before it's aborted. This is useful to avoid deadlock situations, when precious resources are being held by a transaction that is currently running.

Constantes de la classe

javax.transaction.Status

■ 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;  
}
```

Constantes de la classe

`javax.transaction.Status`

CONSTANT	MEANING
STATUS_ACTIVE	A transaction is currently happening and is active.
STATUS_NO_TRANSACTION	No transaction is currently happening.
STATUS_MARKED_ROLLBACK	The current transaction will eventually abort because it's been marked for rollback. This could be because some party called <i>UserTransaction.setRollbackOnly()</i> .
STATUS_PREPARING	The current transaction is preparing to be committed (during Phase One of the two-phase commit protocol).
STATUS_PREPARED	The current transaction has been prepared to be committed (Phase One is complete).
STATUS_COMMITTING	The current transaction is in the process of being committed right now (during Phase Two).
STATUS_COMMITED	The current transaction has been committed (Phase Two is complete).
STATUS_ROLLING_BACK	The current transaction is in the process of rolling back.
STATUS_ROLLEDBACK	The current transaction has been rolled back.
STATUS_UNKNOWN	The status of the current transaction cannot be determined.

Exemple de transaction gérée par programmation

```
import javax.ejb.*;
import javax.annotation.Resource;
import javax.persistence.PersistenceContext;
import javax.persistence.EntityManager;
import javax.transaction.UserTransaction;

@Stateless()
@TransactionManagement(javax.ejb.TransactionManagementType.BEAN)
public class TellerBean implements Teller {
    @PersistenceContext private EntityManager em;
    @Resource private javax.transaction.UserTransaction userTx;

    public void transferFunds(float amount, String fromAccount, String
                               toAccount)
    {
        // Lookup for accts with the provided account Ids
        try {
            userTx.begin();

            BankAccount acct1 = em.find(BankAccount.class, fromAccount);
            BankAccount acct2 = em.find(BankAccount.class, toAccount);

            if (acct1.balance < amount)
                userTx.rollback();

            acct1.withdraw(amount);
            acct2.deposit(amount);
        }
        finally {
            userTx.commit();
        }
    }
}
```



Exemple de transaction gérée par programmation (suite)

```
if (acct1.balance < amount)
    userTx.rollback();

acct1.withdraw(amount);
acct2.deposit(amount);

em.persist(acct1);
em.persist(acct2);

userTx.commit();
} catch (Exception e) {
    System.out.println("Exception occurred during transfer of
                      funds." + e.getMessage());
}
}
```

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 {  
    /* 1: Set environment up. You must set the JNDI Initial Context factory, the  
       Provider URL, and any login names or passwords necessary to access JNDI.  
       See your application server product's documentation for details on their  
       particular JNDI settings. */  
  
    java.util.Properties env = ...  
    /* 2: Get the JNDI initial context */  
    Context ctx = new InitialContext(env);  
    /* 3: Look up the JTA UserTransaction interface via JNDI. The container is  
       required to make the JTA available at the location java:comp/UserTransaction. */  
    userTran =  
        (javax.transaction.UserTransaction)ctx.lookup("java:comp/UserTransaction");  
    /* 4: Execute the transaction */  
    userTran.begin();  
    // perform business operations  
    userTran.commit();  
} catch (Exception e) {  
    // deal with any exceptions, including ones  
    // indicating an abort.  
}
```

Isolation de transaction

- Le "I" de ACID !
- L'isolation coûte cher en termes de performances,
- Nécessité de pouvoir régler le niveau d'isolation entre transactions concurrentes !

Isolation de transaction

- Un exemple : deux instances A et B d'un même composant accèdent à une même donnée X dans une BD.
- Chacune
 1. Lit la valeur de X,
 2. Ajoute 10 à la valeur lue,
 3. Écrit la nouvelle valeur dans la BD
- Si chaque instance effectue ces opérations de manière atomique, pas de problèmes !

Isolation de transaction

- Ceci n'est pas *garanti* selon *le niveau d'isolation choisi*.
- Par exemple, si on ne prend pas de précaution
 1. A lit X, dans la BD on a X=0,
 2. B lit X, dans la BD on a X=0,
 3. A ajoute 10 à X et écrit le résultat dans la BD.
Dans la BD on a X=10,
 4. B ajoute 10 à sa copie de X et écrit le résultat dans la BD. Dans la BD on a X=10, ce qui est anormal !
- On a perdu le traitement effectué par A !
287

Isolation de transaction

- Solution : utiliser un verrou.
- Chaque composant
 1. Demande un verrou,
 2. Fait le travail précédent sur X,
 3. Libère le verrou
- Gestion de file d'attente... exclusion mutuelle, etc...
- Tout ceci peut-être réglé avec les EJBs

Niveau d'isolation avec les EJB

- 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 brouillée,
 2. Lecture ne pouvant être répétée,
 3. Lecture fantôme.

Niveau d'isolation avec les EJB

■ Lecture brouillée

- 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 avec les EJB

■ 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 avec les EJB

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...)

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
java.sql.Connection.SetTransactionIsolation(...).
- 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 d'isolation ???

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
 - 1. Stratégie pessimiste
 - Tout peut foirer, on prend donc un verrou lors des accès BD, on fait notre travail, puis on libère le verrou.
 - 2. Stratégie optimiste
 - Peu de chance que ça foire, espère que tout va bien se passe.
 - 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);  
}
```

- Uniquement pour les session beans stateful

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*

```
public class CountBean implements SessionBean, SessionSynchronization {  
    public int val;  
    public int oldVal;  
  
    public void ejbCreate(int val) {  
        this.val=val;  
        this.oldVal=val;  
    }  
  
    public void afterBegin() { oldVal = val; }  
    public void beforeCompletion() {}  
    public void afterCompletion(boolean b) { if (b == false) val = oldVal; }  
  
    public int count() { return ++val; }  
    public void ejbRemove() {}  
    public void ejbActivate() {}  
    public void ejbPassivate() {}  
    public void setSessionContext(SessionContext ctx) {}  
}
```

Relations avec les entity beans

On complique un peu l'étude des entity beans !

- Les entity beans représentant des données dans une BD, il est logique d'avoir envie de s'occuper de gérer des relations
- Exemples
 - Une commande et des lignes de commande
 - Une personne et une adresse
 - Un cours et les élèves qui suivent ce cours
 - Un livre et ses auteurs
- Nous allons voir comment spécifier ces relations dans notre modèle EJB

Concepts abordés

- Cardinalité (1-1, 1-n, n-n...),
- Direction des relations (bi-directionnelles, uni-directionnelles),
- Agrégation vs composition et destructions en cascade,
- Relations récursives, circulaires, agressive-load, lazy-load,
- Intégrité référentielle,
- Accéder aux relations depuis un code client, via des Collections,
- Comment gérer tout ça !

Direction des relations (*directionality*)

■ Unidirectionnelle

- On ne peut aller que du bean A vers le bean B

■ Bidirectionnelle

- On peut aller du bean A vers le bean B et inversement

Cardinalité

- La cardinalité indique combien d'instances vont intervenir de chaque côté d'une relation
- One-to-One (1:1)
 - Un employé a une adresse...
- One-to-Many (1:N)
 - Un PDG et ses employés...
- Many-to-Many (M:N)
 - Des étudiants suivent des cours...

Cardinalité



Relations 1:1

- Représentée typiquement par une clé étrangère dans une BD
- Ex : une commande et un colis

OrderPK	OrderName	Shipment ForeignPK
12345	Software Order	10101
ShipmentPK	City	ZipCode
10101	Austin	78727

Relations 1:1, le bean Order

```
    @Entity(name= "OrderUni")
public class Order implements Serializable {
    private int id;
    private String orderName;
    private Shipment shipment;

    public Order() {
        id = (int)System.nanoTime();
    }

    @Id
    public int getId() {
        return id;
    }

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

Relations 1:1, le bean Order

```
...
// other setters and getters go here
...

@OneToOne(cascade={CascadeType.PERSIST})
public Shipment getShipment() {
    return shipment;
}

public void setShipment(Shipment shipment) {
    this.shipment = shipment;
}
```

Relations 1:1, le bean Shipment

```
...
@Entity(name="ShipmentUni")
public class Shipment implements Serializable {
    private int id;
    private String city;
    private String zipcode;

    public Shipment() {

        id = (int)System.nanoTime();
    }

    @Id
    public int getId() {
        return id;
    }

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

    ...
    // other setters and getters go here
    ...
}
```

Exemple de code pour insérer une commande avec une livraison reliée

```
...
@Stateless
public class OrderShipmentUniBean implements OrderShipment {
    @PersistenceContext
    EntityManager em;

    public void doSomeStuff() {

        Shipment s = new Shipment();
        s.setCity("Austin");
        s.setZipcode("78727");

        Order o = new Order();
        o.setOrderName("Software Order");
        o.setShipment(s);

        em.persist(o);
    }

    public List getOrders() {
        Query q = em.createQuery("SELECT o FROM OrderUni o");
        return q.getResultList();
    }
}
```

Relations 1:1, exemple de client (ici un main...)

```
...
InitialContext ic = new InitialContext();
OrderShipment os =
    (OrderShipment) ic.lookup(OrderShipment.class.getName());

os.doSomeStuff();

System.out.println("Unidirectional One-To-One client\n");

for (Object o : os.getOrders()) {
    Order order = (Order)o;
    System.out.println("Order "+order.getId()+": "+
        order.getOrderName());
    System.out.println("\tShipment details: "+
        order.getShipment().getCity()+" "+
        order.getShipment().getZipcode());
}
...
```

Version bidirectionnelle (on modifie Shipment)

```
...
@Entity(name="ShipmentBid")
public class Shipment implements Serializable {
    private int id;
    private String city;
    private String zipcode;
    private Order order;

    public Shipment() {
        id = (int)System.nanoTime();
    }

    @Id
    public int getId() {
        return id;
    }

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

Version bidirectionnelle (suite)

```
...
// other setters and getters go here
...

@OneToOne(mappedBy="shipment")
public Order getOrder() {
    return order;
}

public void setOrder(Order order) {
    this.order = order;
}
}
```

Version bi-directionnelle (suite, code qui fait le persist)

- On peut maintenant ajouter au code de tout à l'heure (celui qui écrit une commande) :

```
...
public List getShipments() {
    Query q = em.createQuery("SELECT s FROM ShipmentBid s");
    return q.getResultList();
}
...
```

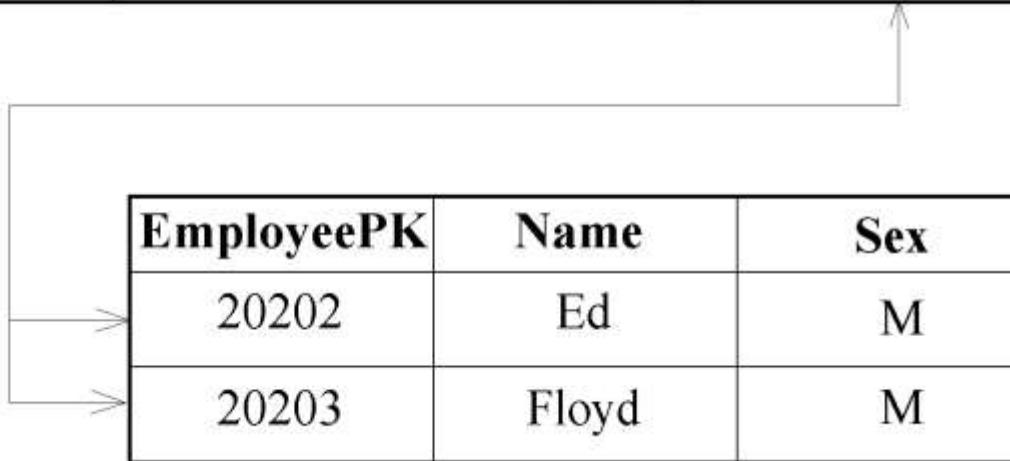
Version bi-directionnelle (suite, code du client)

```
...
for (Object o : os.getShipments()) {
    Shipment shipment = (Shipment)o;
    System.out.println("Shipment: "+
        shipment.getCity()+" "+
        shipment.getZipcode());
    System.out.println("\tOrder details: "+
        shipment.getOrder().getOrderName());
}
...
```

Relations 1:N

- Exemple : une entreprise a plusieurs employés

CompanyPK	Name	Employee FKs
12345	The Middleware Company	<Vector BLOB>



Relations 1:N

- Exemple : une entreprise a plusieurs employés
 - Solution plus propre (éviter les BLOBs!)

The diagram illustrates a 1:N relationship between two tables: Company and Employee. The Company table has one row with CompanyPK 12345 and Name 'The Middleware Company'. The Employee table has two rows: one with EmployeePK 20202, Name 'Ed', Sex 'M', and Company 12345; and another with EmployeePK 20203, Name 'Floyd', Sex 'M', and Company 12345. A curved arrow points from the CompanyPK column of the Company table to the CompanyPK column of the Employee table.

CompanyPK	Name
12345	The Middleware Company

EmployeePK	Name	Sex	Company
20202	Ed	M	12345
20203	Floyd	M	12345

Relations 1:N exemple

```
...
@Entity(name="CompanyOMUni")
public class Company implements Serializable {
    private int id;
    private String name;
    private Collection<Employee> employees;

    ...
    // other getters and setters go here
    // including the Id
    ...

    @OneToMany(cascade={CascadeType.ALL},fetch=FetchType.EAGER)
    public Collection<Employee> getEmployees() {
        return employees;
    }

    public void setEmployees(Collection<Employee> employees) {
        this.employees = employees;
    }
}
```

Relations 1:N exemple

```
...
@Entity(name="EmployeeOMUni")
public class Employee implements Serializable {
    private int id;
    private String name;
    private char sex;

    ...
    // other getters and setters go here
    // including the Id
    ...
}
```

Exemple de code qui insère des compagnies

```
...
@Stateless
public class CompanyEmployeeOMUniBean implements CompanyEmployeeOM {
    @PersistenceContext
    EntityManager em;

    public void doSomeStuff() {
        Company c = new Company();
        c.setName("M*Power Internet Services, Inc.");

        Collection<Employee> employees = new ArrayList<Employee>();
        Employee e = new Employee();
        e.setName("Micah Silverman");
        e.setSex('M');
        employees.add(e);

        e = new Employee();
        e.setName("Tes Silverman");
        e.setSex('F');
        employees.add(e);

        c.setEmployees(employees);
        em.persist(c);
```

Exemple de code qui liste des compagnies

```
public List getCompanies() {  
    Query q = em.createQuery("SELECT c FROM CompanyOMUni c");  
    return q.getResultList();  
}
```

Exemple de client

```
...
    InitialContext ic = new InitialContext();
    CompanyEmployeeOM ceom = (CompanyEmployeeOM)ic.lookup(
        CompanyEmployeeOM.class.getName());

ceom.doSomeStuff();

for (Object o : ceom.getCompanies()) {
    Company c = (Company)o;
    System.out.println("Here are the employees for company: "+
        c.getName());
    for (Employee e : c.getEmployees()) {
        System.out.println("\tName: "+
            e.getName(), Sex: "+e.getSex());
    }
    System.out.println();
}
...
...
```

Version bidirectionnelle

```
...
@Entity(name="CompanyOMBid")
public class Company implements Serializable {
    private int id;
    private String name;
    private Collection<Employee> employees;

    ...
    @OneToMany(cascade={CascadeType.ALL},
               fetch=FetchType.EAGER,
               mappedBy="company")
    public Collection<Employee> getEmployees() {
        return employees;
    }

    public void setEmployees(Collection<Employee> employees) {
        this.employees = employees;
    }
    ...
}
```

Version bidirectionnelle

```
...
@Entity(name="EmployeeOMBid")
public class Employee implements Serializable {
    private int id;
    private String name;
    private char sex;
private Company company;

...
@ManyToOne
public Company getCompany() {
    return company;
}

public void setCompany(Company company) {
    this.company = company;
}
}
```

Version bidirectionnelle

```
...
@Stateless
public class CompanyEmployeeOMBidBean implements CompanyEmployeeOM {
    @PersistenceContext
    EntityManager em;

    public void doSomeStuff() {
        Company c = new Company();
        c.setName("M*Power Internet Services, Inc.");

        Collection<Employee> employees = new ArrayList<Employee>();
        Employee e = new Employee();
        e.setName("Micah Silverman");
        e.setSex('M');
e.setCompany(c);

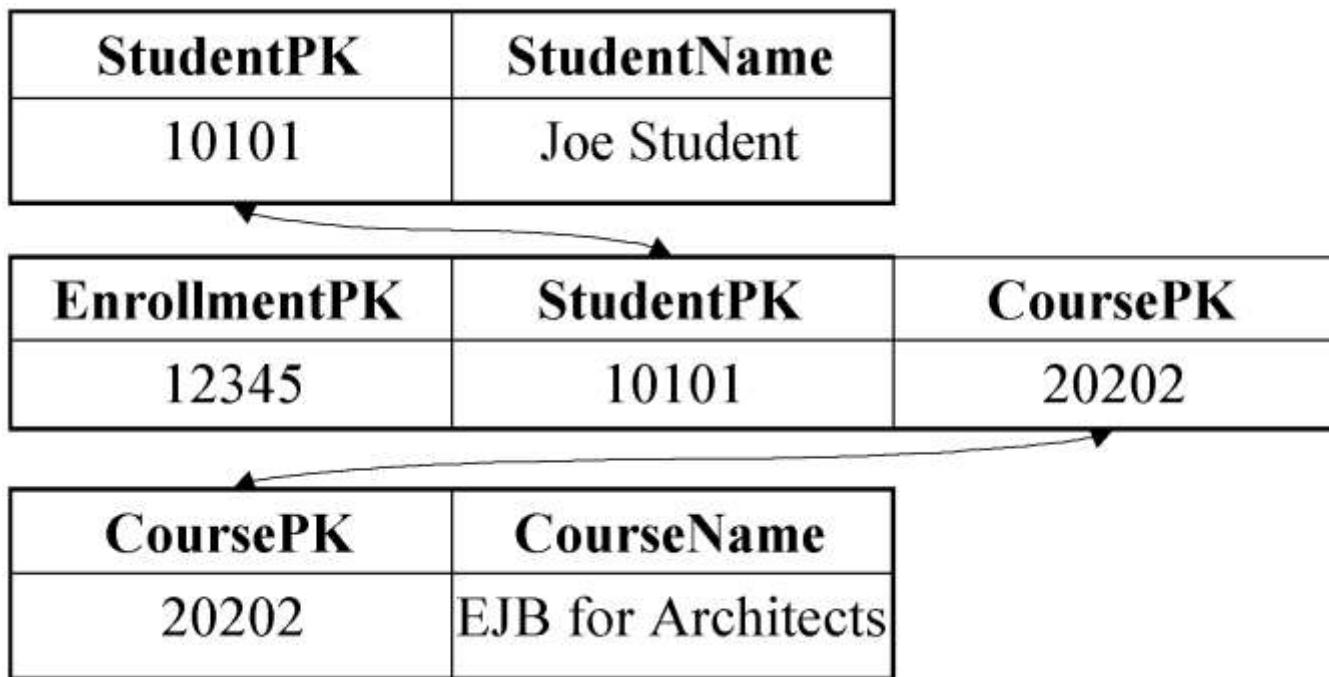
        employees.add(e);

        e = new Employee();
        e.setName("Tes Silverman");
        e.setSex('F');
e.setCompany(c);
        employees.add(e);

        c.setEmployees(employees);
        em.persist(c);
```

Relations M:N

- Un étudiant suit plusieurs cours, un cours a plusieurs étudiants inscrits
 - Table de jointure nécessaire.



Relations M:N, choix de conception

- Deux possibilités lorsqu'on modélise cette relation avec des EJBs
 1. Utiliser un troisième EJB pour modéliser la table de jointure. On veut peut-être mémoriser la date où un étudiant s'est inscrit, etc... Cet EJB possèdera deux relations 1:N vers le bean Student et le vers le bean Course
 2. Si on a rien besoin de plus à part la relation, on peut utiliser simplement deux EJB, chacun ayant un attribut correspondant à une Collection de l'autre EJB...

Relations M:N, exemple

```
...
@Entity(name="StudentUni")
public class Student implements Serializable {
    private int id;
    private String name;
    private Collection<Course> courses = new ArrayList<Course>();

    public Student() {
        id = (int) System.nanoTime();
    }

    @Id
    public int getId() {
        return id;
    }

    ...
    //other setters and getters go here
    ...

    @ManyToMany(cascade={CascadeType.ALL}, fetch=FetchType.EAGER)
    @JoinTable(name="STUDENTUNI_COURSEUNI")
    public Collection<Course> getCourses() {
        return courses;
    }

    public void setCourses(Collection<Course> courses) {
        this.courses = courses;
    }
}
```

Relations M:N, exemple

```
...
@Entity(name="CourseUni")
public class Course implements Serializable {
    private int id;
    private String courseName;
    private Collection<Student> students = new ArrayList<Student>();

    ...
    //setters and getters go here
    ...
}
```

Relations M:N, exemple

```
...
@Stateless
public class StudentCourseUniBean implements StudentCourse {
    @PersistenceContext
    EntityManager em;

    public void doSomeStuff() {
        Course c1 = new Course();
        c1.setCourseName("EJB 3.0 101");

        Course c2 = new Course();
        c2.setCourseName("EJB 3.0 202");

        Student s1 = new Student();
        s1.setName("Micah");

s1.getcourses().add(c1);

c1.getStudents().add(s1);

        Student s2 = new Student();
        s2.setName("Tes");

s2.getcourses().add(c1);
s2.getcourses().add(c2);

c1.getStudents().add(s2);
```

Relations M:N, exemple

```
Student s2 = new Student();
s2.setName("Tes");

s2.getcourses().add(c1);
s2.getcourses().add(c2);

c1.getStudents().add(s2);
c2.getStudents().add(s2);

em.persist(s1);
em.persist(s2);
}

public List<Student> getAllStudents() {
    Query q = em.createQuery("SELECT s FROM StudentUni s");
    return q.getResultList();
}
}
```

Relations M:N, exemple

```
...
    InitialContext ic = new InitialContext();
    StudentCourse sc = (StudentCourse)ic.lookup(
        StudentCourse.class.getName());

    sc.doSomeStuff();

    for (Student s : sc.getAllStudents()) {
        System.out.println("Student: "+s.getName());
        for (Course c : s.getcourses()) {
            System.out.println("\tCourse: "+c.getCourseName());
        }
    }
...
}
```

La directionalité et le modèle de données dans la DB

- La directionalité peut ne pas correspondre à celle du modèle de données dans la DB

Schéma normalisé

PersonPK	PersonName	Address ForeignPK
12345	Ed Roman	10101

AddressPK	City	ZipCode
10101	Austin	78727

Schéma dénormalisé

PersonPK	PersonName	Address ForeignPK
12345	Ed Roman	10101

AddressPK	City	ZipCode	Person ForeignPK
10101	Austin	78727	12345

Choisir la directionnalité ?

- Premier critère : la logique de votre application,
- Second critère : si le schéma relationnel existe, s'adapter au mieux pour éviter de mauvaises performances.

Lazy-loading des relations

■ Agressive-loading

- Lorsqu'on charge un bean, on charge aussi tous les beans avec lesquels il a une relation.
- Cas de la Commande et des Colis plus tôt dans ce chapitre.
- Dans le `ejbLoad()` on appelle des *finders*...
- Peut provoquer un énorme processus de chargement si le graphe de relations est grand.

■ Lazy-loading

- On ne charge les beans en relation que lorsqu'on essaie d'accéder à l'attribut qui illustre la relation.
- Tant qu'on ne demande pas quels cours il suit, le bean Etudiant n'appelle pas de méthode *finder* sur le bean Cours.

Agrégation vs Composition et destructions en cascade

■ Relation par Agrégation

- Le bean *utilise* un autre bean
- Conséquence : si le bean A *utilise* le bean B, lorsqu'on détruit A on ne détruit pas B.
- Par exemple, lorsqu'on supprime un étudiant on ne supprime pas les cours qu'il suit. Et vice-versa.

■ Relation par Composition

- Le bean *se compose d'un* autre bean.
- Par exemple, une commande se compose de lignes de commande...
- Si on détruit la commande on détruit aussi les lignes correspondantes.
- Ce type de relation implique des destructions en cascade..

Relations et EJB-QL

- Lorsqu'on définit une relation en CMP, on peut aussi indiquer la requête qui permet de remplir le champs associé à la relation.
- On fait ceci à l'aide d'EJB-QL

SELECT o.customer

Renvoie tous les clients qui ont placé une commande

FROM Order o

- Principale différence avec SQL, l'opérateur `".."`

- Pas besoin de connaître le nom des tables, ni le nom des colonnes...

Relations et EJB-QL

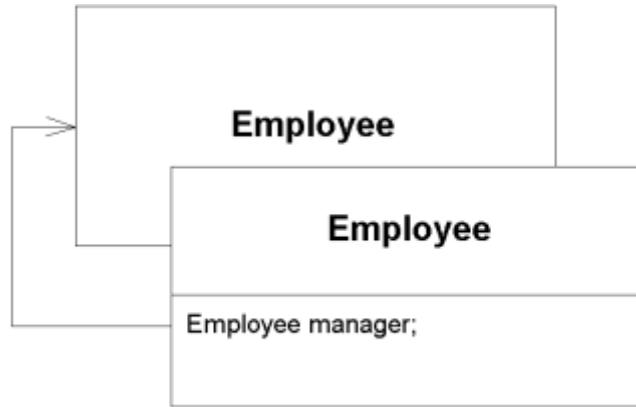
- On peut aller plus loin...

```
SELECT o.customer.address.homePhoneNumber  
FROM Order o
```

- On se promène le long des relations...

Relations récursives

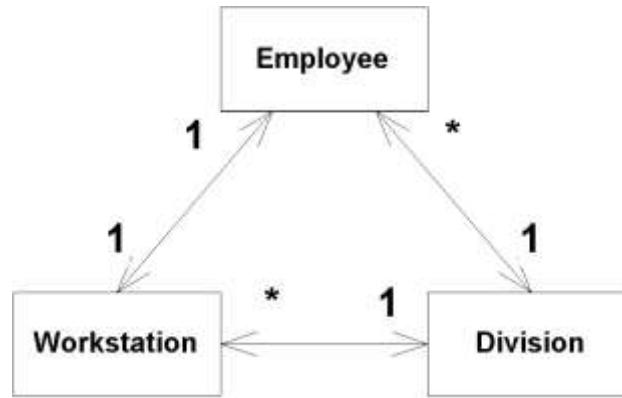
- Relation vers un bean de la même classe
 - Exemple : Employé/Manager



- Rien de particulier, ces relations sont implémentées exactement comme les relations non récursives...

Relations circulaires

- Similaire aux relations récursives sauf qu'elles impliquent plusieurs types de beans
 - Ex : un employé travaille dans une division, une division possède plusieurs ordinateurs (workstation), une workstation est allouée à un employé...



- Ce type de relation, en cas de aggressive-loading peut mener à une boucle sans fin...
 - Même problème avec les destructions en cascade... 342

Relations circulaires

■ Plusieurs stratégies sont possibles

1. Certains containers proposent d'optimiser le chargement d'un bean en chargeant toutes ses relations en cascade dans le ejbLoad(). Attention si relations circulaires !
2. Supprimer une des relations (!!!) si le modèle de conception le permet.
3. Supprimer la bidirectionnalité d'une des relations pour briser le cercle, si le modèle de conception le permet.
4. Utiliser le lazy-loading et ne pas faire de destruction en cascade.
5. Les meilleurs moteurs CMP détectent les relations circulaires et vous permettent de traiter le problème avant le runtime.

Intégrité référentielle

- Un bean Compagnie, Division, etc... a des relations avec un bean Employé
 - Si on supprime un employé, il faut vérifier qu'il est bien supprimé partout où on a une relation avec lui.
- Problème classique dans les SGBDs
 - Résolus à l'aide de *triggers*. Ils se déclenchent sitôt qu'une perte d'intégrité risque d'arriver et effectuent les opérations nécessaires.
 - On peut aussi utiliser des procédures stockées via JDBC. Ces procédures effectuent la vérification d'intégrité.

Intégrité référentielle

- Gérer l'intégrité dans le SGBD est intéressant si la BD est attaquée par d'autres applications que les EJBs...
- Autre approche : gérer l'intégrité dans les EJBs
 - Solution plus propre,
 - Le SGBD n'est plus aussi sollicité,
 - Avec les EJB: le travail est fait tout seul !

Intégrité référentielle

- Et dans un contexte distribué ?
- Plusieurs serveurs d'application avec le même composant peuvent accéder à des données sur le même SGBD,
- Comment mettre à jour les relations ?
- Problème résolu par les transactions !!!

Concepts avancés sur la persistence

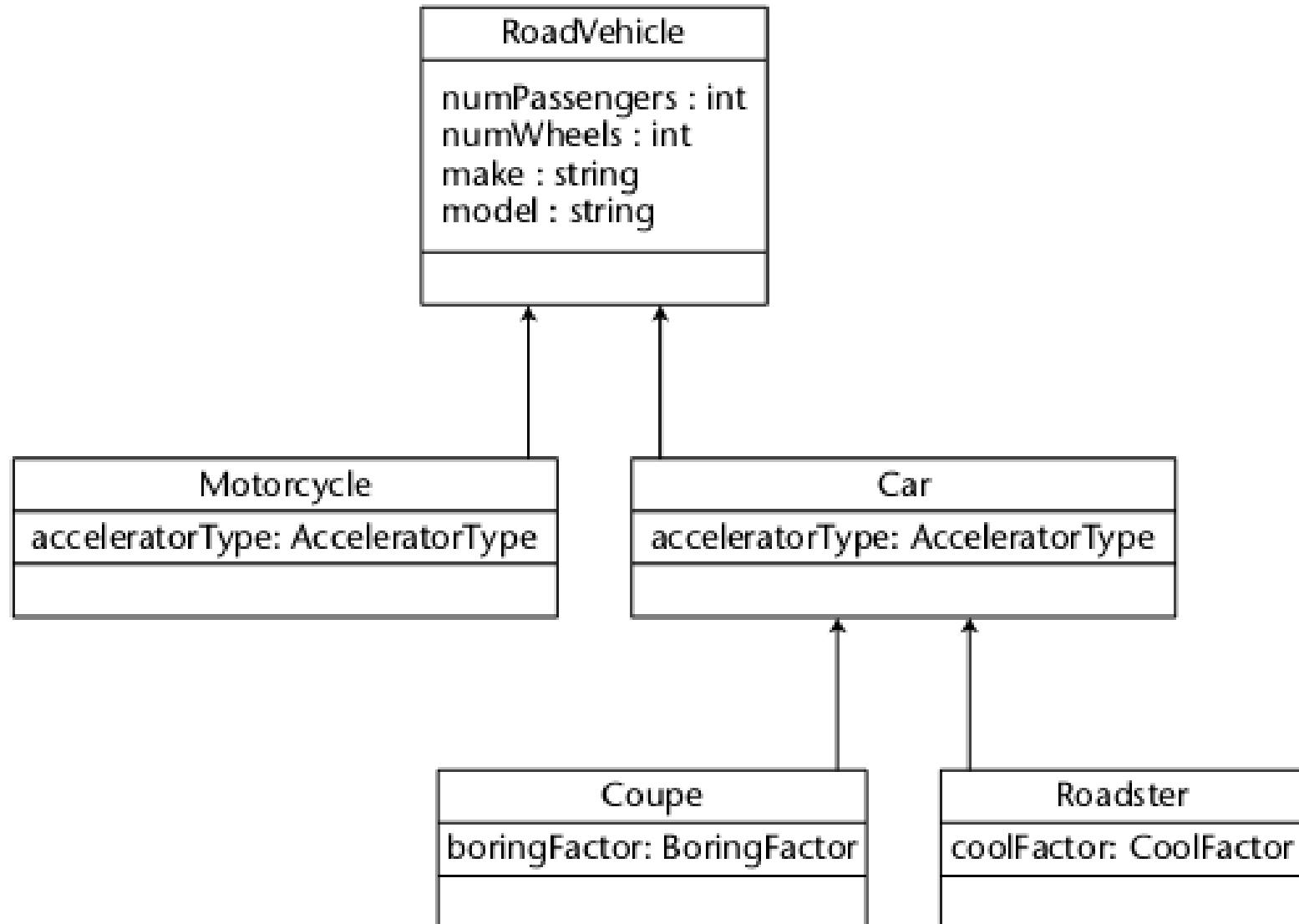
Introduction

- Et le polymorphisme ?
- Et l'héritage ?
- Et EJB-QL ?

Héritage

- Stratégies de mapping entre classes et tables quand on a de l'héritage ?
 - Une table pour toute la hiérarchie de classes ?
 - Une table par classe/sous-classe ?
 - Autres solutions ?

Un exemple !



Code de RoadVehicle.java (classe racine)

```
package examples.entity.single_table;

public class RoadVehicle {
    public enum AcceleratorType {PEDAL, THROTTLE};

    protected int numPassengers;
    protected int numWheels;
    protected String make;
    protected String model;

    // setters and getters go here
    ...

    public String toString() {
        return "Make: "+make+
            ", Model: "+model+
            ", Number of passengers: "+numPassengers;
    }
}
```

Code de Motorcycle.java

```
package examples.entity.single_table;

public class Motorcycle extends RoadVehicle {
    public final AcceleratorType acceleratorType =
        AcceleratorType.THROTTLE;

    public Motorcycle() {
        numWheels = 2;
        numPassengers = 2;
    }

    public String toString() {
        return "Motorcycle: "+super.toString();
    }
}
```

Code de Car.java

```
package examples.entity.single_table;

public class Car extends RoadVehicle {
    public final AcceleratorType acceleratorType =
        AcceleratorType.PEDAL;

    public Car() {
        numWheels = 4;
    }

    public String toString() {
        return "Car: "+super.toString();
    }
}
```

Code de Roadster.java

```
package examples.entity.single_table;

public class Roadster extends Car {
    public enum CoolFactor {COOL, COOLER, COOLEST};

    private CoolFactor coolFactor;

    public Roadster() {
        numPassengers = 2;

    }

    // setters and getters go here
    ...

    public String toString() {
        return "Roadster: "+super.toString();
    }
}
```

Code de Coupe.java

```
package examples.entity.single_table;

public class Coupe extends Car {
    public enum BoringFactor {BORING,BORINGER,BORINGEST};

    private BoringFactor boringFactor;

    public Coupe() {
        numPassengers = 5;
    }

    // setters and getters go here
    ...

    public String toString() {
        return "Coupe: "+super.toString();
    }
}
```

Premier cas : une seule table !

- Une seule table représente toute la hiérarchie.
- Une colonne de « discrimination » est utilisée pour distinguer les sous-classes.
- Cette solution supporte le polymorphisme.
- Désavantages :
 - Une colonne pour chaque champ de chaque classe,
 - Comme une ligne peut être une instance de chaque classe, des champs risquent de ne servir à rien (nullable)

Regardons le code avec les annotations !

```
package examples.entity.single_table;

// imports go here

@Entity(name="RoadVehicleSingle")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="DISC",
    discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("ROADVEHICLE")
public class RoadVehicle implements Serializable {
    public enum AcceleratorType {PEDAL, THROTTLE};
```

(suite)

```
@Id  
protected int id;  
protected int numPassengers;  
protected int numWheels;  
protected String make;  
protected String model;  
  
public RoadVehicle() {  
    id = (int) System.nanoTime();  
}  
  
// setters and getters go here  
...  
}
```

Motorcycle.java annoté !

```
package examples.entity.single_table;

// imports go here

@Entity
@DiscriminatorValue("MOTORCYCLE")
public class Motorcycle extends RoadVehicle implements Serializable {
    ...
    AcceleratorType acceleratorType = AcceleratorType.THROTTLE;

    public Motorcycle() {
        super();
        numWheels = 2;
        numPassengers = 2;
    }
}
```

Car.java annoté

```
package examples.entity.single_table;

// imports go here

@Entity
@DiscriminatorValue("CAR")
public class Car extends RoadVehicle implements Serializable {
    public final AcceleratorType acceleratorType =
        AcceleratorType.PEDAL;

    public Car() {
        super();
        numWheels = 4;
    }
}
```

Roadster.java annoté

```
package examples.entity.single_table;

// imports go here

@Entity
@DiscriminatorValue("ROADSTER")
public class Roadster extends Car implements Serializable {
    public enum CoolFactor {COOL, COOLER, COOLEST};

    private CoolFactor coolFactor;

    public Roadster() {
        super();
        numPassengers = 2;
    }

    }

    // setters and getters go here
    ...
}
```

Coupe.java annoté

```
package examples.entity.single_table;

// imports go here

@Entity
@DiscriminatorValue("COUPE")
public class Coupe extends Car implements Serializable {
    public enum BoringFactor {BORING, BORINGER, BORINGEST};

    private BoringFactor boringFactor;

    public Coupe() {
        super();
        numPassengers = 5;
    }

    // setters and getters go here
    ...
}
```

Table correspondante

```
CREATE TABLE ROADVEHICLE (
    ID INTEGER NOT NULL,
    DISC VARCHAR(31),
    NUMWHEELS INTEGER,
    MAKE VARCHAR(255),
    NUMPASSENGERS INTEGER,
    MODEL VARCHAR(255),
    ACCELERATORTYPE INTEGER,
    COOLFACTOR INTEGER,
    BORINGFACTOR INTEGER
);
```

Quelques objets persistants !

```
...
@PersistenceContext
EntityManager em;
...

    Coupe c = new Coupe();
    c.setMake("Bob");
    c.setModel("E400");
    c.setBoringFactor(BoringFactor.BORING);
em.persist(c);

    Roadster r = new Roadster();
    r.setMake("Mini");
    r.setModel("Cooper S");
    r.setCoolFactor(CoolFactor.COOLEST);
em.persist(r);

    Motorcycle m = new Motorcycle();
em.persist(m);
...
```

Et les données correspondantes

ID	DISC	MAKE	MODEL	COOL FACTOR	BORINGFACTOR
1818876882	COUPE	Bob	E400	NULL	0
1673414469	MOTORCYCLE	NULL	NULL	2	NULL
1673657791	ROADSTER	Mini	Cooper S	NULL	NULL

Deuxième stratégie : une table par classe

- Il suffit de modifier quelques annotations !
 - Dans RoadVehicle.java

```
@Entity(name="RoadVehicleJoined")
@Table(name="ROADVEHICLEJOINED")
@Inheritance(strategy=InheritanceType.JOINED)
public class RoadVehicle {
    ...
}
```

- Il faut retirer les @Discriminator des sous-classes,
- Le champ Id de la classe RoadVehicle sera une clé étrangère dans les tables des sous-classes,
- Remarque : on utilise ici @TABLE pour ne pas que la table porte le même nom que dans l'exemple précédent (facultatif)

Les tables !

Table 9.2 ROADVEHICLEJOINED Table

ID	DTYPE	NUMWHEELS	MAKE	MODEL
1423656697	Coupe	4	Bob	E400
1425368051	Motorcycle	2	NULL	NULL
1424968207	Roadster	4	Mini	Cooper S

Table 9.3 MOTORCYCLE Table

ID	ACCELERATORTYPE
1425368051	1

Les tables (suite)

Table 9.4 CAR Table

ID	ACCELERATORTYPE
1423656697	0
1424968207	0

Table 9.5 COUPE Table

ID	BORINGFACTOR
1423656697	0

Table 9.6 ROADSTER Table

ID	COOLFACTOR
1423656697	2

Requête SQL pour avoir tous les Roadsters

- Il faut faire des joins !
- Plus la hierarchie est profonde, plus il y aura de jointures : problèmes de performance !

```
SELECT
    rvj.NumWheels, rvj.Make, rvj.Model,
    c.AcceleratorType, r.CoolFactor
FROM
    ROADVEHICLEJOINED rvj, CAR c, ROADSTER r
WHERE
    rvj.Id = c.Id and c.Id = r.Id;
```

Conclusion sur cette approche

- Supporte le polymorphisme,
- On alloue juste ce qu'il faut sur disque,
- Excellente approche si on a pas une hiérarchie trop profonde,
- A éviter sinon...

Autres approches

- Des classes qui sont des entity bean peuvent hériter de classes qui n'en sont pas,
- Des classes qui ne sont pas des entity beans peuvent hériter de classes qui en sont,
- Des classes abstraites peuvent être des entity beans,
- (déjà vu : une classe qui est un entity bean hérite d'une autre classe qui est un entity bean)

Cas 1 : Entity Bean étends classe java

- On utilise l'attribut `@mappedsuperclass` dans la classe mère
 - Indique qu'aucune table ne lui sera associée

```
...
@MappedSuperclass
public class RoadVehicle {
    public enum AcceleratorType {PEDAL, THROTTLE};

    @Id
    protected int id;
    protected int numPassengers;
    protected int numWheels;

    protected String make;
    protected String model;
    ...
}

...
```

Cas 1 (les sous-classes entities)

```
@Entity
public class Motorcycle extends RoadVehicle {
    public final AcceleratorType ac = AcceleratorType.THROTTLE ;
    ...
}

...
...


@Entity
public class Car extends RoadVehicle {
    public final AcceleratorType ac = AcceleratorType.PEDAL;
    ...
}
```

Cas 1 : les tables

Table 9.7 Database Table Layout Mapped for Roadster.java

ID	NUMPASSENGERS	NUMWHEELS	MAKE	MODEL	ACCELERATORTYPE	COOLFACTOR
----	---------------	-----------	------	-------	-----------------	------------

Table 9.8 Database Table Layout Mapped for Coupe.java

ID	NUMPASSENGERS	NUMWHEELS	MAKE	MODEL	ACCELERATORTYPE	BORINGFACTOR
----	---------------	-----------	------	-------	-----------------	--------------

Table 9.9 Database Table Layout Mapped for Motorcycle.java

ID	NUMPASSENGERS	NUMWHEELS	MAKE	MODEL	ACCELERATORTYPE
----	---------------	-----------	------	-------	-----------------

Remarques sur le cas 1

- RoadVehicle n'aura jamais sa propre table,
- Les sous-classes auront leur propre table, avec comme colonnes les attributs de RoadVehicle en plus des leurs,
- Si on avait pas mis @MappedSuperclass dans RoadVehicle.java, les attributs hérités n'auraient pas été des colonnes dans les tables des sous-classes.

Classe abstraite et entity bean

- Une classe abstraite peut être un entity bean (avec @entity)
- Elle ne peut pas être instanciée, ses sous-classes concrètes oui,
- Elle aura une table dédiée,
- Elle pourra faire l'objet de requêtes (polymorphisme) : très intéressant !

Polymorphisme ! Exemple avec un SessionBean

```
...
@Stateless
public class RoadVehicleStatelessBean implements RoadVehicleStateless {
    @PersistenceContext(unitName="pu1")
    EntityManager em;

    public void doSomeStuff() {
        Coupe c = new Coupe();
        c.setMake("Bob");
        c.setModel("E400");
        c.setBoringFactor(BoringFactor.BORING);
        em.persist(c);

        Roadster r = new Roadster();
        r.setMake("Mini");
        r.setModel("Cooper S");
        r.setCoolFactor(CoolFactor.COOLEST);
        em.persist(r);

        Motorcycle m = new Motorcycle();
        em.persist(m);
    }
}
```

Polymorphisme (suite)

■ Des requêtes polymorphes ! Si ! Si !

```
public List getAllRoadVehicles() {  
    Query q = em.createQuery(  
        "SELECT r FROM RoadVehicleSingle r");  
    return q.getResultList();  
}
```

```
...  
}

```
...
}
```


```

Polymorphisme : code client

```
...
public class RoadVehicleClient {
    public static void main(String[] args) {
        InitialContext ic;
        try {
            ic = new InitialContext();
            String name =
                RoadVehicleStateless.class.getName();
            RoadVehicleStateless rvs =
                (RoadVehicleStateless)ic.lookup(name);

            rvs.doSomeStuff();

            for (Object o : rvs.getAllRoadVehicles()) {
                System.out.println("RoadVehicle: "+o);
            }
        } catch (NamingException e) {
            e.printStackTrace();
        }
    }
}
```

Polymorphisme : oui, ça marche !

- C'est bien la méthode `toString()` de chaque sous-classe qui est appelée !
- La requête à récupéré tous les `RoadVehicle` (s)

```
RoadVehicle: Coupe: Car:  
    Make: Bob, Model: E400, Number of passengers: 5  
RoadVehicle: Motorcycle:  
    Make: null, Model: null, Number of passengers: 2  
RoadVehicle: Roadster: Car:  
    Make: Mini, Model: Cooper S, Number of passengers: 2
```

EJB QL : Quelques exemples

- Voir le fichier PDF fourni avec les TPs !

```
// customers 20-30 named 'Joe', ordered by last name
Query q = em.createQuery("select c from Customer c where
    c.firstName = :fname order by c.lastName") ;
q.setParameter("fname", "Joe") ;
q.setFirstResult(20) ;
q.setMaxResults(10) ;
List<Customer> customers = (List<Customer>) q.getResultList() ;
```

EJB QL : Quelques exemples (suite)

```
// all orders, as a named query

@Entity
@NamedQuery(name="Order:findAllOrders", query="select o from Order o");
public class Order { ... }

Query q = em.createNamedQuery("Order:findAllOrders");
```

EJB QL : Quelques exemples (suite)

```
// all people, via a custom SQL statement
Query q = em.createNativeQuery("SELECT ID, VERSION, SUBCLASS,
    FIRSTNAME, LASTNAME FROM PERSON", Person.class);
List<Person> people = (List<Person>) q.getResultList();

// single-result aggregate: average order total price
Query q = em.createQuery("select avg(i.price) from Item i");
Number avgPrice = (Number) q.getSingleResult();
```

EJB QL : Quelques exemples (suite)

- Liste toutes les commandes qui ne comprennent pas (LEFT) de produit dont le prix est supérieur à une certaine quantité

```
// traverse to-many relations  
  
Query q = em.createQuery("select o from Order o  
    left join o.items li where li.price > :price");  
  
q.setParameter("price", 1000);  
  
List<Order> orders = (List<Order>) q.getResultList();
```

■ Table des compagnies

ID	NAME
1	M*Power Internet Service, Inc.
2	Sun Microsystems
3	Bob's Bait and Tackle

Table D.1 Company

■ Table des employés

ID	NAME	COMPANY_ID
1	Micah Silverman	1
2	Tes Silverman	1
3	Rima Patel	2

EJB QL : Quelques exemples (suite)

- Cette requête récupère trois compagnies :

```
SELECT DISTINCT c FROM CompanyOMBid c
```

- Mais celle-ci uniquement deux :

```
SELECT DISTINCT c FROM CompanyOMBid c JOIN c.employees
```

- Celle-là : les trois (même si join condition absente)

```
SELECT DISTINCT c FROM CompanyOMBid c LEFT JOIN c.employees
```

EJB QL : Quelques exemples (suite)

- Provoque le chargement des entities reliées

```
SELECT DISTINCT c FROM CompanyOMBid c JOIN FETCH c.employees
```

- Prend le devant sur @FetchType.LAZY
- Autre exemple :

```
SELECT DISTINCT c
FROM CompanyOMBid c, IN(c.employees) e
WHERE e.name='Micah Silverman'
```

EJB QL : Quelques exemples (suite)

■ WHERE et requêtes paramétrées

```
Query q =  
    em.createQuery("SELECT c FROM CompanyOMBid c WHERE c.name = ?1") .  
        setParameter(1, "M*Power Internet Services, Inc.");
```

■ Autre exemple avec paramètres nommés

```
Query q =  
    em.createQuery("SELECT c FROM CompanyOMBid c WHERE c.name = :cname") .  
        setParameter("cname", "M*Power Internet Services, Inc.");
```

EJB QL : Quelques exemples (suite)

■ Expressions

```
SELECT r FROM RoadVehicleSingle r WHERE r.numPassengers between 4 AND 5
```

```
SELECT r FROM RoadVehicleSingle r WHERE r.numPassengers IN(2,5)
```

```
SELECT r FROM RoadVehicleSingle r WHERE r.make LIKE 'M%'
```

■ Le % dans le LIKE = suite de caractères, le _ = un caractère

```
SELECT r FROM RoadVehicleSingle r WHERE r.model IS NOT NULL
```

```
SELECT c FROM CompanyOMBid c WHERE c.employees IS NOT EMPTY
```

EJB QL : Quelques exemples (suite)

■ MEMBER OF

```
"SELECT e FROM EmployeeOMBid e, CompanyOMBid c  
WHERE e MEMBER OF c.employees"
```

■ Sous-Requêtes

```
SELECT c FROM CompanyOMBid c WHERE  
(SELECT COUNT(e) FROM c.employees e) = 0
```

Fonctions sur chaînes, arithmétique

```
functions returning strings ::=  
    CONCAT(string primary, string primary) |  
    SUBSTRING(string_primary,  
                simple arithmetic expression,  
                simple arithmetic expression) |  
    TRIM([[trim_specification] [trim_character] FROM  
            string primary) |  
    LOWER(string primary) |  
    UPPER(string_primary)  
trim specification ::= LEADING | TRAILING | BOTH  
  
functions_returning_numerics ::=  
    LENGTH(string primary) |  
    LOCATE(string_primary, string_primary[,  
            simple_arithmetic_expression])
```

Fonctions sur chaînes, arithmétique (suite)

```
functions returning numerics::=
    ABS(simple_arithmetic_expression) |
    SQRT(simple arithmetic expression) |
    MOD(simple_arithmetic_expression, simple_arithmetic_expression) |
    SIZE(collection_valued_path_expression)
```

EJB QL : Quelques exemples (suite)

```
// bulk update: give everyone a 10% raise
Query q = em.createQuery("update Employee emp
    set emp.salary = emp.salary * 1.10");
int updateCount = q.executeUpdate();

// bulk delete: get rid of fulfilled orders
Query q = em.createQuery("delete from Order o
    where o.fulfilledDate is not null");
int deleteCount = q.executeUpdate();
```

EJB QL : Quelques exemples (suite)

```
// subselects: all orders with an expensive line item  
  
Query q = em.createQuery("select o from Order o where exists  
(select li from o.items li where li.price > 10000)");
```

Bonnes pratiques de persistance

On manipule des données persistantes

- Application moderne = manipuler des données, les lire, les écrire, les modifier.
- Le plus souvent, ces données sont dans un SGBD.
- Lorsqu'on développe une application basée sur les EJB, la gestion de la persistance peut être aussi simple que *mapper* un EJB sur une table,
- Mais aussi devenir effroyablement complexe !

Dans ce chapitre...

- Quand utiliser des entity beans,
- Choisir entre BMP et CMP,
- Quelques design patterns...

Quand utilise des entity beans

- Utiliser simplement des sessions beans + JDBC ? Pourquoi pas ?
 - Contrôle total via JDBC,
 - Approche .NET
- Utiliser des entity beans BMP ou CMP
 - Le container fait une partie du travail (BMP) ou la totalité (CMP),
 - Il appelle `ejbLoad()`, `ejbStore()` dans votre dos...
 - Perte de contrôle parfois déroutante : éduquer les développeurs !

Analogie avec le passage de paramètres

- Lorsqu'on appelle un session bean, les résultats sont passés par valeur,
- Lorsqu'on appelle un entity bean, on récupère des *stubs* sur les objets résultats.
 - Similaire à un passage par référence.
- La plupart des temps les session beans sont clients des entity beans, et y accèdent via leur interface locale.
- Les clients des sessions peuvent être des servlets ou une application classique.

Analogie avec le passage de paramètres

- Réfléchissez...
- Est-ce plus avantageux pour un client normal de recevoir des résultats distants par référence
 - NON ! Il vaut mieux les recevoir par valeur, sérialisés.
- Est-ce plus avantageux pour un session bean de recevoir des résultats par référence ?
 - OUI, il tourne dans le même container que le bean serveur !
- CONCLUSION : utiliser des sessions beans comme front-end pour les clients finaux !
 - Sérialisent les résultats finaux vers le client.

Cache

- Les sessions beans ne représentent pas des données, et ne peuvent donc pas être cachées,
- Les données manipulées ne durent que le temps de la session.
- D'un autre côté, les entity beans peuvent être cachés à travers plusieurs transactions
 - Dans le descripteur de déploiement spécifique...
 - Par exemple, on cachera les 1000 livres les plus demandés chez Amazon.com !

Cache

- Mais inutile de cacher des données peu partagées
 - Données personnelles...
- Il est très intéressant d'utiliser des entity beans car souvent les données sont partagées et sont plus souvent lues que modifiées.

Un bon schéma relationnel

- Parfois changer deux colonnes dans une table peut prendre deux mois pour trois développeurs !
 - Véridique ! A cause de code SQL incompréhensible dans tous les coins !
- Utiliser une couche objet (entity beans) permet d'abstraire SQL et donc de faciliter la maintenance
- Pas facile à faire avec des sessions beans

Choix entre BMP et CMP

■ Facilité de programmation : CMP

- Réduction du code à écrire,
- Énormes descripteurs, mais le plus souvent ils sont générés par des wizards,
- RAD,
- Super pour prototyper, si on veut on peut toujours passer en BMP par la suite...

■ Performances : CMP

- Bien réglés, les beans CMP sont bien plus performants que les BMP,
- Le container peut résoudre les relations en un seul, gros ordre SQL,
- On peut néanmoins optimiser les BMP avec la pattern Fat Key

Choix entre BMP et CMP

■ Debugging : CMP

- Parfois, les bugs dans les CMPs sont durs à trouver car tout se passe en coulisse...
- Un descripteur mal spécifié...
- Avec CMP on contrôle tout...

■ Contrôle : BMP

- Avec BMP on contrôle tout,
- Mais les meilleurs containers ont des moteurs CMP très puissants et paramétrables...

■ Portabilité : CMP

- Indépendant de la base de données.

Choix entre BMP et CMP

■ Gestion des relations : CMP

- Depuis EJB 2.0, nombreuses améliorations.
- Gestion automatique de l'intégrité, langage EJB-QL, etc...
- Containers très performants aujourd'hui.
- Avec BMP : beaucoup de code à écrire, projet plus long à développer, l'argent rentre moins vite !!!

■ Courbe d'apprentissage : BMP mais le plus souvent par simple peur de ne pas se lancer dans CMP

Choix de la granularité

- Doit-on faire de gros ou de petits composants?
- Depuis EJB 2.0 et les interfaces locales, il n'est plus nécessaires de concentrer les traitements pour minimiser les échanges entre beans.
 - On peut faire de "petits" beans !
- Exemple : commande contient des lignes de commandes.
 - On n'hésite plus, chaque ligne est un entity bean !
 - Auparavant, on pouvait utiliser des classes normales sérialisables... Mais on perd tout l'avantage des relations en EJB 2.0 !

Trucs et astuces sur la persistance

- Parfois, pour des traitements nécessitant l'examen d'un grand nombre de données, il est préférable d'utiliser des procédures stockées.
 - Exemple : trouver les comptes bancaires débiteurs...
- Éviter de partager une source de données entre une application à base d'EJB et une application externe.
 - Sinon, là encore, les procédures stockées peuvent centraliser une partie de la logique...

Trucs et astuces sur la persistance

- Attention, il y a aussi de nombreuses raisons pour NE PAS utiliser des procédures stockées
 - Tout repose sur le SGBD, goulot d'étranglement
 - Peu portable,
 - Mauvaise flexibilité, migration difficile,
 - Maintenance ?

Trucs et astuces sur la persistance

- Si on peut, adapter le modèle de données au modèle EJB.
 - Évident si la source de données n'existe pas au début du projet.
- N'oubliez pas que les données sont les données, les objets sont les données + le comportement !