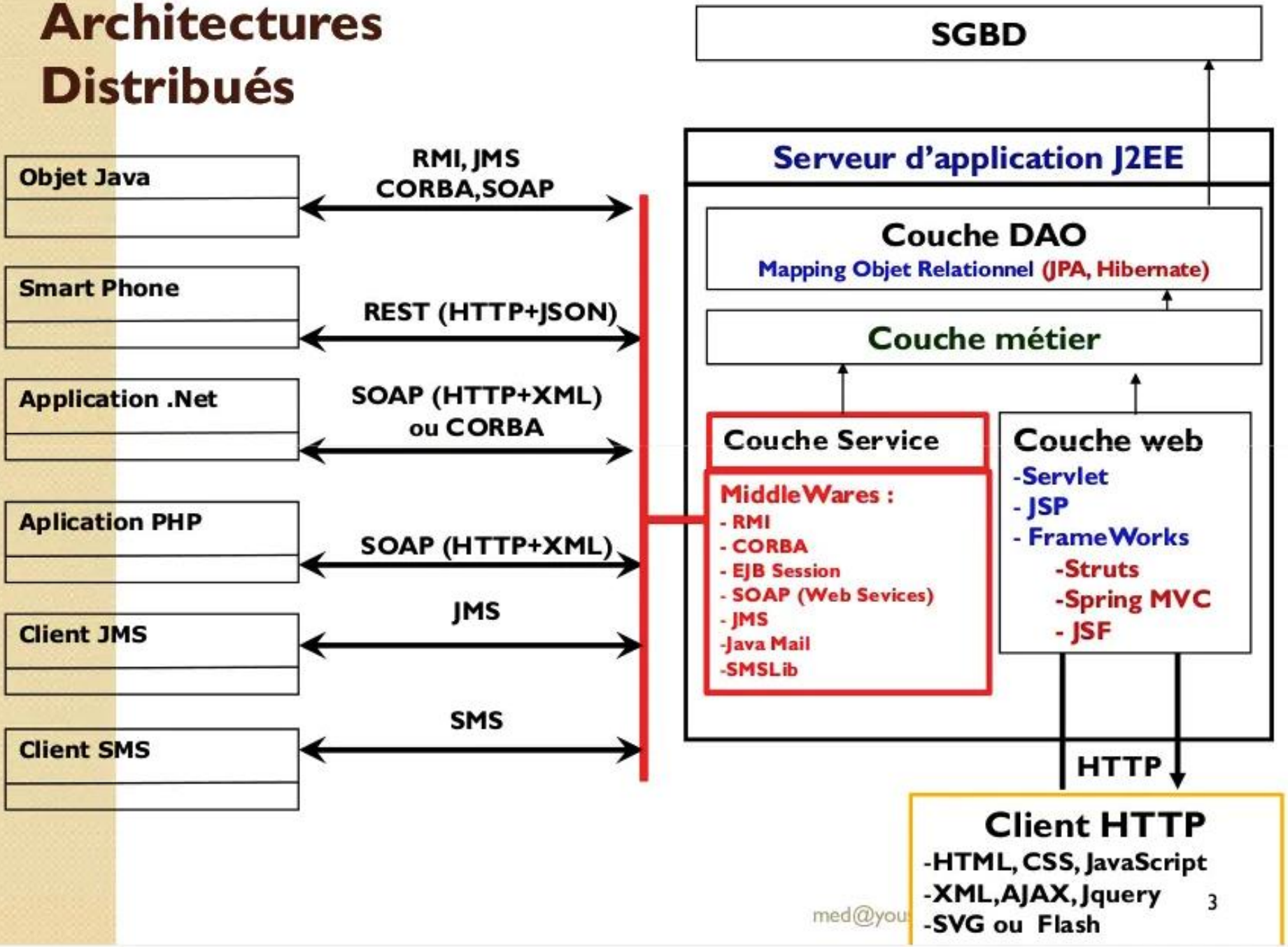
Enterprise JavaBeans

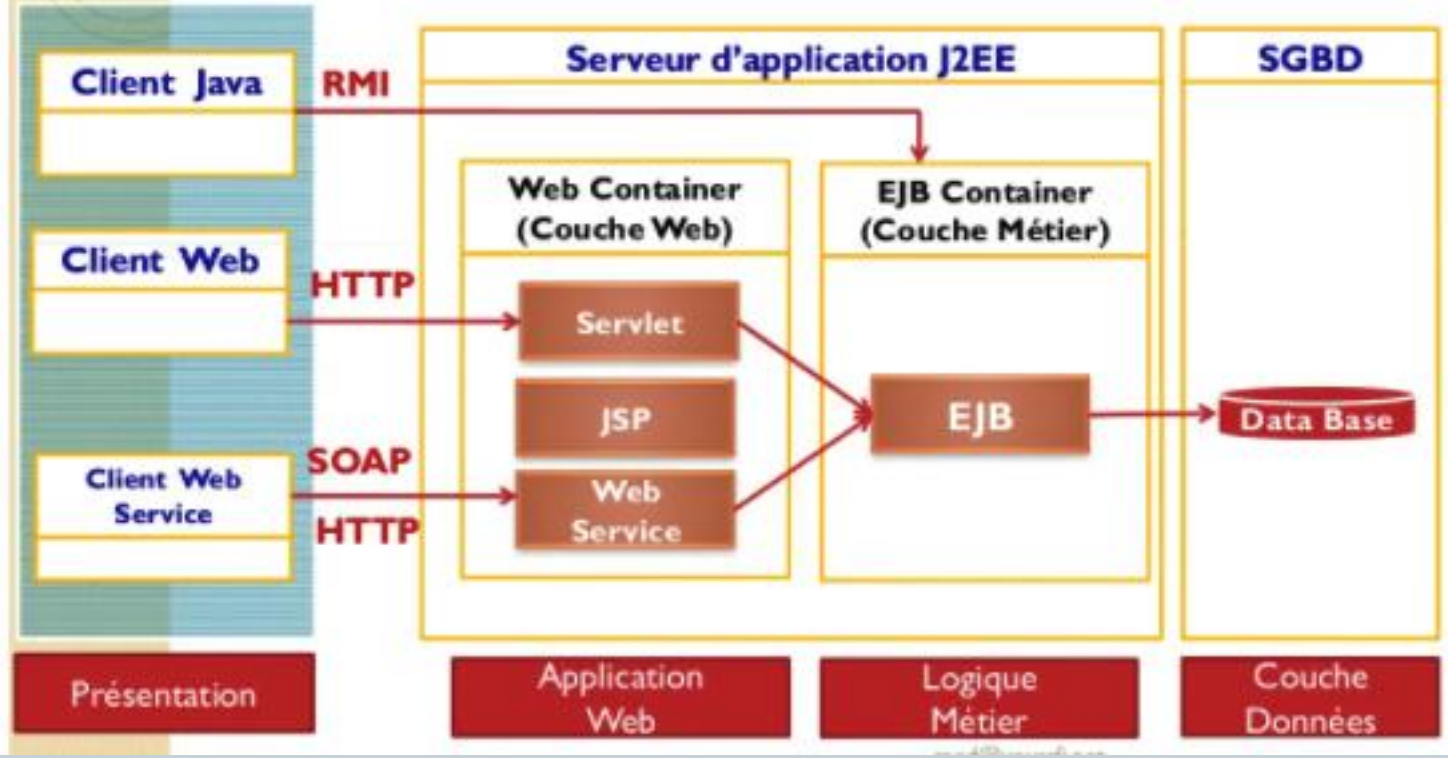
Enterprise JavaBeans

- Exigences de qualité d'un système logiciel: la qualité d'un logiciel se mesure par rapport à plusieurs critères:
- Exigences fonctionnelles
- Exigences techniques
  - Les performances :
    - rapidité d'exécution et temps de réponse
    - Eviter le problème de montée en charge
  - La maintenance : une application doit évoluer avec le temps
  - Sécurité
  - Portabilité
  - Capacité de communiquer avec des composants distants
  - Disponibilité et tolérance aux pannes
  - Capacité de fournir des services à différents types de clients
  - Des interfaces conviviales et user-friendly
  - Coût du logiciel

Architectures Distribués



Couches d'une application



Couche Présentation

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

Couche Application

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

Couche Métier

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

Enterprise JavaBeans

- **Pourquoi utiliser les EJBs?**

Supposons que vous avez besoin de développer une application multi-tiers pour consulter et mettre à jour une base de données à travers une interface web. Vous pouvez accéder à la BD en utilisant JDBC, créer une interface web en utilisant les JSP/servlets, et gérer un système distribué avec RMI.

Mais ! Les contraintes que vous devez considérer quand vous développez un système objet distribué d'entreprise, en plus des APIs, sont :

Performance: Les objets distribués que vous créez doivent être performants, à fin de servir plusieurs clients en même temps. Vous aurez besoin d'utiliser des techniques optimisantes telles que la mise en cache et le pooling des ressources telles que les connexions de base de données. Vous aurez également besoin de gérer le cycle de vie de vos objets distribués.

Sécurité: un objet distribué doit gérer les autorisations des clients qui l'accèdent.

Transactions distribuées : Un objet distribué doit être capable de référencer les transactions distribuées de façon transparente

Réusabilité: Les objets distribués doivent pouvoir être déplacés vers un serveur d'application d'un autre vendeur sans modification, ou recompilation

Enterprise JavaBeans

Ces considerations, en plus du problème métier que vous voulez résoudre, peuvent consister en un projet de développement de taille. Cependant, toutes ces contraintes, à part votre problème métier, sont redondantes (même solution pour toutes les applications distribuées).

Sun, avec d'autres vendeurs, a réalisé que tous les développeurs auront redévelopper ces solutions, et a créé les spécifications Enterprise JavaBeans (EJB).

La plate-forme Java EE propose de mettre en oeuvre les couches métiers et persistance avec les EJB. Particulièrement intéressants dans des environnements fortement distribués

Les EJB décrivent un modèle de composant coté serveur qui met en oeuvre toutes les considérations mentionnées en utilisant une approche standard qui permet aux développeurs de créer des composants métier appelés EJBs qui s'intéresse uniquement à la logique métier.

Puisque les EJBs sont définis de façon standard, ils sont indépendants des vendeurs de serveur.

Plusieurs vendeurs:

- WebLogic (BEA), Sun One (Sun Microsystems), WebSphere (IBM)
- Open source : jBoss, JOnAs

Entreprise Java Beans (EJB)

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

Enterprise JavaBeans

- JavaBeans vs. EJBs

En raison de la similarité des noms, il existe une confusion entre le modèle des composants JavaBeans et les spécifications des Enterprise JavaBeans.

Les JavaBeans et les spécifications des Enterprise JavaBeans partagent les mêmes objectifs de réutilisabilité et de portabilité; cependant les motifs derrière chaque spécification sont destinés à gérer différents problèmes :

- Les standards définis dans le modèle des composants JavaBeans sont designés pour créer des composants réutilisables qui peuvent être **visuels**.
- Les spécifications des Enterprise JavaBeans définissent un modèle de composants pour développer du code java **coté serveur**. Parce que les EJBs peuvent tourner sur différentes plateformes coté serveur, y compris les mainframes qui n'ont pas un affichage visuel , un EJB ne peut pas utiliser des librairies graphiques telles que AWT ou Swing.

Enterprise JavaBeans

Rôle	Responsabilité
Fournisseur de l'Enterprise Bean	Développeur responsable de la création des composants EJB.
Assembleur de l'application	Crée et assemble les applications à partir de collection de composants EJB, ainsi que la création de servlets, JSP, swing, etc
Déployeur	Prend les collections des applications EJB et les déploie dans un environnement d'exécution (un ou plusieurs conteneurs)
Conteneur EJB/Fournisseur de serveur	Fournit un environnement d'exécution et des outils utilisés pour déployer, administrer et exécuter les composants EJB
Administrateur du système	Gère et configure les différents composants et services, s'assure que le système est en cours d'exécution.

Enterprise JavaBeans

- **Les composants EJB**

Les composants EJB sont des éléments de logique métier réutilisables qui adhèrent aux spécifications des EJBs. Ceci permet aux composants d'être portables, et permet la réalisation d'autres services (sécurité, mise en cache, transactions distribuées). Un fournisseur d'Enterprise Bean est responsable du développement des composants EJBs.

Le descripteur de déploiement n'est plus obligatoire puisqu'il peut être remplacé par l'utilisation d'annotations dédiées directement dans les classes des EJB.

1- Conteneur ou Serveur EJB

Le conteneur EJB est un environnement d'exécution qui contient et exécute les composants EJB et fournit à ces composants un ensemble de services standards.

Les responsabilités du conteneur EJB sont strictement définies par les spécifications (non pas le vendeur).

Le conteneur EJB vous permet de facilement gérer les transactions distribuées, la sécurité, le cycle de vie des bean, la mise en cache, le threading, et les sessions.

Les EJB et l'injection de dépendances

- **Java Naming and Directory Interface (JNDI) utilisé par les EJB 2**
  - **JNDI** est une API Java (J2SE) qui permet :
    - d'accéder à différents services de nommage ou de répertoire de façon uniforme ;
    - d'organiser et rechercher des informations ou des objets par nommage ;
    - de faire des opérations sur des annuaires
  - Un service de nommage permet d'associer un nom unique à un objet et de faciliter ainsi l'obtention de cet objet.
  - Il existe plusieurs types de services de nommage parmi lesquels :
    - DNS (Domain Name System) : service de nommage utilisé sur internet pour permettre la correspondance entre un nom de domaine et une adresse IP
    - LDAP(Lightweight Directory Access Protocol) : annuaire
    - NIS (Network Information System) : service de nommage réseau développé par Sun Microsystems
  - JNDI propose donc une abstraction pour permettre l'accès à ces différents services de manière standard. Cependant, avec les EJB 3, les annotations remplacent JNDI.
- **L'utilisation de l'injection de dépendances remplace l'utilisation implicite de JNDI.**

Les EJB et l'injection de dépendances

- **L'utilisation de l'injection de dépendances remplace l'utilisation implicite de JNDI.**
- L'EJB déclare les ressources dont il a besoin à l'aide d'annotations. Le conteneur va injecter ces ressources lorsqu'il va instancier l'EJB
- Ces ressources peuvent être de diverses natures : référence vers un autre EJB, contexte de sécurité, contexte de persistance, contexte de transaction, ...
- Plusieurs annotations sont définies pour mettre en oeuvre l'injection de dépendances :
  - L'annotation @EJB permet d'injecter une ressource de type EJB.
  - L'annotation @Resource permet d'injecter une ressource qui est obtenue par JNDI (EntityManager, UserTransaction, SessionContext, ...)
  - ...

Enterprise JavaBeans

- **Java Naming and Directory Interface (JNDI)**
  - L'API JNDI est contenue dans cinq packages :

Packages	Rôle
javax.naming	Classes et interfaces pour utiliser un service de nommage
javax.naming.directory	Etend les fonctionnalités du package javax.naming pour l'utilisation des services de type annuaire
javax.naming.event	Classes et interfaces pour la gestion des événements lors d'un accès à un service
javax.naming.ldap	Etend les fonctionnalités du package javax.naming.directory pour l'utilisation de la version 3 de LDAP
javax.naming.spi	Classes et interfaces dédiées aux Service Provider pour le développement de pilotes

Enterprise JavaBeans

- **Java Naming and Directory Interface (JNDI)**

  - **Comment utiliser JNDI ?**
  - Un service de nommage permet d'associer un nom à un objet. Cette association est nommée **binding**. Un ensemble d'associations nom/objet est nommé un **contexte** qui est utilisé lors de l'accès à un élément contenu dans le service.
  - Il existe deux types de contexte :
    - Contexte racine (Par exemple c:\ )
    - Sous contexte (par exemple un répertoire)
  - Pour pouvoir utiliser un service de nommage, il faut tout d'abord obtenir un contexte racine qui va encapsuler la connexion au service.
  - À partir de ce contexte, il est possible de réaliser plusieurs opérations :
    - bind : associer un objet avec un nom
    - rebind : modifier une association
    - unbind : supprimer une association
    - lookup : obtenir un objet à partir de son nom
    - list : obtenir une liste des associations

Enterprise JavaBeans

- **Java Naming and Directory Interface (JNDI)**
  - **Obtenir un objet :**

```
import javax.naming.*;
public String getValeur() throws NamingException {
    Context context = new InitialContext();
    return (String) context.lookup("/config/monApplication");
}
```

- La classe javax.Naming.InitialContext encapsule le contexte racine : c'est le noeud qui sert de point d'entrée lors de la connexion avec le service de nommage.
- Toutes les opérations réalisées avec JNDI sont relatives à ce contexte racine.

Enterprise JavaBeans

- **Java Naming and Directory Interface (JNDI)**

- Pour obtenir une instance de la classe InitialContext et ainsi réaliser la connexion au service, plusieurs paramètres sont nécessaires :
  - **java.naming.factory.initial** permet de préciser le nom de la fabrique proposée par le fournisseur. Cette fabrique est en charge de l'instanciation d'un objet de type InitialContext
  - **java.naming.provider.url** : URL du context racine

- **jndi.properties** : contient les informations qui permettront à l'application cliente de se connecter au service de nommage du **serveur JBoss** :

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
java.naming.provider.url=localhost:1099
```

java.naming.factory.url.pkgs: le nom de la propriété environnement qui spécifie le nom du package utilisé par JBoss pour charger l'url du context factory.

Enterprise JavaBeans

- **Java Naming and Directory Interface (JNDI)**

- Ou bien manuellement dans le code de l'EJB :

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,"org.jnp.interfaces.NamingC
ontextFactory");
env.put(Context.PROVIDER_URL, "localhost:1099");
InitialContext ic = new InitialContext(env);
```

Enterprise JavaBeans

- **Java Naming and Directory Interface (JNDI)**

- Il est aussi possible de rechercher un fichier dans un répertoire. Dans ce cas, le contexte initial précisé est le répertoire dans lequel le fichier doit être recherché. La méthode lookup() recherche uniquement dans ce répertoire

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,"com.sun.jndi.fscontext.RefFSCont
    extFactory");
env.put(Context.PROVIDER_URL, "file:c:/");
 try {
    Context ctx = new InitialContext(env);
    File fichier = (File) ctx.lookup("boot.ini");
    System.out.println("objet trouve = " + fichier);
} catch (NamingException e) {
e.printStackTrace();
}
```

Enterprise JavaBeans

- **Java Naming and Directory Interface (JNDI)**

  - **Stocker un objet :**

```
import javax.naming.*;
public void createName() throws NamingException {
    Context context = new InitialContext();
    context.bind("/config/monApplication", "valeur");
}
```

Enterprise JavaBeans

- **API deTransaction java/ Service de transaction java (JTA/JTS)**

JTA/JTS est utilisé avec les Enterprise JavaBeans comme l'API des transactions. Un fournisseur d'Enterprise Bean peut utiliser le JTS pour créer une transaction.

Le déployeur peut définir les attributs des transactions d'un composant EJB au moment du déploiement.

Le conteneur EJB est responsable du support des transactions locales ou distribuées.

- **CORBA et RMI/IIOP**

CORBA est une architecture logicielle, pour le développement de composants qui sont assemblés afin de construire des applications complètes, et peuvent être écrits dans des langages de programmation distincts.

Les spécifications EJB définissent l'interopérabilité avec CORBA : possibilité de réaliser une application avec des technologies hétérogènes EJBs et CORBA.

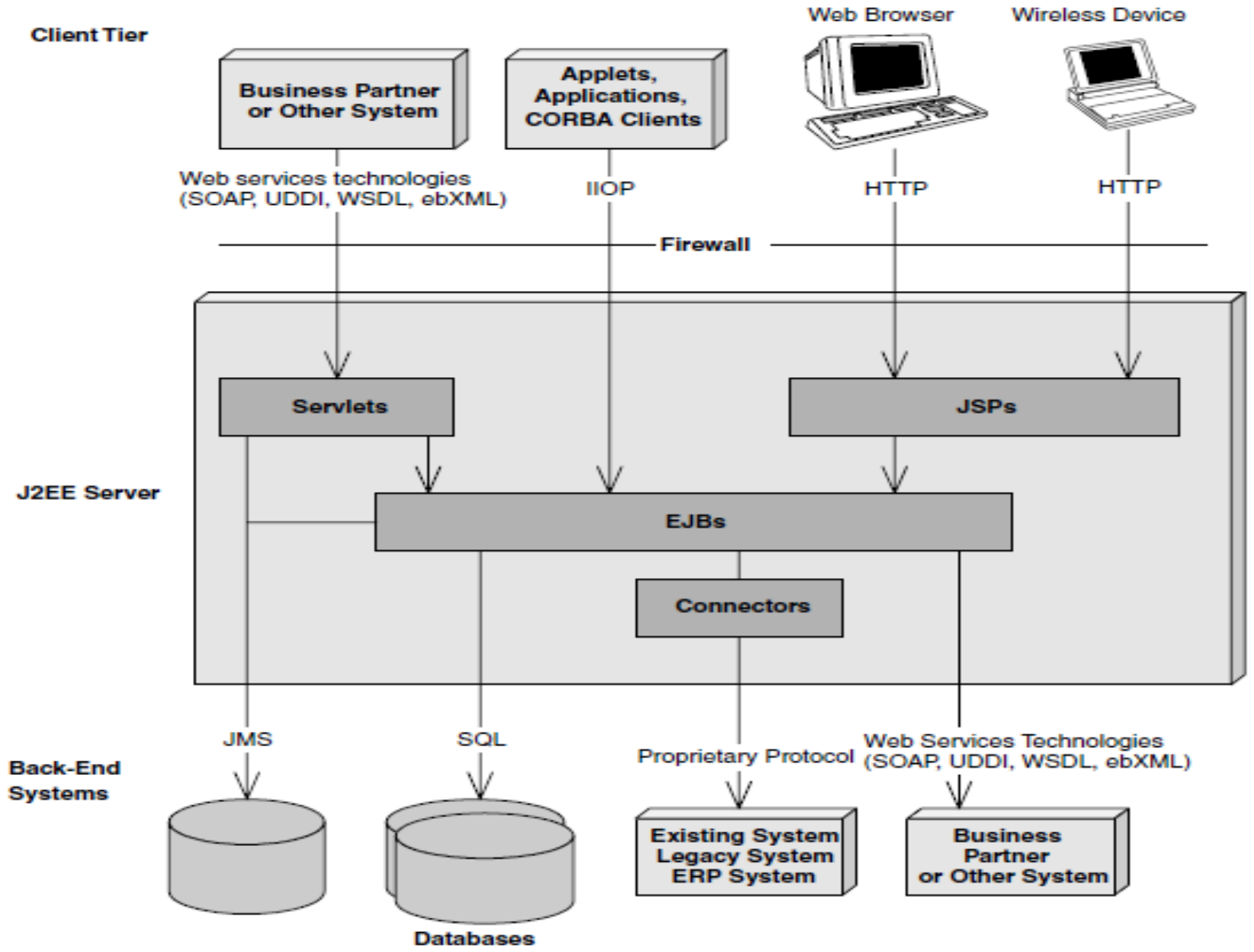
Enterprise JavaBeans

- EJB / Servlets et JSP :
  - EJB et Servlets/JSP ne sont pas des technologies concurrentes mais **complémentaires.**
  - Les Servlets sont orientés requêtes : support pour le front end (UI, client), adaptés pour les applications simples de requête/réponse .
  - Les EJBs sont orientés serveur : support pour les applications demandant plus de performances.
  - La plus importante différence entre EJB et Servlet est qu'avec les EJBs il y'a séparation entre le code de l'interface et entre le code de la logique métier .

Enterprise JavaBeans

- EJB / Servlets et JSP :
  - Ce que peut réaliser les Servlets/JSPs et pas les EJBs :
    - Répondre à des requêtes http/https
    - Fournir un moyen simple de création de sortie HTML
  - Ce que peut réaliser les EJBs et pas les Servlets/JSPs :
    - Gestion déclarative des transaction : le conteneur supporte la gestion, spécification automatique du début et fin de la transaction. Si un problème arrive, l'EJB le signale au conteneur qui annule la transaction. Quand l'EJB finit son traitement, il rend le contrôle au conteneur qui met fin à la transaction.
    - Gestion déclarative de la sécurité : autorisation d'accès automatique gérées ar les annotations

    Avec les Servlets, la gestion se fait par programmation : il faut écrire manuellement le code pour gérer adéquatement les transactions et la sécurité (hard code: logique métier mêlée au code de gestion de transaction ou de sécurité).

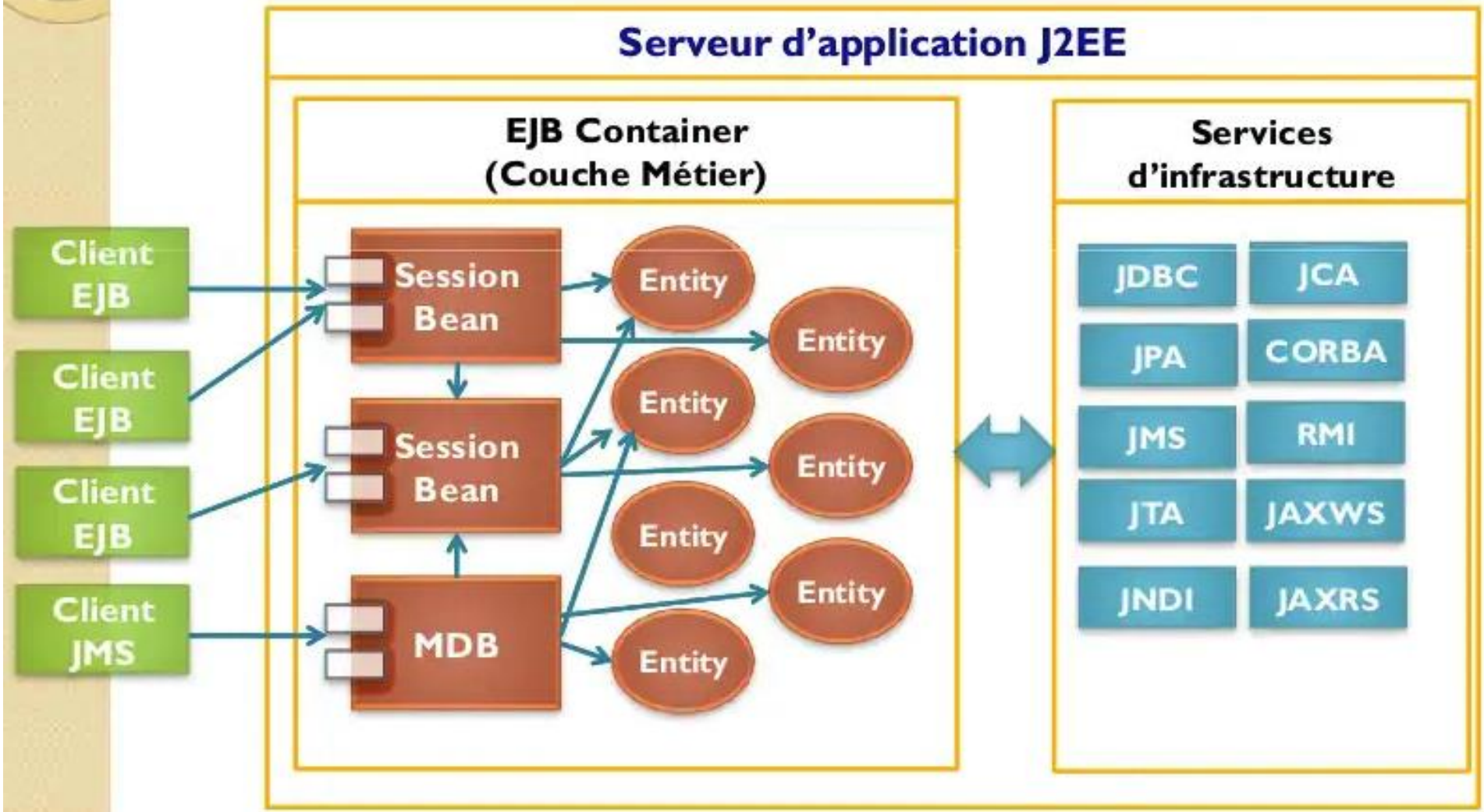

Client Tier
Business Partner or Other System
Applets, Applications, CORBA Clients
Web Browser
Wireless Device
Web services technologies (SOAP, UDDI, WSDL, ebXML)
IIOP
HTTP
HTTP
Firewall
J2EE Server
Servlets
JSPs
EJBs
Connectors
Back-End Systems
JMS
SQL
Proprietary Protocol
Web Services Technologies (SOAP, UDDI, WSDL, ebXML)
Databases
Existing System Legacy System ERP System
Business Partner or Other System

Différents types d'EJB

Il existe trois types d'EJB :

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

Conteneur EJB et Services d'infrastructures



Conteneur EJB

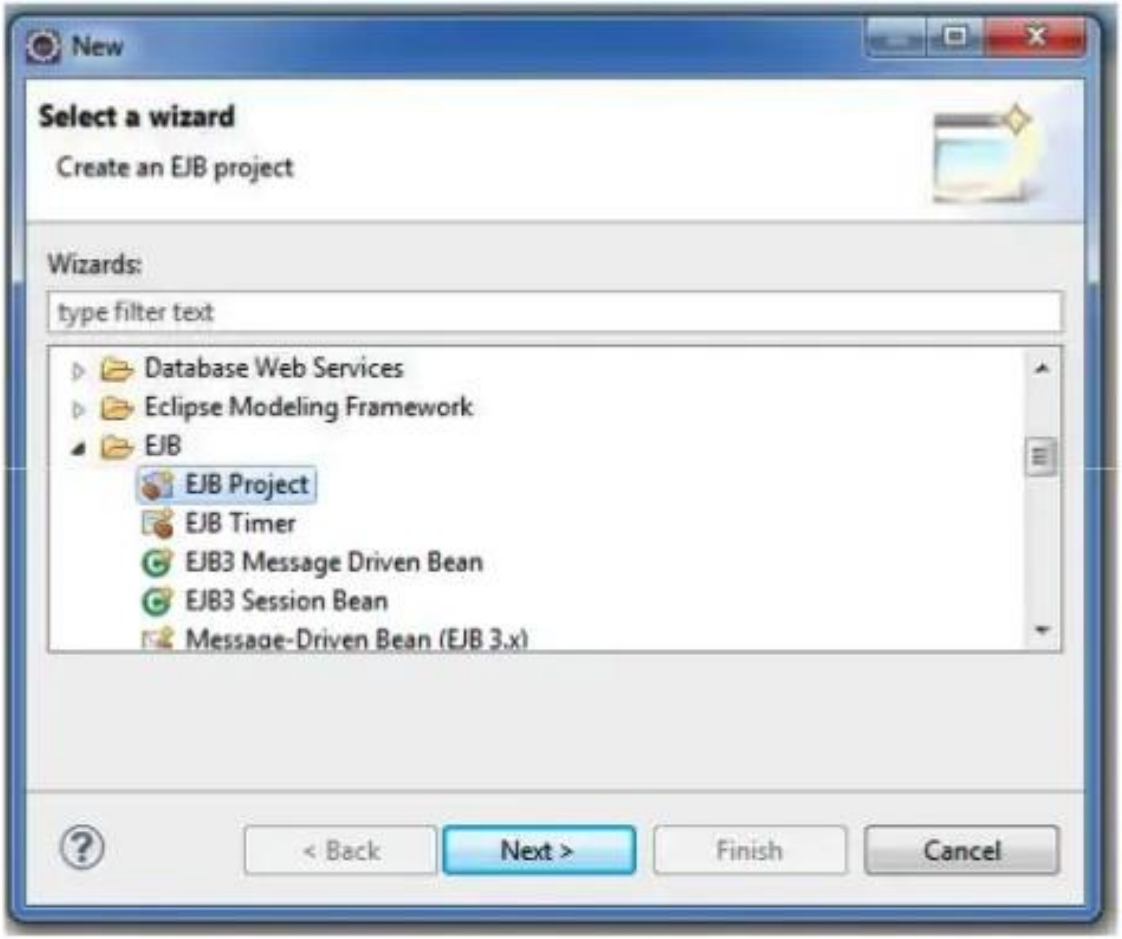
- Le rôle du conteneur EJB et de :
  - Gérer le cycle de vie des composants EJB de votre application
    - Instanciation
    - Accès au bean
    - Sérialisation
    - Désérialisation
    - Destruction du bean
  - Permettre aux EJB d'accéder aux services d'infrastructures offerts par le serveur d'application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

Enterprise JavaBeans

Bean session :

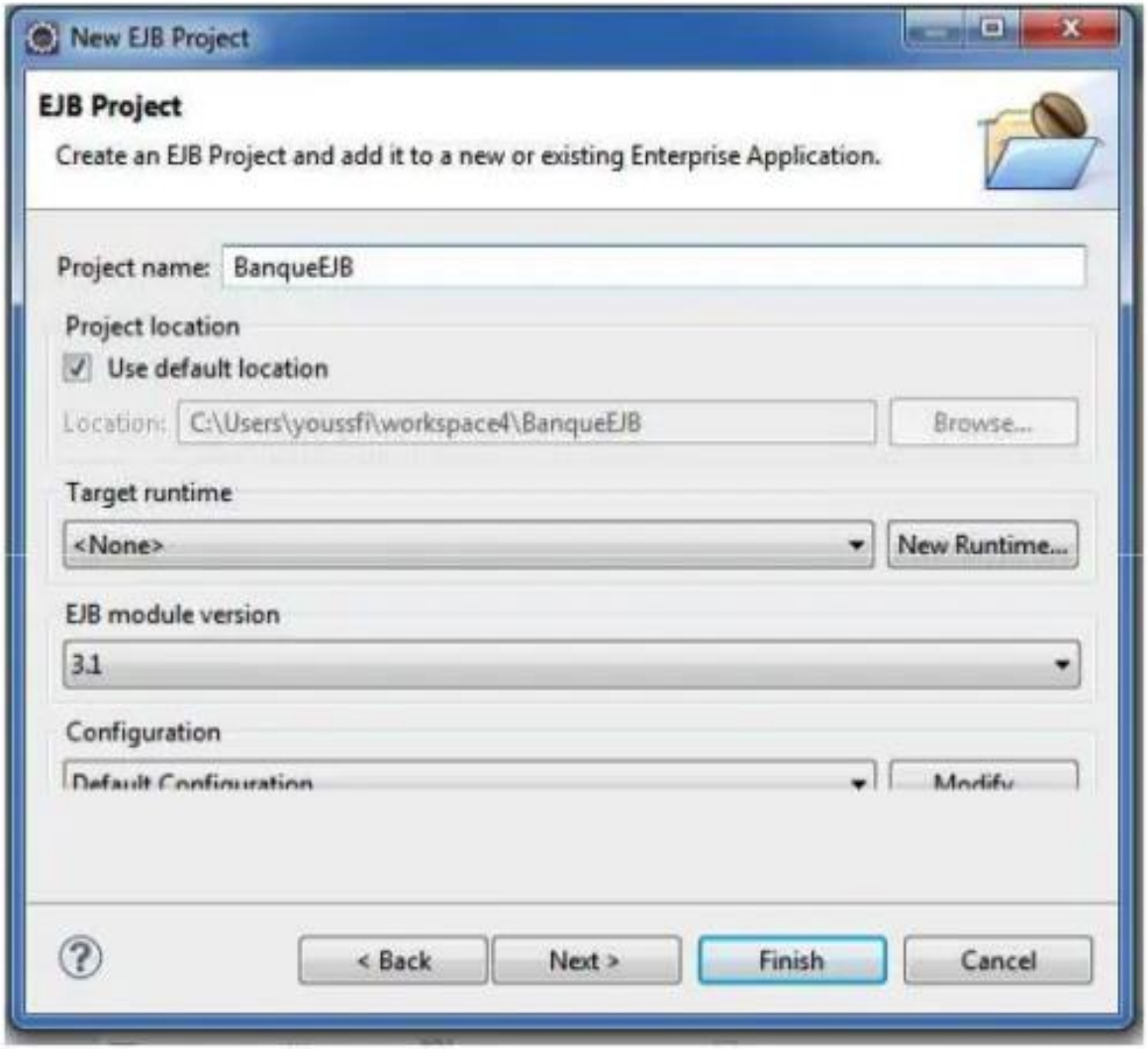
- Les Beans session sont dédiés à implémenter la logique de l'application, il sont responsables d'un groupe de fonctionnalités
- Une application peut contenir plusieurs Beans session
- Exemple : une application de gestion d'une école peut posséder un Bean session contenant la logique nécessaire pour gérer les étudiants, un autre Bean session contiendra la liste des cours et des programmes

Création d'un projet EJB



- New > EJB Project

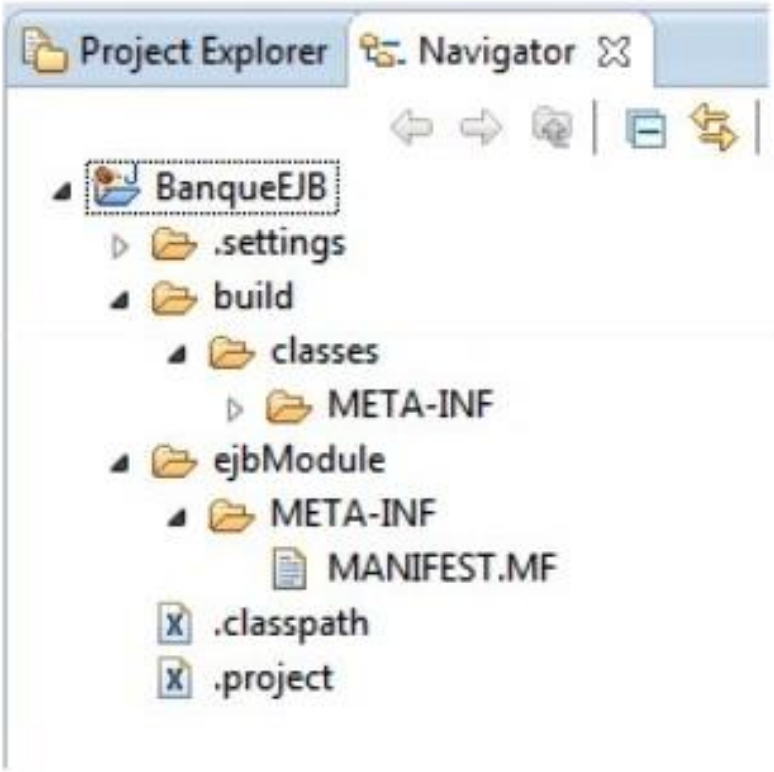
Création d'un projet EJB



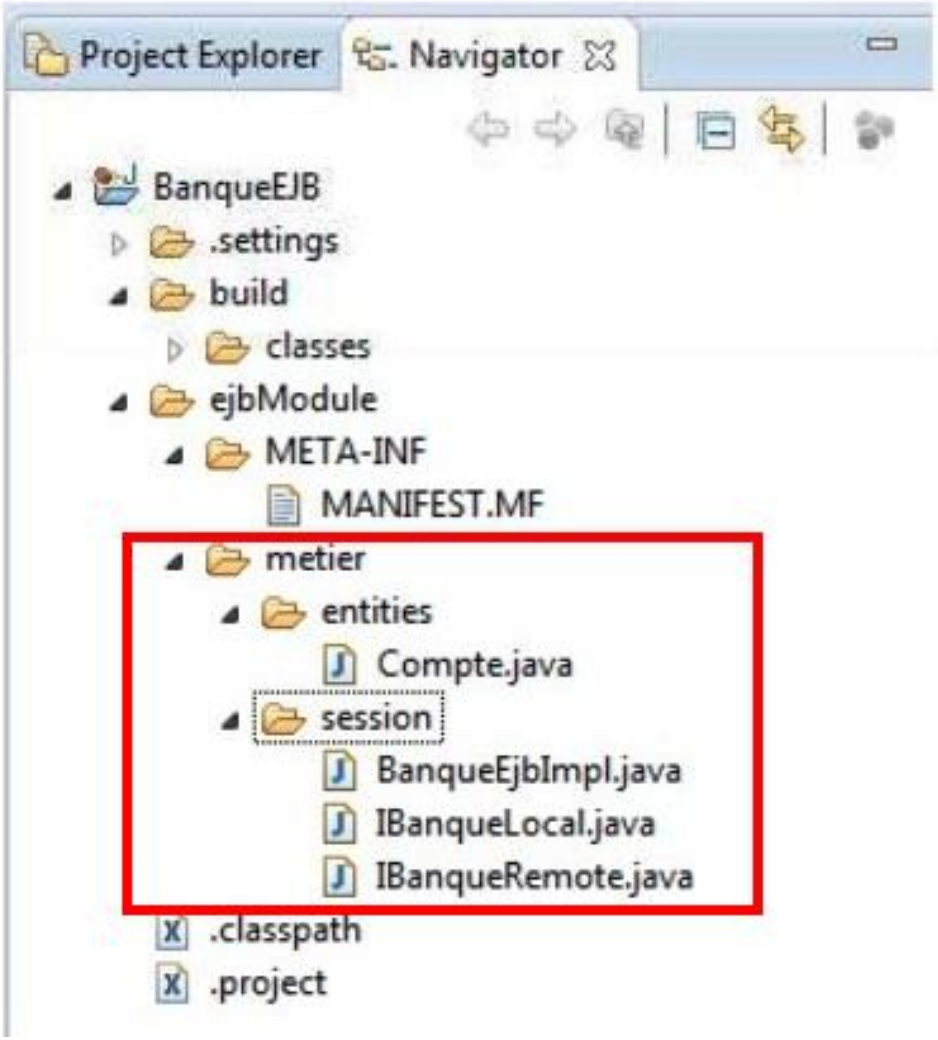
- Après avoir spécifié le nom du projet,
- Cliquez sur le bouton New Runtime, pour associer un serveur à ce projet

Structure du Projet EJB

- Projet Vide



- Projet à créer



Enterprise JavaBeans

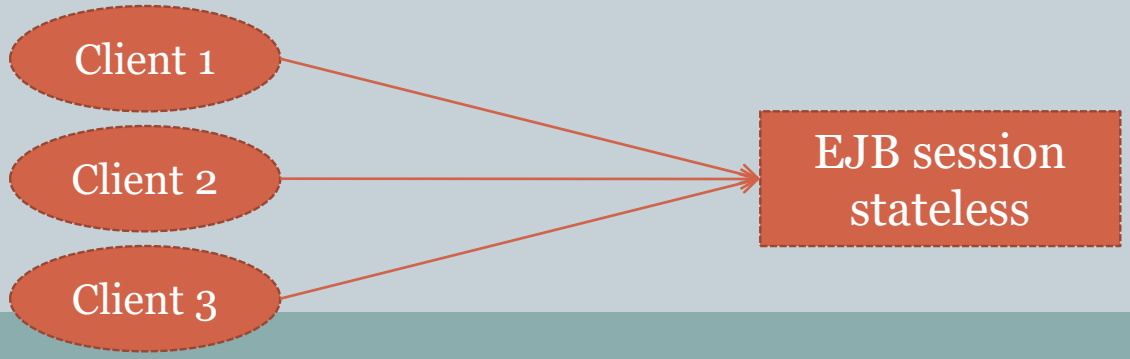
Deux type de Bean session :

- Sans état (stateless): ces Beans ne gardent aucun état. Ils ne stockent aucune information par rapport au clients ou leurs requêtes.

Un Bean session stateless sert uniquement à l'exécution d'un traitement grâce aux paramètres qui lui sont passés, et retourne le résultat au client. Une fois le traitement terminé, le Bean stateless ne concerve aucun souvenir de cette interaction.

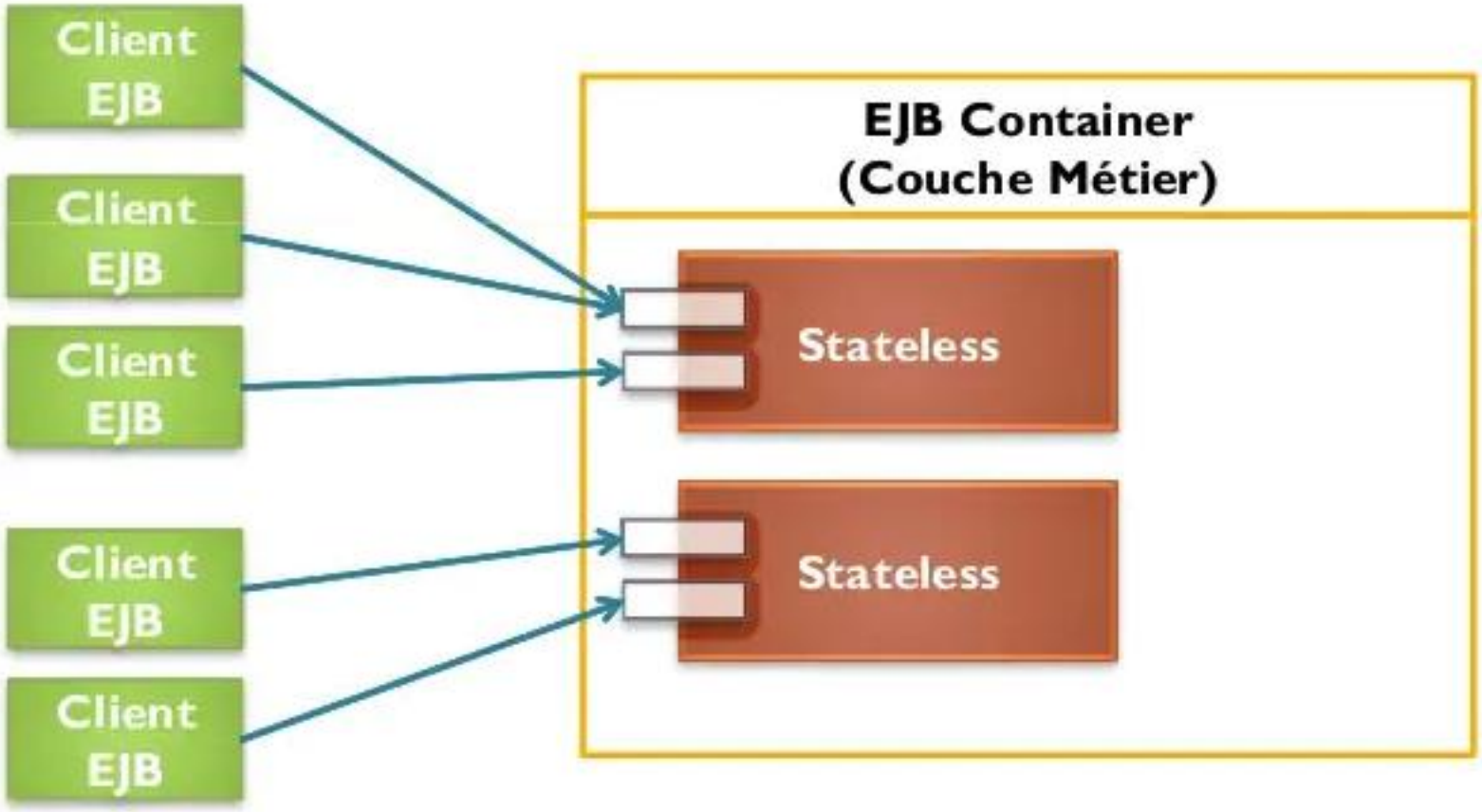
➔ Le Bean Stateless peut être utilisé (appelé) simultanément par plusieurs clients: le même EJB peut retourner un résultat à un client et commencer immédiatement le même traitement pour un autre client.

Il existe une relation n-à-un entre le client et le Bean stateless.



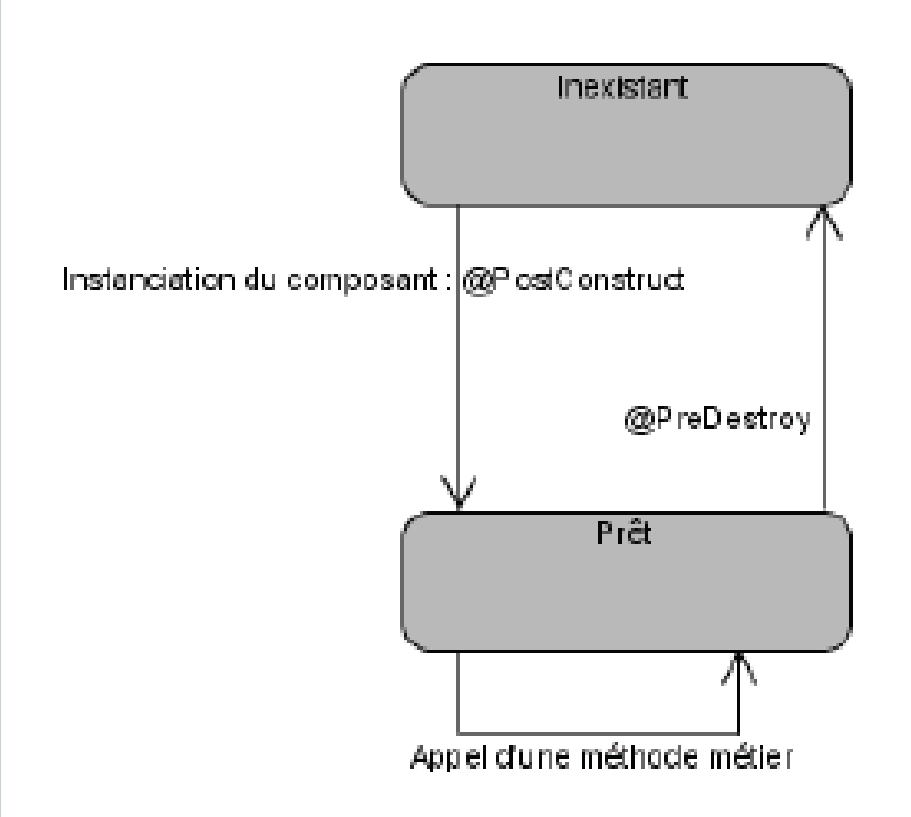
EJB Session Stateless

- Création d'un pool d'instances



Enterprise JavaBeans

- Cycle de vie d'un EJB session stateless:



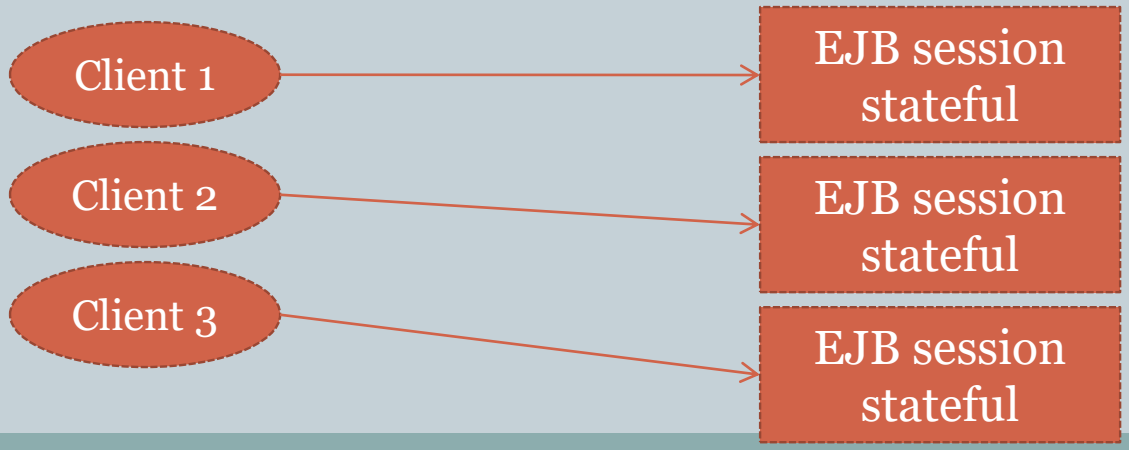
Enterprise JavaBeans

- Bean session avec état (stateful): ces Beans peuvent mémoriser l'état du client entre deux appels de méthodes. Le client peut initialiser les variables du Bean stateful, et elles restent disponibles lors des appels de méthodes suivantes.

Le Bean stateful offre un ensemble de traitements au client, mais aussi la possibilité de stocker des données.

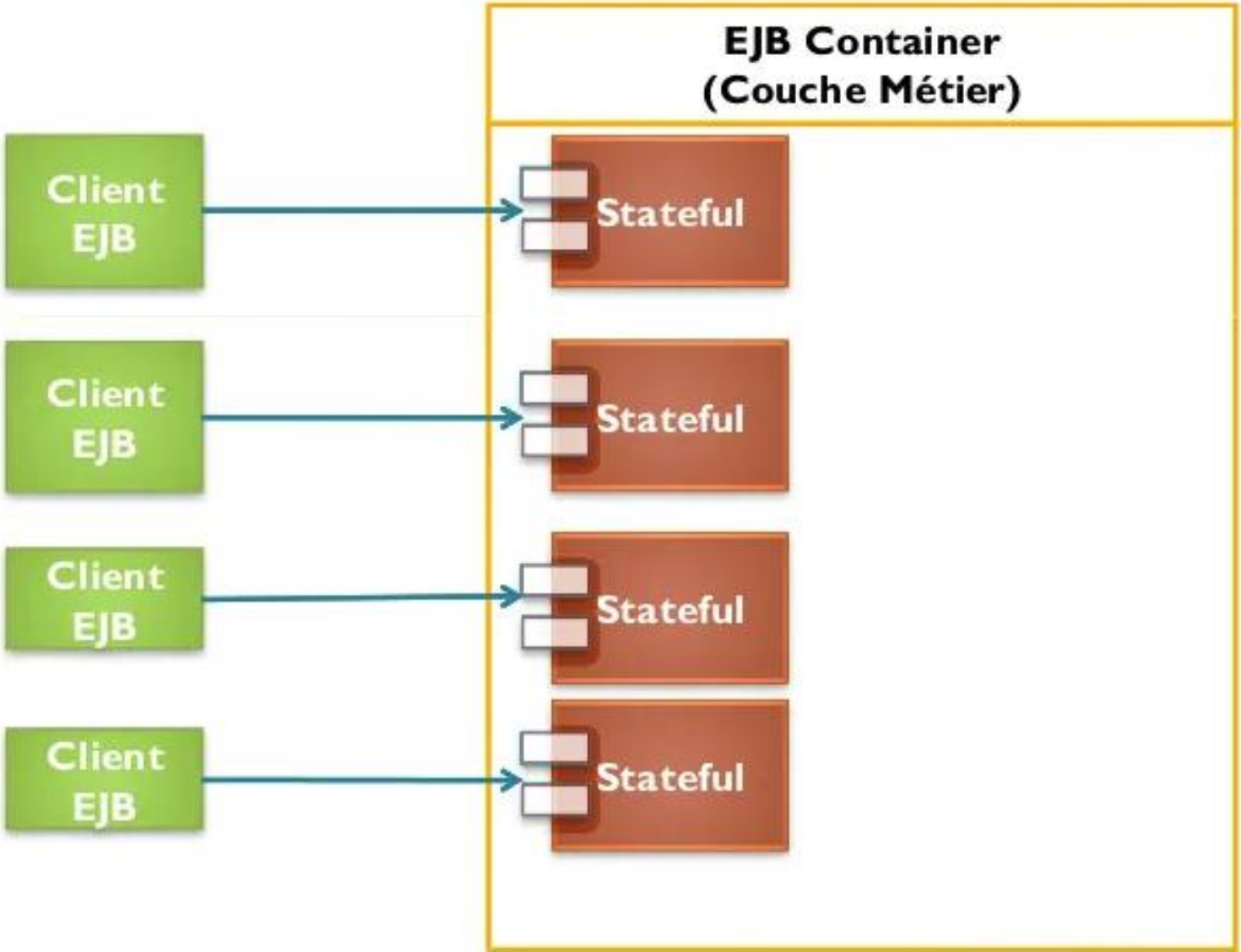
Mais: ces données sont détruites quand le Bean est détruit **à la fin de la session** du client.

Pour chaque client, un Bean stateful est associé : Il existe une relation un-à-un entre le client et le Bean stateful.



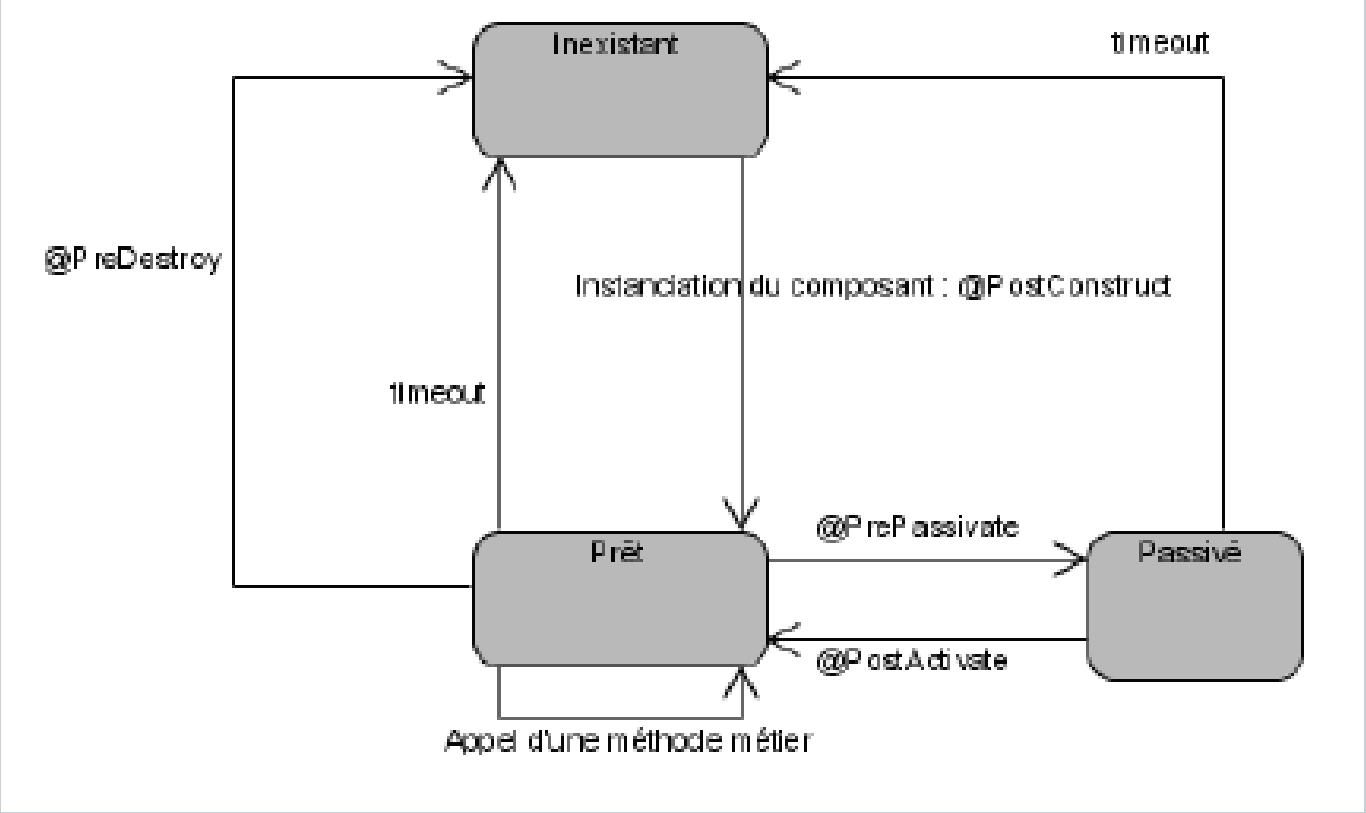
EJB Session Stateful

- Création d'une instance pour chaque connexion



Enterprise JavaBeans

- Cycle de vie d'un EJB session stateful:



Enterprise JavaBeans

Remarque :

- Les Beans session sans état supportent bien la montée en charge, car stocker l'état des beans est une opération qui demande beaucoup de ressources au serveur ➔ les beans avec état ne se prêtent pas bien à l'augmentation du nombre de requêtes envoyées au serveur.

Enterprise JavaBeans

Les Beans contrôlés par les messages:

- Les Bean session stateless et stateful offrent des services de manière **synchrone**: le client émet une requête puis attend le résultat (protocole RMI)
- Lorsqu'une application doit recevoir de manière **asynchrone** des messages provenant d'autres systèmes, elle peut mettre en œuvre des Beans contrôlés par messages : Les clients n'appellent pas directement les méthodes mais utilisent **JMS** pour produire un **message** et le publier dans une file d'attente.
- Le Bean contrôlés par les messages reste à l'écoute; à la réception d'un nouveau message, il l'extrait de la file d'attente et le traite.
- Exemple: dans une application de commerce électronique, un module de vente en gros pourrait utiliser un composant de type Bean contrôlé par les messages pour recevoir les commandes envoyés par les détaillants.

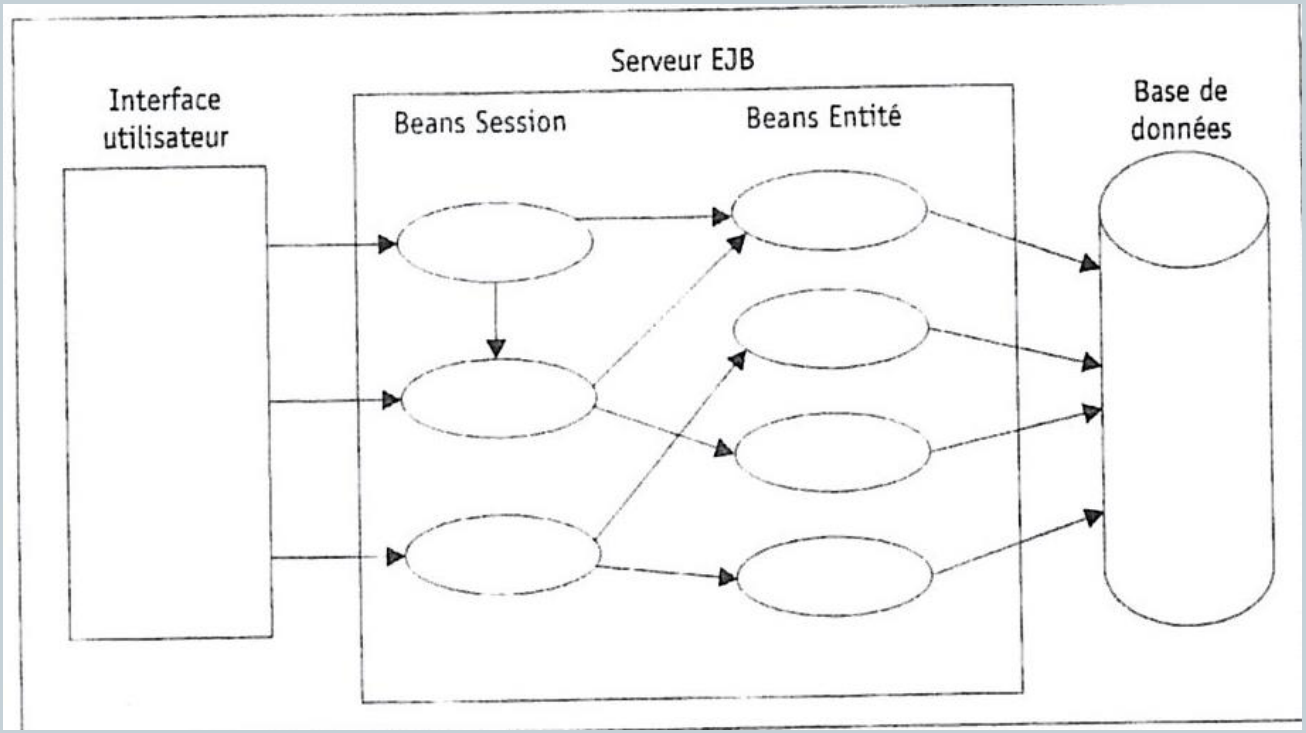
Enterprise JavaBeans

Les Beans entité:

- Utiliser des applications orientés objet avec des bases de données relationnelles pose un problème de différence fondamentale entre les deux technologies
- Les Beans session et les Beans contrôlés par les messages ne peuvent pas stocker les données de façon permanente: données métier qui durent après la fin de la session.
- L'utilisation des Beans entités permet de bénéficier du meilleur des deux mondes objet et relationnel:
  - Les beans entités sont des objets➔approche orientée objet
  - Les données contenues dans ces objets résident dans des BDs relationnelles ➔les développeurs bénéficient de tous les avantages de cette technologie

- Les Beans entité représentent les **données persistantes** d'une application EJB (voir framework de mapping Objet-relationnel Hibernate)

Enterprise JavaBeans



Enterprise JavaBeans

- **EJB 2 : Les éléments d'un composant EJB :**

Un EJB consiste en un nombre d'éléments, incluant le Bean lui-même, l'implémentation de quelques interfaces, et un fichier d'information. Tout ceci est packagé dans un fichier jar spécial :

1- Enterprise Bean

2- interface Home

3- interface distante

4- Descripteur de déploiement :

- Les attributs configurés définis dans le descripteur de déploiement incluent :
  - Le nom de l'interface Home et de l'interface distante requis par l'EJB
  - Le nom à publier dans le JNDI pour l'interface Home de l'EJB
  - Des attributs de transaction pour chaque méthode de l'EJB
  - Les liste du contrôle d'accès pour l'authentification

Enterprise JavaBeans

- EJB 2 pour J2EE 1.4
- EJB 3 pour J2EE 1.5
  - Simplification de la manipulation de l'EJB : **élimination des deux interfaces de spécification home et remote (component) de la spécification EJB 2.0 au profit d'une seule interface en 3.0 (remote ou business).**
  - Simplification de la classe d'implémentation : **élimination de la nécessité d'implémenter une sous interface de l'interface javax.ejb.EnterpriseBean**.
  - Les enterprise beans session peuvent avoir plus d'une interface remote
  - Introduction du **mécanisme d'injection**
  - Introduction des **annotations** Java 1.5 (metadata) comme alternative aux descripteurs de déploiement.

  Les annotations permettent de marquer ou annoter des éléments afin de leur ajouter une propriété particulière. Ces annotations sont utilisées lors de la compilation ou l'exécution pour automatiser certaines tâches (ex. appels de JNDI)

- Codage de l'Enterprise Bean : L'enterprise bean a besoin du code suivant:
  - Interface Remote (ou business)
  - Classe de l'Enterprise bean

Enterprise JavaBeans

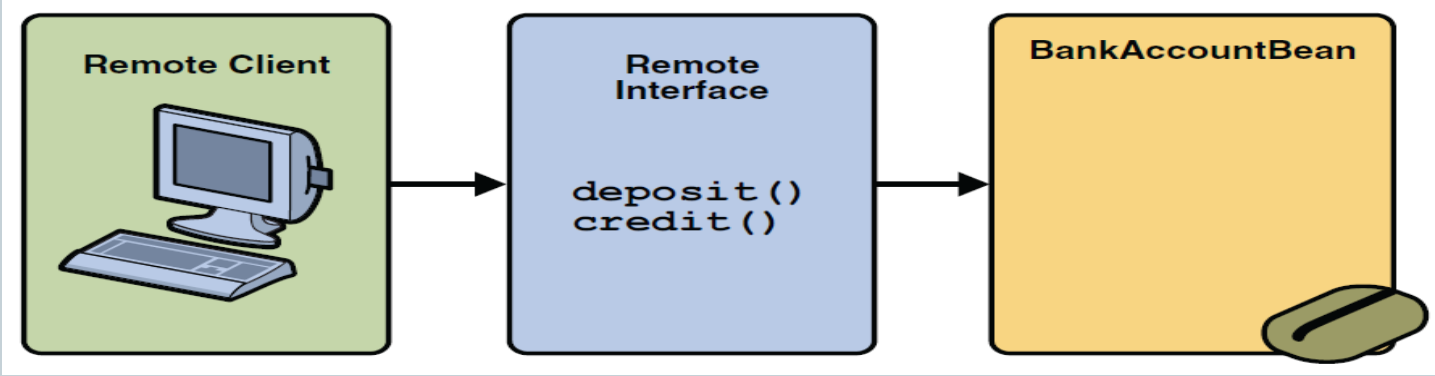
- Quand on conçoit une application JEE, l'une des premières décisions à faire est de déterminer le type du client autorisé à accéder aux enterprise beans: distant, local ou web service

1- Clients distants

- Un client distant d'un enterprise bean a les caractéristiques suivantes :
  - Il peut tourner sur différentes machines et différentes Java virtual machine (JVM) que l'enterprise bean auquel il accède
  - Il peut être un composant web, une application cliente, ou un autre enterprise bean.
- Les paramètres sont passés par le processus de sérialisation: La sérialisation est un procédé qui permet de rendre un objet persistant. Cet objet est mis sous une forme binaire, sous laquelle il pourra être reconstitué à l'identique. Ainsi il pourra être stocké sur un disque dur ou transmis au travers d'un réseau pour le créer dans une autre JVM. C'est le procédé qui est utilisé par RMI.

La sérialisation est aussi utilisée par les beans pour sauvegarder leurs états.

Enterprise JavaBeans



Enterprise JavaBeans

- Pour créer un enterprise bean qui permet l'accès aux clients distants , vous devez faire l'une des choses suivantes:
  - Ajouter à l'interface remote de l'enterprise bean l'annotation **@Remote**

```
@Remote
public interface InterfaceName { ... }
```

  - Ajouter à la classe du bean l'annotation @Remote, spécifiant l'interface ou les interfaces remote:

```
@Remote(InterfaceName.class)
public class BeanName implements InterfaceName { ... }
```

Enterprise JavaBeans

- **Clients locaux**
- Un client local a deux caractéristiques :
  - Il doit tourner dans la même JVM que l'enterprise bean auquel il accède
  - Il peut être un composant web ou un autre enterprise bean.

- L'interface remote local définit les méthodes du bean

- Pour construire un enterprise bean qui permet uniquement l'accès aux clients locaux, vous pouvez faire l'une des choses suivantes:

  - Ajouter à l'interface remote de l'enterprise bean l'annotation **@Local**

  Exemple :

  @Local

  public interface InterfaceName { ... }

Enterprise JavaBeans

■ Ajouter à la classe du bean par l'annotation @Local en spécifiant l'interface.
Exemple :

```
@Local(InterfaceName.class)
public class BeanName implements InterfaceName { ... }
```

- *Si l'interface remote ne contient pas l'annotation* @Local ou @Remote, et la classe du bean ne spécifie pas l'interface en utilisant @Local or @Remote, l'interface remote est par défaut une interface **locale**.

Enterprise JavaBeans

- **Décider du type de l'interface: Remote ou Local**
- La décision dépend des facteurs suivants:
  - **Couplage fort ou lâche** : les beans fortement couplés dépendent l'un de l'autre.

  Par exemple, un bean session qui traite les ordres de vente appelle un bean session qui envoie un email de confirmation au client. Ces deux beans sont de bons candidats pour un accès local, ils s'appèlent souvent et doivent bénéficier de la performance augmentée avec l'accès local.
  - **Type of client**: si l'enterprise bean est accédé par des application clientes, alors il doit autoriser l'accès distant : les clients tournent sur des machines différentes du serveur d'application. Si les clients de l'enterprise bean sont des composants web ou d'autres enterprise beans, alors le type du client dépend du choix du développeur.
  - **La distribution des composants**: dans une application distribuée, les composants web peuvent tourner sur des serveurs différents de celui de l'enterprise bean; dans ce scénario, l'enterprise bean doit permettre l'accès aux clients distants.
  - **Performance**:à cause de facteurs tel que la latence du réseau, les appels distants peuvent être plus lents que les appels locaux. D'un autre coté, si on distribue les composants sur différents serveurs, on peut améliorer les performances globales de l'application.

  Les clients distants ajoutent du coût à cause du processus de sérialisation qui permet de transmettre les paramètres via le réseau (appel distant de méthode).

Enterprise JavaBeans

- Si vous n'êtes pas sûr du type du client (distant ou local), choisissez le type distant. Cela vous donne plus de flexibilité: dans le futur, vous pouvez distribuer vos composants pour faire évoluer votre application.

- Vous pouvez déclarer deux interfaces, distante et local, avec des méthodes différentes dans chaque interface

Enterprise JavaBeans

Client Web Service :

- L'Architecture orienté service (SOA) définit un type d'architecture logicielle basée sur les services
- Les web Services permettent d'implémenter la SOA à l'aide de SOAP, WSDL, UDDI
- Utiliser l'annotation @WebMethod dans l'interface remote

Enterprise JavaBeans

Exemple:

- **Codage de l'interface remote**
- L'**interface** remote définit les méthodes métier qu'un client peut appeler. Les méthodes métier sont implémentées dans la classe de l'enterprise bean.

- Exemple : interface remote de l'Enterprise Bean Converter :

```
package com.sun.tutorial.javaee.ejb;

import java.math.BigDecimal;
import javax.ejb.Remote;

@Remote
 public interface Converter {
 public BigDecimal dollarToYen(BigDecimal dollars);
 public BigDecimal yenToEuro(BigDecimal yen);
 }
```

Remarquez l'annotation @Remote insérée dans la définition de l'interface; ceci permet au conteneur de savoir que ConverterBean va être accédé par des clients distants.

Enterprise JavaBeans

- **Codage de la classe de l'Enterprise Bean**
- La classe de l'enterprise bean implémente les méthodes de l'interface remote.
- Exemple : classe de l'Enterprise Bean Converter. Cette classe est appelé ConverterBean. Elle implémente les deux méthodes (dollarToYen et yenToEuro) :

```
package com.sun.tutorial.javaee.ejb;
import java.math.BigDecimal;
import javax.ejb.*;

@Stateless
public class ConverterBean implements Converter {
private BigDecimal yenRate = new BigDecimal("115.3100");
private BigDecimal euroRate = new BigDecimal("0.0071");

public BigDecimal dollarToYen(BigDecimal dollars) {
BigDecimal result = dollars.multiply(yenRate);
return result.setScale(2, BigDecimal.ROUND_UP); }

public BigDecimal yenToEuro(BigDecimal yen) {
BigDecimal result = yen.multiply(euroRate);
return result.setScale(2, BigDecimal.ROUND_UP);
} }
```

Remarquez l'annotation @Stateless qui permet au conteneur de savoir que ConverterBean est un bean session sans état.

Enterprise JavaBeans

- **Codage de l'application cliente de l'enterprise bean**
- La classe **ConverterClient.java** contient les tâches de:
  - Création d'une instance de l'enterprise bean
  - Invocation des méthodes

- **Création d'une référence à une instance Enterprise Bean**
- Les clients des applications Java EE refèrent aux instances des enterprise beans avec l'annotation **@EJB**.

Avec **le mécanisme d'injection**, le code suivant déclare une variable de type de l'interface 'converter', et l'annotation @EJB appelle automatiquement JNDI pour obtenir une référence, et l'injecter dans la variable locale 'convert':

```
@EJB
private static Converter convert;
```

Enterprise JavaBeans

- **Invocation des méthodes**
- Appeler une méthode de l'enterprise bean se fait à partir de l'objet déclaré convert.
- Le conteneur de l'EJB va invoquer les méthodes correspondantes de l'instance ConverterBean qui tourne sur le serveur
- Exemple :

  BigDecimal param = new BigDecimal ("100.00");

  BigDecimal amount = convert.dollarToYen(param);

Enterprise JavaBeans

- Code source de ConverterClient **:**

```
package com.sun.tutorial.javaee.ejb;
import java.math.BigDecimal;
import javax.ejb.EJB;
public class ConverterClient {
//Déclaration de variables
@EJB
private static Converter convert;

public ConverterClient(String[] args) {
}
public static void main(String[] args) {
ConverterClient client = new ConverterClient(args);
client.doConversion();
}
public void doConversion() {
try {
BigDecimal param = new BigDecimal("100.00");
BigDecimal yenAmount = convert.dollarToYen(param);
System.out.println("$" + param + " is " + yenAmount
+ " Yen.");
BigDecimal euroAmount = convert.yenToEuro(yenAmount);
System.out.println(yenAmount + " Yen is " + euroAmount
+ " Euro.");
System.exit(0);
} catch (Exception ex) {
System.err.println("Caught an unexpected exception!");
ex.printStackTrace();
}}}
```

Enterprise JavaBeans

- Localiser l'EJB avec l'utilisation explicite de JNDI:

```
InitialContext ic = new InitialContext();
converter = (Converter) ic.lookup(ConverterBean/remote);
```

- **jndi.properties** : contenant les informations qui permetteront à l'application cliente de se connecter au service de nommage du serveur JBoss :

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
java.naming.provider.url=localhost:1099
```

Enterprise JavaBeans

- **Création du client converterWeb**
- Le client web est une page JSP :

```
<%@ page import="converter.ejb.Converter,
java.math.*, javax.naming.*"%>
<%!
private Converter converter = null;
public void jspInit() {
try {
InitialContext ic = new InitialContext();
convert = (Converter)
ic.lookup(ConverterBean /remote));
} catch (Exception ex) {
System.out.println("Couldn't create converter bean."+
ex.getMessage());
}
}
public void jspDestroy() { contenu dans une page JSP
converter = null;
}
%>
<html><head><title>Converter</title></head>
<body bgcolor="white« ><h1>Converter</h1>
<hr> <p>Enter an amount to convert:</p>
<form method="get">
<input type="text" name="amount" size="25">
<br>
<p>
<input type="submit" value="Submit">
<input type="reset" value="Reset">
</form>
<%
String amount = request.getParameter("amount");
if ( amount != null && amount.length() > 0 ) {
BigDecimal d = new BigDecimal(amount);
BigDecimal yenAmount =
    convert.dollarToYen(d);
%>
<p> <%= amount %> dollars are <%= yenAmount %>
    Yen.
<p><% BigDecimal euroAmount =
convert.yenToEuro(yenAmount); %>
<%= amount %> Yen are <%= euroAmount %> Euro.
<% } %>
</body> </html>
```

TP

Projet Site Web Marchand

Le site doit permettre les cas d'utilisation suivants :

- Un client peut consulter des articles dans un catalogue
- Un client peut choisir des articles et les ajouter à un panier
- Un client peut commander les articles choisis
- Le vendeur peut ajouter, enlever ou modifier des articles au catalogue
- Le vendeur peut consulter, ajouter, modifier ou supprimer des commandes.
- Le vendeur peut consulter, ajouter, modifier ou supprimer des fiches client.

Informations complémentaires :

- Une commande comporte la liste des articles choisis, une référence sur la fiche client, l'adresse de livraison, l'adresse de facturation.
- Une fiche client comporte les coordonnées du client : nom, prénom, téléphone, adresse de facturation préféré, adresse de livraison préféré, …
- Un article comporte son nom, sa référence, son prix, sa description, ses catégories …
- Un catalogue permet de rechercher des articles selon différents critères : nom, référence, catégories, …

Les beans Entité

EJB 3 : Entity Bean

- **Persistance des objets :**

- A la fin d'une session d'utilisation d'une application orientée objet toutes les données des objets existants dans la mémoire vive de l'ordinateur sont perdues

- Rendre persistant un objet c'est sauvegarder ses données sur un support non volatile de telle sorte qu'un objet identique à cet objet pourra être recréé lors d'une session ultérieure

- Une application écrite avec un langage objet utilise généralement une base de données relationnelle pour la persistance des données des objets : Les SGBD objet n'offrent pas la performance et la capacité d'adaptation à la charge des SGBD relationnels

- ➔Problème d'adaptation de la structure des données relationnelles au modèle objet

EJB 3 : Entity Bean

- **Pourquoi les Beans Entité ?**
  - Les stateful session beans sont détruits lorsque la session du client se termine. Ils ne peuvent donc pas être utilisés pour stocker de façon permanente les informations de l'application. Les EJB entités peuvent répondre à ce besoin puisqu'ils sont persistants.; leur état est sauvegardé après la fin d'une session.

- **JPA (Java Persistence API):**
  - Depuis les débuts de J2EE, le modèle de persistance ne cesse d'évoluer : entity beans 1.0, entity beans 2.1., spécification (JDO, *Java Data Object) et différents outils de mapping* objet/relationnel payants ou libres (TopLink, Hibernate…).
  - Java EE 5 nous apporte un nouveau modèle de persistance : JPA (*Java Persistence API). Fortement inspirés par des outils* Open Source tels qu'Hibernate ou par JDO
  - JPA s'appuie sur JDBC pour communiquer avec la base de données : grâce à l'abstraction apportée par JPA, nous n'aurons nul besoin d'utiliser directement JDBC dans le code Java.

EJB 3 : Entity Bean

- **Couche de mapping objet/relationnel**

- Le principe du mapping objet-relationnel (ORM ou *object-relational mapping) consiste à déléguer l'accès aux données à des outils ou frameworks* externes. Son avantage est de proposer une vue orientée objet d'une structure de données relationnelle (lignes et colonnes).
- Les outils de mapping mettent en correspondance bidirectionnelle les données de la base de données et les objets. Pour cela, ils utilisent l'API JDBC pour exécuter les requêtes SQL.
- Il existe plusieurs API et frameworks permettant de faire du mapping objet-relationnel : les entity beans, Hibernate, TopLink, JDO et JPA.
- La couche de mapping objet/relationnel transforme la représentation physique des données en une représentation objet et inversement.

EJB 3 : Entity Bean

- Dans le modèle de persistance JPA, un entity bean est une simple classe java : on déclare, instancie et utilise cet entity bean tout comme n'importe quelle autre classe.
- Un entity bean possède des attributs (son état) qui peuvent être manipulés via des accesseurs (méthodes get et set).
- Grâce aux **annotations**, ces attributs peuvent être rendus persistants en base de données.
- Premier exemple simple d'entity bean :

```
@Entity
public class Address {
@Id
private Long id;
private String street1;
private String street2;
private String city;
private String state;
private String zipcode;
private String country;
// Accesseurs get/set
}
```

EJB 3 : Entity Bean

- **Remarques :**
- L'annotation **@javax.persistence.Entity** permet à JPA de reconnaître cette classe comme une classe persistante et non comme une simple classe Java.
- L'annotation **@javax.persistence.Id**, définit l'identifiant unique de l'objet. Elle donne à l'entity bean une identité en mémoire en tant qu'objet.
- Tous les attributs seront rendus persistants par JPA en appliquant les paramétrages par défaut : le nom de la colonne est identique à celui de l'attribut et le type String est converti en varchar(255).
- Cet exemple ne comporte que des attributs, mais la classe peut aussi avoir des méthodes métier.
- Notez que cet entity bean Address est une simple classe java. Elle n'implémente aucune interface, se contente d'être annotée par @javax.persistence. Entity et d'avoir un identifiant unique (@javax.persistence.Id). Pour être un entity bean, une classe doit au minimum utiliser ces deux annotations et posséder un constructeur par défaut.
- Grâce à ces annotations, JPA peut **synchroniser les données** entre les attributs de l'entity bean Address et les colonnes de la table Address.
- Ainsi, si l'attribut zipcode est modifié par l'application, JPA se chargera de modifier cette valeur dans la colonne zipcode (en exécutant une requête sql)

EJB 3 : Entity Bean

- **Table correspondante :**

Génération de la base de données relationnelle

```
CREATE TABLE ADDRESS (
ID BIGINT NOT NULL,
STREET1 VARCHAR(255),
STREET2 VARCHAR(255),
CITY VARCHAR(255),
STATE VARCHAR(255),
ZIPCODE VARCHAR(255),
COUNTRY VARCHAR(255),
PRIMARY KEY (ID)
)
```

- Le nom de la table est identique à celui de la classe.
- Les attributs de la classe sont stockés dans des colonnes qui portent le même nom que les attributs de l'entity bean.

EJB 3 : Entity Bean

- L'annotation **@javax.persistence.Table** permet de définir les valeurs par défaut liées à la table. Elle n'est pas obligatoire mais permet, par exemple, de spécifier le nom de la table dans laquelle les données seront stockées.
- Si cette annotation est omise, le nom de la table sera le même que celui de la classe.
- Ainsi, si nous voulions changer le nom de la table en t_address, nous devrions écrire le code suivant :

```
@Entity
@Table(name = "t_address")
public class Address {
@Id
private Long id;
private String street1;
// (...)
}
```

EJB 3 : Entity Bean

- **Clé primaire**
- Comme nous l'avons vu précédemment, un entity bean doit avoir au minimum les annotations @Entity et @Id. **@javax.persistence.Id** annote un attribut comme étant un identifiant unique.
- La valeur de cet identifiant peut être, soit générée manuellement par l'application, soit générée automatiquement grâce à l'annotation **@javax.persistence.GeneratedValue**. Trois valeurs sont alors possibles :
  - La génération de la clé unique se fait de manière automatique **(AUTO)** par la base de données (valeur par défaut).
  - On utilise une séquence **(SEQUENCE)** pour obtenir cette valeur.
  - Les identifiants sont stockés dans une table **(TABLE)**. Simulation de Sequence par une table

- **Exemple :** la classe Address avec une génération automatique de l'identifiant :

```
@Entity
@Table(name = "t_address")
public class Address {
@Id
@GeneratedValue (strategy = GenerationType.AUTO)
private Long id;
private String street1;
// (...)}
```

EJB 3 : Entity Bean

- Exemple 2 :

```
@Entity
@SequenceGenerator(name="ArticleSeq", sequenceName="ART_SEQ")

 public class Article
{
    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE,
  generator="ArticleSeq")
    private long id;
    ...
}
```

EJB 3 : Entity Bean

- **Colonne :**
- L'annotation **@javax.persistence.Column** définit les propriétés d'une colonne. On peut ainsi changer son nom (qui par défaut porte le même nom que l'attribut), préciser son type, sa taille et si la colonne autorise ou non la valeur null.
- Ainsi, pour redéfinir les valeurs par défaut de l'entity bean Address, on peut utiliser l'annotation @Column de différentes manières :

```
@Entity
@Table(name = "t_address")
public class Address {
 @Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
@Column(nullable = false)
private String street1;
private String street2;
 @Column(nullable = false, length = 100)
private String city;
private String state;
```

L'entity bean est stocké dans la table t_address.

L'attribut id est l'identifiant de cet entity bean. Sa valeur est générée automatiquement par la base de données.

L'attribut street1 ne peut être null.

L'attribut city ne peut être null et sa longueur maximale est de 100 caractères.

EJB 3 : Entity Bean

- **Colonne : suite**

```
@Column(name = "zip_code", nullable = false, length = 10)
private String zipcode;
@Column(nullable = false, length = 50)
private String country;
// accesseurs get/set
public Long getId() {return id;}

public String getStreet1() {return street1;}
public void setStreet1(String street1) {
this.street1 = street1;
}
public String getStreet2() {return street2;}
public void setStreet2(String street2) {
this.street2 = street2;
}
}
```

L'attribut zipcode est mappé dans la colonne zip_code de 10 caractères de long.

Il n'y a pas de méthode setId puisque ce n'est pas l'application qui génère l'identifiant mais la base de données!!

Accesseurs des attributs.

EJB 3 : Entity Bean

- **Table correspondante :**

```
CREATE TABLE T_ADDRESS (
ID BIGINT NOT NULL,
CITY VARCHAR(100) NOT NULL,
STATE VARCHAR(255),
STREET2 VARCHAR(255),
ZIP_CODE VARCHAR(10) NOT NULL,
STREET1 VARCHAR(255) NOT NULL,
COUNTRY VARCHAR(50) NOT NULL,
PRIMARY KEY (ID)
)
```

EJB 3 : Entity Bean

- **Code de l'annotation @javax.persistence.Column**

```
package javax.persistence;
 @Target({METHOD, FIELD}) @Retention(RUNTIME)

public @interface Column {
String name() default "";
boolean unique() default false;
boolean nullable() default true;
boolean insertable() default true;
boolean updatable() default true;
int length() default 255;
int precision() default 0;
.....
}
```

Cette annotation s'applique à une méthode ou à un attribut.

Nom de la colonne.

La valeur doit-elle être unique ?
La valeur null est-elle autorisée ?
Autorise-t-on la colonne dans un ordre insert ou update ?

Longueur maximale pour une colonne de type Varchar.
Pour les colonnes de type numérique, on peut rajouter la précision.

EJB 3 : Entity Bean

- Annotations avancées:
- Lors du mapping objet-relationnel, on peut spécifier le type grâce à l'annotation **@javax.persistence.Temporal**. Elle peut prendre trois valeurs possibles : DATE (java.sql.Date), TIME (java.sql.Time) ou TIMESTAMP (java.sql.Timestamp), qui est la valeur par défaut.
- Exemple : Entity bean Customer avec date de naissance de type @Temporal

```
@Entity
@Table(name = "t_customer")
public class Customer {
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
@Column(name = "date_of_birth")
@Temporal(TemporalType.DATE)
private Date dateOfBirth;
// (...)
}
```

EJB 3 : Entity Bean

- **Données non persistées**
- Avec JPA, dès qu'une classe est annotée persistante (@Entity), ses attributs sont tous automatiquement stockés dans une table. Si l'on veut qu'un attribut ne soit pas rendu persistant, on doit utiliser l'annotation **@javax.persistence.Transient**.
- Par exemple, l'âge du client n'a pas besoin d'être rendu persistant en base puisqu'il peut être calculé à partir de la date de naissance.
- Exemple : Entiy bean Customer avec un attribut Transient

```
@Entity
@Table(name = "t_customer")
public class Customer {
(...)
@Column(name = "date_of_birth")
@Temporal(TemporalType.DATE)
private Date dateOfBirth;
@Transient
 private Integer age;                    // L'âge du client n'est pas stocké dans la base de données
}
```

EJB 3 : Entity Bean

- **Englober deux objets dans une seule table :**
- JPA permet d'englober les attributs de deux classes dans une seule table de manière relativement simple.
- La classe englobée utilise l'annotation **@Embeddable** alors que la classe englobante utilise **@Embedded**.
- Prenons l'exemple d'un bon de commande (Order) que l'on règle à l'aide d'une carte bancaire (CreditCard). Ces deux classes peuvent être englobées dans une seule et même table (t_order) :

EJB 3 : Entity Bean

- **Entity bean Order englobant l'entity bean CreditCard**

```
@Entity
@Table(name = "t_order")
public class Order {
@Id
@GeneratedValue
private Long id;
@Embedded
private CreditCard creditCard;
(...)
}
```

Le bon de commande est stocké dans la table t_order.

Notez que nous pouvons ne pas spécifier la stratégie de génération de clé si on utilise la stratégie par défaut (strategy = GenerationType.AUTO).

La classe Order englobe les attributs de la classe CreditCard en utilisant l'annotation **@Embedded.**

- **L'entity bean CreditCard est englobable**

```
@Embeddable
public class CreditCard {
private String creditCardNumber;
private String creditCardType;
private String creditCardExpDate;
}
```

L'entity bean CreditCard est englobable **@Embeddable** La carte de crédit n'est pas annotée par @Entity mais par **@Embeddable.** Cela signifie que ses attributs se retrouvent dans la table de la classe englobante.

EJB 3 : Entity Bean

- **Relations :**
- Le monde de l'orienté objet permet de définir des relations entre classes (associations unidirectionnelles, multiples, héritages, etc.). JPA permet de rendre persistante cette information de telle sorte qu'une classe peut être liée à une autre dans un modèle relationnel.
- Il existe plusieurs types d'associations entre entity beans
- Une association possède **un sens** et peut être unidirectionnelle ou bidirectionnelle (c'est-à-dire qu'on peut naviguer d'un objet vers un autre et inversement).
- Chaque association possède une cardinalité, c'est-à-dire que nous pouvons avoir des liens 0:1, 1:1, 1:n ou n:m.

- **1- Jointures**
- Dans le monde relationnel, il existe deux manières d'avoir une relation entre deux tables : en utilisant les clés étrangères ou les tables de jointures intermédiaires.

EJB 3 : Entity Bean

- **Relation unidirectionnelle 1:1**
- Une relation unidirectionnelle 1:1 entre classes est un lien de cardinalité 1 qui ne peut être accédé que dans un sens.
- Exemple : un bon de commande doit posséder une et une seule adresse de livraison (cardinalité 1). Il est important de naviguer du bon de commande vers l'adresse, par contre, il n'est pas utile de pouvoir naviguer dans le sens inverse. Le lien est donc unidirectionnel.
- JPA utilise l'annotation **@OneToOne** pour définir ce type de lien.
- Relation unidirectionnelle 1:1 entre bon de commande et adresse :

```
@Entity
@Table(name = "t_order")
public class Order {
@Id @GeneratedValue
private Long id;
@OneToOne
@JoinColumn(name = "address_fk", nullable = false)
private Address deliveryAddress;
(...)
}
```

L'annotation **@OneToOne** définit un lien 1-1 entre le bon de commande et l'adresse de livraison.

On utilise l'annotation **@JoinColumn**. pour spécifier que la clé étrangère ne doit pas accepter la valeur null (nullable=false).

EJB 3 : Entity Bean

- À partir des annotations, JPA génère une contrainte d'intégrité référentielle entre la colonne address_fk de t_order et la clé primaire de la table t_address.

- Table correspondante :

```
CREATE TABLE T_ORDER (
ID BIGINT NOT NULL,
ADDRESS_FK BIGINT NOT NULL
PRIMARY KEY (ID)
)
ALTER TABLE T_ORDER
ADD CONSTRAINT T_ORDER_ADDRESS_FK
FOREIGN KEY (ADDRESS_FK) REFERENCES T_ADDRESS (ID)}
```

- Pour rendre ce lien unidirectionnel, il suffit de ne pas avoir d'attribut Order dans la classe Address.

EJB 3 : Entity Bean

- **Relation unidirectionnelle 0:1**
- Une relation unidirectionnelle 0:1 est implémentée de la même manière qu'une relation 1:1. La seule différence réside dans le fait qu'elle est optionnelle.
- Exemple d'un client et de son adresse : un client n'est pas obligé de fournir son adresse au système. Par contre, si le client souhaite la saisir, il ne peut en avoir qu'une.
- Pour transformer une relation 1:1 en 0:1, il suffit d'autoriser la valeur null dans la colonne. Pour cela, il est nécessaire de positionner l'attribut **nullable** de l'annotation **@JoinColumn** à true.
- Relation unidirectionnelle 0:1 entre client et adresse :

```
@Entity
@Table(name = "t_customer")
public class Customer {
@Id @GeneratedValue
private Long id;
@OneToOne
@JoinColumn(name = "address_fk", nullable = true)
private Address homeAddress;
(...)
}
```

EJB 3 : Entity Bean

- **Table correspondante :**

```
CREATE TABLE T_CUSTOMER (
ID BIGINT NOT NULL,
ADDRESS_FK BIGINT
PRIMARY KEY (ID)
)
ALTER TABLE T_ CUSTOMER
ADD CONSTRAINT T_CUSTOMER_ADDRESS_FK
FOREIGN KEY (ADDRESS_FK) REFERENCES T_ADDRESS (ID)}
```

EJB 3 : Entity Bean

- **Relation bidirectionnelle 1:n**
  - Une relation 1:n signifie qu'un objet fait référence à un ensemble d'autres objets (cardinalité n).
  - Exemple : une catégorie d'un catalogue contient plusieurs produits. De plus, elle est bidirectionnelle puisque le produit a connaissance de la catégorie à laquelle il appartient.
  - Cette information de cardinalité en Java est décrite par les structures du paquetage **java.util : Collection, List, Map et Set.**
  - JPA utilise les annotations **@OneToMany** et **@ManyToOne**.
- **Les collections en Java :**
  - Les collections (paquetage java.util) proposent une série de classes, d'interfaces et d'implémentations pour gérer les structures de données (listes, ensembles).
  - Chaque implémentation utilise une stratégie avec des avantages et des inconvénients : certaines collections acceptent les doublons, d'autres non ; certaines sont ordonnées, d'autres pas.
    - Les java.util.Set (ensembles) sont un groupe d'éléments uniques.
    - Les java.util.List (listes) sont une suite d'éléments ordonnés accessibles par leur rang dans la liste. Les listes ne garantissent pas l'unicité des éléments.
    - Les java.util.Map mémorisent une collection de couples clé-valeur. Les clés sont uniques, mais la même valeur peut-être associée à plusieurs clés.

EJB 3 : Entity Bean

- Exemple : Une catégorie possède une liste de produits

```
@Entity
@Table(name = "t_category")
public class Category
(...)
@OneToMany(mappedBy = "category")
private List<Product> products;
(...)
}
```

Une (One) catégorie possède plusieurs (Many) produits. Remarquez l'utilisation des **génériques** pour la liste de produits

- Le produit a connaissance de sa catégorie

```
@Entity
@Table(name = "t_product")
public class Product
(...)
@ManyToOne
@JoinColumn(name = "category_fk")
private Category category;
(...)
}
```

Plusieurs (Many) produits peuvent être rattachés à une (One) même catégorie.
On renomme la colonne de la clé étrangère en category_fk.

EJB 3 : Entity Bean

- JPA peut utiliser deux modes de jointure : le système de clé étrangère ou la table de jointure. Si aucune annotation n'est utilisée, la table de jointure est le mode par défaut. On se retrouve alors avec une table (t_category_product par exemple) contenant deux colonnes permettant de stocker la relation entre catégorie et produit.
- Si ce mode par défaut n'est pas satisfaisant, il faut utiliser l'attribut **mappedBy** de l'annotation @OneToMany.
- Dans notre exemple, le fait que l'entity bean Category déclare @OneToMany(mappedBy = "category") précise à JPA qu'il doit utiliser le système de clé étrangère
- Tables correspondantes :

```
CREATE TABLE T_CATEGORY (
ID BIGINT NOT NULL,
PRIMARY KEY (ID)
)
CREATE TABLE T_PRODUCT (
ID BIGINT NOT NULL,
CATEGORY_FK BIGINT,
PRIMARY KEY (ID)
)
ALTER TABLE T_PRODUCT
ADD CONSTRAINT T_PRODUCT_CATEGORY_FK
FOREIGN KEY (CATEGORY_FK) REFERENCES t_category (ID)
```

EJB 3 : Entity Bean

- **Association multiple sans générique**
- L'entity bean Category utilise les génériques (<>) pour la liste des produits.
- Grâce aux génériques qui typent cette liste, JPA sait que la persistance doit se faire entre Category et Product.
- Si vous n'utilisez pas les génériques, JPA ne saura pas sur quel autre entity bean pointer. Il faut alors spécifier la classe de l'entity bean dans la relation à l'aide de l'attribut **targetEntity** de l'annotation @OneToMany :

```
@Entity
@Table(name = "t_category")
public class Category
(...)
@OneToMany(mappedBy = "category", targetEntity = Product.class)
private List products;
(...)
}
```

EJB 3 : Entity Bean

- **Relation unidirectionnelle 1:n :**

- Les relations unidirectionnelles 1:n sont particulières. En effet, JPA ne permet pas l'utilisation des clés étrangères pour mapper cette relation mais uniquement le système de table de jointure.
- Par défaut, le nom de cette table intermédiaire est composé du nom des deux entity beans séparés par le caractère '_'.
- Les annotations JPA permettent de redéfinir ces valeurs par défaut (en utilisant **@JoinTable**).
- Exemple : un bon de commande (Order) est composé de plusieurs lignes de commande (OrderLine). La relation est donc multiple et la navigation se fait uniquement dans le sens 'Order vers OrderLine'.

EJB 3 : Entity Bean

- Entity bean Order avec une relation unidirectionnelle 1:n vers OrderLine avec l'annotation @JoinTable :

```
@Entity
@Table(name = "t_order")
public class Order
(...)
@OneToMany
@JoinTable(name = "t_order_order_line",
joinColumns = {@JoinColumn(name = "order_fk")},
inverseJoinColumns = {@JoinColumn(name = "order_line_fk")})
private List<OrderLine> orderLines;
(...)
}
```

1- La table de jointure est renommée t_order_order_line ainsi que les colonnes des clés étrangères.

2- **joinColumns :** Colonne de la clé étrangère faisant référence à la clé primaire de la table qui maintient la relation.

3- **inverseJoinColumns :** Colonne de la clé étrangère faisant référence à la clé primaire de la seconde table.

EJB 3 : Entity Bean

- **Tables correspondantes :**

```
CREATE TABLE T_ORDER (
ID BIGINT NOT NULL,
PRIMARY KEY (ID)
)
 CREATE TABLE T_ORDER_LINE (
ID BIGINT NOT NULL,
PRIMARY KEY (ID)
)
CREATE TABLE T_ORDER_ORDER_LINE (
ORDER_FK BIGINT NOT NULL,
ORDER_LINE_FK BIGINT NOT NULL,
PRIMARY KEY (ORDER_FK, ORDER_LINE_FK)
)
ALTER TABLE T_ORDER_ORDER_LINE
ADD CONSTRAINT T_ORDER_FK
FOREIGN KEY (ORDER_FK) REFERENCES T_ORDER (ID)

ALTER TABLE T_ORDER_ORDER_LINE
ADD CONSTRAINT TRORDERLINE_FK
FOREIGN KEY (ORDER_LINE_FK) REFERENCES T_ORDER_LINE (ID)
```

EJB 3 : Entity Bean

L'Héritage:

Les notions de classes abstraites et les stratégies d'héritage utilisées dans JPA sont : (les mêmes définies dans Hibernate)

- Une table pour une hiérarchie de classe
- Une table par classe concrète
- Une table par classe

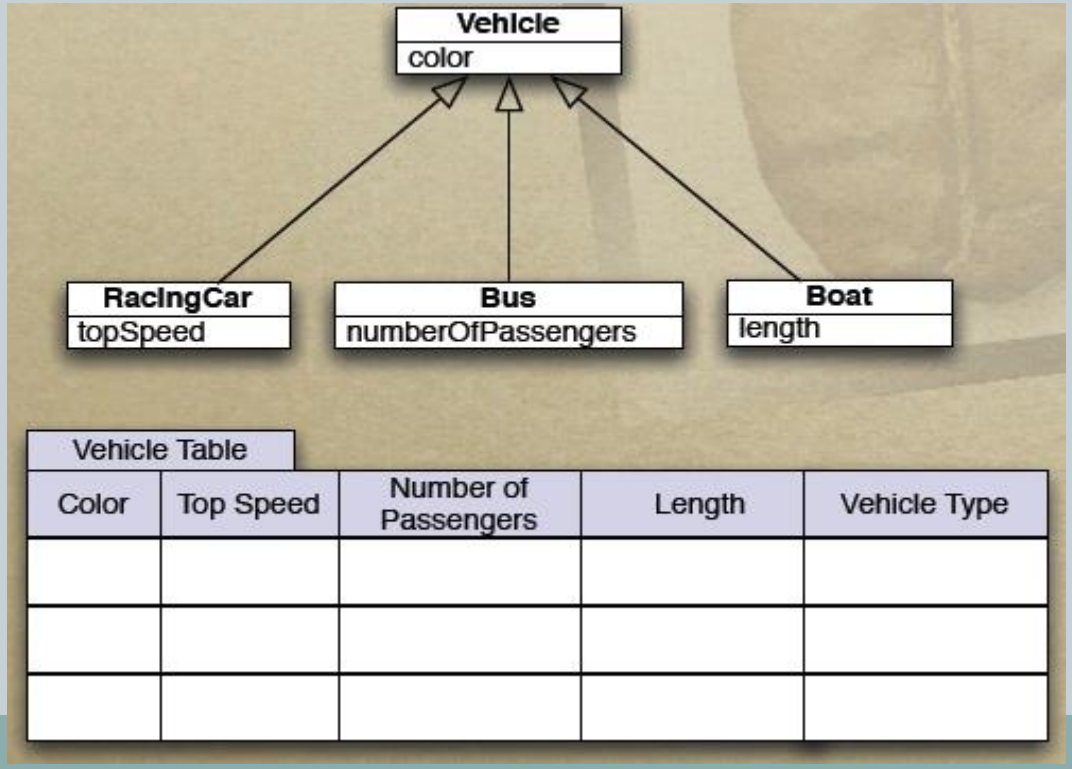
- On choisit le type d'héritage par l'annotation @Inheritance(...)
  - strategy=InheritanceType.ONLY_ONE_TABLE (ou SINGLE_TABLE)
  - strategy=InheritanceType.TABLE_PER_CLASS
  - strategy=InheritanceType.JOINED

EJB 3 : Entity Bean

1- Une table pour une hiérarchie de classe :

● Toutes les propriétés de toutes les classes parentes et classes filles sont mappées dans la même table

● Les instances sont différenciées par une colonne spéciale discriminante



EJB 3 : Entity Bean

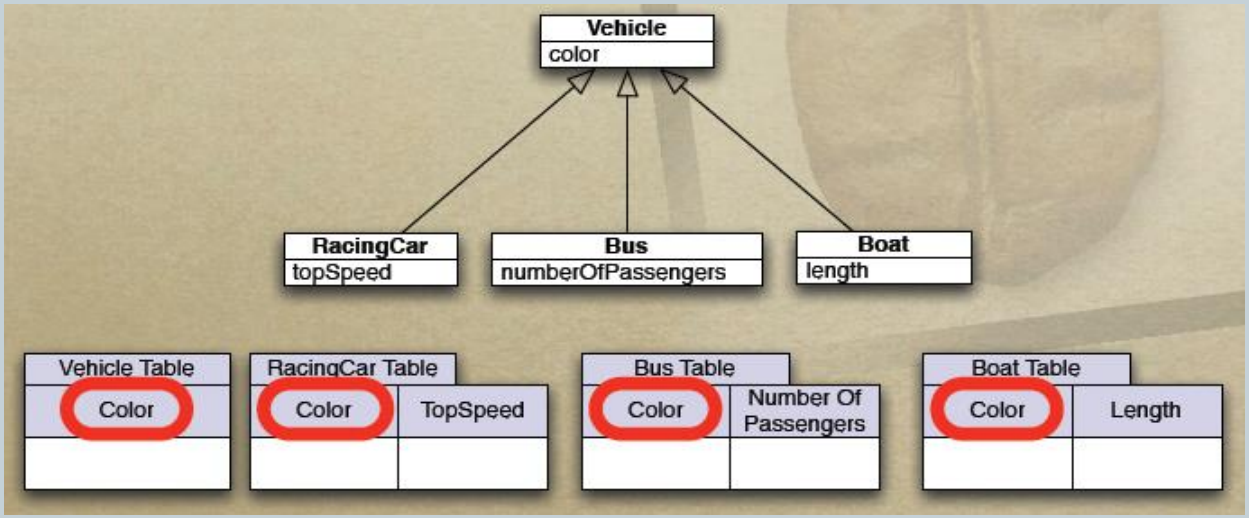
```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
name="vehicletype",
discriminatorType=DiscriminatorType.STRING
)
@DiscriminatorValue("Vehicule")
public class vehicule { … }

@Entity
@DiscriminatorValue("Bus")
public class Bus extends Vehicule { … }
```

La colonne discriminante contient la valeur qui identifie la classe à laquelle appartient l'instance représentée par la ligne

L'annotation DiscriminatorValue peut être utilisée pour préciser la valeur pour chaque classe entité

EJB 3 : Entity Bean

- **Une table par classe concrète :**
- Une table pour chaque entité; les données de la classe mère sont toutes reprises dans les classes filles

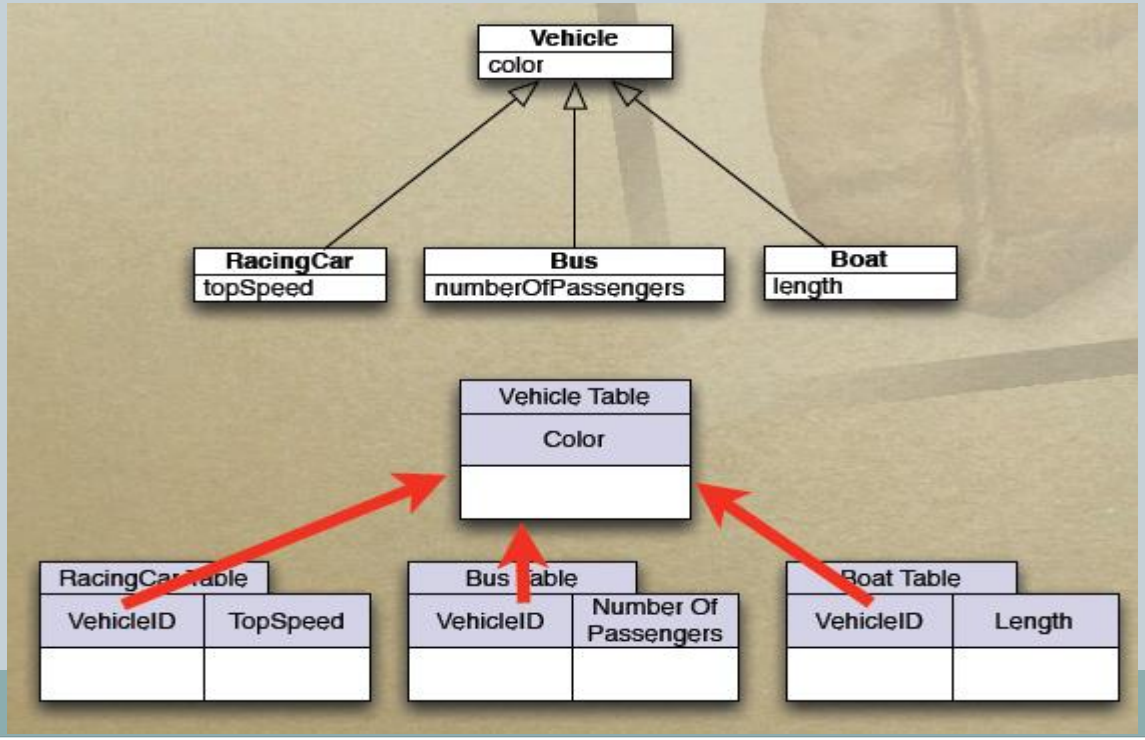



```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Vehicule { ... }
@Entity
public class Bus extends Vehicule { ... }
```

EJB 3 : Entity Bean

Une table par classe :

- Une table pour chaque entité : classe mère ou fille.
- Ce modèle relationnel est le plus proche du modèle objet : à chaque classe, qu'elle soit concrète ou abstraite, correspond une table



EJB 3 : Entity Bean

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Vehicule { … }

@Entity
public class Bus extends Vehicule { … }

@Entity
public class Boat extends Vehicule { … }
```

La classe mère est représentée par une table, chaque sous classe a une table séparée contenant les champs spécifiques à cette sous classe

EJB 3 : Entity Bean

- **Ordonner une association multiple :**

- Les bases de données relationnelles ne préservent pas d'ordre dans leur table. Ainsi, si on veut récupérer une liste ordonnée de telle ou telle manière, il faut utiliser le mot-clé **order by** dans les ordres SQL. Il en va de même pour les listes des entity beans.
- Exemple de la catégorie et ses produits : on veut pouvoir récupérer cette liste de manière ordonnée (par nom de produits ascendant). Pour cela, JPA propose l'annotation **@OrderBy** que l'on peut utiliser sur les annotations @OneToMany et @ManyToMany.
- @OrderBy prend en paramètre les noms des attributs sur lesquels on souhaite effectuer un tri, ainsi que le mode (ascendant ASC ou descendant DESC).

EJB 3 : Entity Bean

- Les produits d'une catégorie sont classés dans l'ordre ascendant du nom :

```
@Entity
public class Category
@OneToMany(mappedBy = "category ")
@OrderBy("name ASC")
private List<Product> products;
(…)
}
@Entity
public class Product
private String name;
@ManyToOne
private Category category;
(…)
}
```

EJB 3 : Entity Bean

- **Cascade :**
- Parfois, lorsqu'on effectue une opération sur un entity bean, on souhaite que celle-ci se propage sur les associations. On parle alors d'action en cascade.
- Exemple : lorsqu'on supprime une catégorie du système, on veut que ses produits soient également supprimés; la catégorie supprime ses produits en cascade
- L'attribut cascade est présent dans les annotations que nous avons vu précédemment @ManyToOne, @OneToMany, et @OneToOne

```
@Entity
public class Category

@OneToMany(cascade = CascadeType.REMOVE)
private List<Product> products;
(...)
}
```

EJB 3 : Entity Bean

- **Stateless Bean et JPA :**
- Quand on veut persister les Entity Beans en base de données, il faut utiliser un **entity manager.**
- Dans JPA, **l'entity manager** est le service centralisant toutes les actions de persistance : Les entity beans ne deviennent persistants que lorsqu'on le précise explicitement dans le code au travers de l'entity manager.
- Celui-ci fournit une API pour créer, rechercher, mettre à jour, supprimer et synchroniser des objets avec la base de données.
- Pour instancier un entity bean en mémoire, il faut utiliser le mot-clé new. Ensuite, pour que les données soient stockées en base, il faut utiliser la méthode **persist()** de l'entity manager

EJB 3 : Entity Bean

- **Utilisation de l'entity manager dans un stateless bean :**

```
@Stateless

public class CustomerBean implements CustomerRemote {
@PersistenceContext (unitName = "GestionClients")
private EntityManager em;
public Customer createCustomer(Customer customer,
Address homeAddress) {
customer.setHomeAddress(homeAddress);
em.persist(customer);
return customer;
}
}
```

Le contexte de persistance informe l'entity manager du type de base de données et des paramètres de connexion.

Déclaration de l'entity manager.

L'entity manager est utilisé pour persister l'entity bean client dans la base de données.

- L'entity manager synchronise automatiquement l'état de l'entity avec la base de données : toutes les opérations effectuées sur l'objet sont exécutées à l'aide de JDBC (SQL) sur la BD.

EJB 3 : Entity Bean

- **Contexte de persistance :**
- Répond à la question : dans quelle base de données doit-il persister ces entity beans ?
- Renseigne sur plusieurs informations : le type de la base de données et les paramètres de connexion à cette base de données
- L'annotation **@javax.persistence.PersistenceContext** est au dessus de la déclaration de l'entity manager.
- Grâce à l'attribut **unitName** = "GestionClients", l'entity manager sait faire le lien avec une unité de persistance qui se nomme GestionClients.
- Cette unité de persistance est définie dans le fichier **persistence.xml**, qui doit être déployé dans le même jar que les entity beans : META-INF/persistence.xml.
- Le fichier persistence.xml définit une ou plusieurs unités persistantes :

- 1- Définir la datasource :

```
<datasources>
<local-tx-datasource>
<jndi-name> GestionClients </jndi-name>
<connection-url>jdbc:mysql://192.168.0.6:3306/ GestionClients </connection-url>
<driver-class>com.mysql.jdbc.Driver</driver-class>
<user-name>user</user-name>
<password>password</password>
</local-tx-datasource> </datasources>
```

EJB 3 : Entity Bean

- 2- Créer le fichier persistance.xml

Nom utilisé dans le statless Bean

```
<persistence>
 <persistence-unit name=" GestionClients ">
<jta-data-source>java:/ GestionClients </jta-data-source>
<properties>
<property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/>
    </properties>
 </persistence-unit>
 </persistence>
```

- EJB 3.0 de JBoss est construit au dessus d'Hibernate 3.0. Vous pouvez avoir besoin de fournir des informations à Hibernate pour qu'il sache le dialecte fournisseur de la bases de données (MySQL, Oracle, etc ..)
- **Remarque** : si vous utilisez la version d'Eclipse qui permet de construire un 'Projet JPA', le fichier persistence.xml sera généré automatiquement

EJB 3 : Entity Bean

- **Extrait de l'API de l'entity manager :**

  ● **persist** : permet de persister une instance d'une entité
  ● **merge** : permet de rattacher ou activer un Bean Entité
  ● **remove** : permet de supprimer une instance d'une entité
  ● **find** : permet de rechercher une entité en fonction de sa clé primaire
  ● **refresh** : permet de réinitialiser l'état d'un objet par rapport à la base de données (annule toute modification)
  ● **createQuery** : méthode retournant un objet Query permettant d'effectuer des requêtes plus spécialisées

EJB 3 : Entity Bean

- **Persister un entity bean**

1-Il faut créer une nouvelle instance de l'entité à l'aide de l'opérateur new

2- affecter les valeurs grâce aux méthodes set

3- lier un entity bean à un autre lorsqu'il y a des associations,

4- et enfin, appeler la méthode EntityManager.persist()

- **Exemple :**

```
Customer customer = new Customer();
customer.setId(1234);
customer.setFirstname("Ahmed");
customer.setLastname("Alawi");
em.persist(customer);
```

EJB 3 : Entity Bean

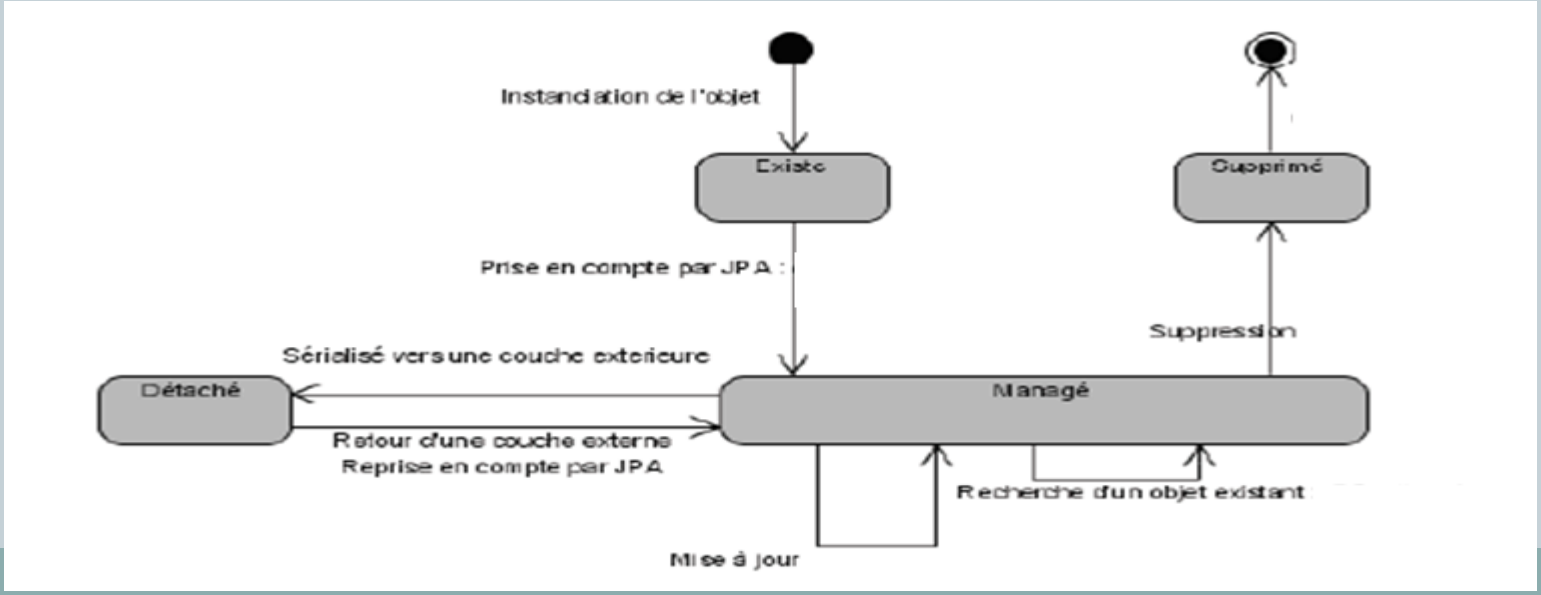
- **Rechercher un entity bean par son identifiant :**
- Pour rechercher un entity bean par son identifiant, on utilise la méthode **EntityManager.find()**. Cette méthode prend en paramètre la classe de l'entity bean ainsi que son identifiant et retourne un entity bean . Si celui-ci n'est pas trouvé, la méthode find retourne la valeur null.

- **Exemple:**

  customer = em.**find(Customer.class, 1234)**

EJB 3 : Entity Bean

- **Rattacher un entity bean**
- Lorsqu'un Entity Bean change de machine virtuelle (mécanisme de sérialisation) par exemple, il est détaché de son contexte et de sa persistance. Un bean entité est dit détaché lorsqu'il n'a aucun lien avec le gestionnaire d'entité: les modifications ne peuvent pas se propager sur la BD.
- Pour rattacher l'entity bean à l'entity manager et resynchroniser ses données avec la base, il suffit d'utiliser la méthode **EntityManager.merge()** qui prend en paramètre l'entity bean à rattacher.
-



EJB 3 : Entity Bean

- **Mettre à jour un entity bean:**

- La mise à jour d'un entity bean implique la méthode **find** de l'entity manager, ainsi que les setxx() des attributs :

- **Exemple :**

```
Customer customer = em.find(Customer.class, 1234);
customer.setFirstname("Ahmed");
customer.setLastname("Alawi");
```

EJB 3 : Entity Bean

- **Supprimer un entity bean :**

- Un entity bean peut être supprimé grâce à la méthode EntityManager. **remove()**. Cette dernière prend en paramètre l'entity bean, et entraîne la suppression des données en base.
- Une fois supprimé, l'entity bean se détache de l'entity manager et ne peut plus être manipulé par ce dernier.

- **Exemple :**

```
customer = em.find(Customer.class, 1234)
em.remove(customer);
```

EJB 3 : Entity Bean

- **Langage de requêtes :**

- JPA est une API de mapping objet-relationnel qui gère des objets et non des lignes et des colonnes. Cependant, pour conserver la possibilité d'effectuer des requêtes sur les entity beans, JPA utilise son propre langage : **JPQL**.
- JPQL, ou Java Persistence Query Language, est un langage de requêtes s'inspirant de la syntaxe de SQL.
- Sa particularité est de manipuler des objets dans sa syntaxe de requête et de retourner des objets en résultat.
- On manipule donc des objets dans une requête JPQL, puis le mécanisme de mapping transforme cette requête JPQL en langage compréhensible par une base de données relationnelle (en SQL).
- Le développeur manipule son modèle objet, et non une structure de données.

EJB 3 : Entity Bean

- **Effectuer des requêtes en JPQL :**

- Les requêtes JPQL se font à l'aide de l'interface **javax.persistence.Query**.
- Cette interface est utilisée pour contrôler l'exécution d'une requête JPQL.
- L'entity manager fabrique un objet **Query** à partir d'un ordre JQPL, et le retourne pour qu'il soit ensuite manipulé par le programme.

- **Exemple :**

```
Query query = em.createQuery("SELECT c FROM Customer c
WHERE c.login=:param");
query.setParameter("param", login);
Customer customer = (Customer) query.getSingleResult();
```

EJB 3 : Entity Bean

- **Exemple 2 : pour avoir la liste de tous les clients :**

```
Query query = em.createQuery("SELECT c FROM Customer c
ORDER BY c.lastname");
List<Customer> customers = query.getResultList();
```

EJB 3 : Entity Bean

- **Exemple 3 :**

```
Query query = em.createQuery("SELECT i FROM Item i WHERE
UPPER(i.name) LIKE :keyword
OR UPPER(i.product.name) LIKE :keyword
ORDER BY i.product.category.name, i.product.name");
query.setParameter("keyword", "%"+keyword.toUpperCase()+"%");
List<Item> items = query.getResultList();
```