Les Enterprise JavaBeans

Les Enterprise JavaBeans

Claude Duvallet

Université du Havre
UFR Sciences et Techniques
25 rue Philippe Lebon - BP 540
76058 LE HAVRE CEDEX
Claude.Duvallet@gmail.com
http://litis.univ-lehavre.fr/~duvallet/

Plan de la présentation

Les serveurs d'applications

Les serveurs d'applications

Contexte

- Vers une évolution des architectures informatiques :
  - Le modèle client-serveur.
  - Les systèmes à objets distribués.
  - ⇒ de nombreuses limitations : problèmes de déploiement, de maintenance, de migration, etc.
- Vers des architectures multi-niveaux avec les serveurs d'applications
  - Se focaliser sur la logique métier des applications en découplant celle-ci des aspects présentation et accès aux données.
- Fonctionalités offertes par les serveurs d'applications :
  - gestion transactionnelle,
  - administration et exploitation des composants,
  - sécurité et robustesse,
  - utilisation de composants COM ou EJB (Enterprise JavaBeans).

Les architectures transactionelles

- Issu des technologies "mainframe" d'IBM des années 70.
- Le terme « transaction » désignait le fait que plusieurs écrans pouvaient s'enchaîner avant qu'une modification ne soit effective.
- Cette notion de transaction a été ensuite reprise dans le monde des SGBD.
- La gestion transactionnelle distribuée et le modèle DTP (Distributed Transaction Processing) :
  - modèle proposé par l'X/Open, il offre deux interfaces :
    - TX : interface entre le Transaction Manager (TM) et l'applicatif. Une API qui permet d'invoquer le moniteur transactionnel.
    - XA : interface du Ressource Manager (RM) directement utilisée par le moniteur transactionnel.
  - Mise en œuvre de protocoles de validation à deux phases (Two-Phase Commit) : préparation de toutes les écritures puis écriture des résultats.

Le poste client

- Différents types de clients peuvent être développées dans les applications réparties :
  - les clients spécifiques pour le système d'exploitation Windows,
  - les clients Java : appplications autonomes ou applet,
  - les applications WEB : jsp ou autres.

Le poste client : applications windows

- Exécution du client développé grâce un IDE sur le poste local.
- Utilisation des composants de traitement via le réseau avec le middleware (DCOM ou Corba).
- Intégration avec les applications bureautiques via OLE.

⇒ Inconvénients :

- nécessité de disposer du protocole DCOM sur les clients,
- DCOM peut se trouver bloquer par les firewall,
- coût élevé de déploiement des applications Windows.

Le poste client : applications ou applet Java

- Similaires aux applications de windows.
- Fonctionne sur le middleware fourni par le serveur d'applications ou plus simplement sur RMI.
- Auto-déploiement de l'application à travers un navigateur pour les applet.

⇒ Inconvénients :

- nécessite de disposer de la bonne version de la machine virtuelle java et de ses bibliothèques de composants,
- pas d'intégration vers des outils bureautiques,
- lenteur des applications.

Le poste client : applications WEB

- Utilisation du protocole HTTP entre le client et le serveur.
- Traitement effectué au niveau du serveur WEB et donc pas de problème de déploiement.
- Application centralisée et donc plus facile à maintenir.
- Possibilité d'intégrer des applications natives et des documents statiques.
- Développement d'applications « universelles », indépendantes du poste client.

⇒ Inconvénients :

- richesse et ergonomie de l'IHM très faible,
- limites inhérentes au langage HTML (compensées en partie par le langage DHTML).

Le composant

- Un objet qui adhère à un modèle, il supporte un ensemble de méthodes normées qui lui permettent :
  - d'être exécuté dans un serveur d'applications conforme au modèle en supportant le mode transactionnel, la sécurité, la concurrence d'accès, l'indépendance à la localisation, etc.
  - d'être facilement intégré à un AGL en phase de développement grâce aux méthodes auto-descriptives d'introspection.
- Les méthodes normées se regroupent logiquement sous des interfaces permettant aux composants d'être indépendants de leur implantation.
- Il existe essentiellement deux modèles :
  - Le modèle COM (Component Object Model), issu de Microsoft et du monde OLE (ActiveX). Indépendant du langage.
  - Le modèle EJB (Enterprise JavaBean), spécification multi-éditeurs dont Sun et IBM sont à l'origine. Il repose sur le langage JAVA.

Les serveurs d'applications

- Il s'agit d'un cadre d'exécution pour des composants obéissant à un modèle (COM ou EJB), c'est-à-dire un ensemble de services permettant la bonne exécution des composants :
  - quatre services de base ;
    - l'accès aux composants,
    - l'optimisation de l'accès aux ressources locales et distantes,
    - la gestion transactionnelle,
    - la répartition de charge.
  - l'administration, l'exploitation ;
  - la sécurité ;
  - les passerelles vers l'existant ;
  - la persistance ;
  - les impressions.

Le standard Java EE (1/3)

- Java 2 Enterprise Edition :
  - Un ensemble de standards.
  - Définition des modes d'accès à un annuaire, à une base de données.
  - Définition du mode de dialogue entre les machines virtuelles Java.
  - Définition des interfaces qu'un composant doit présenter pour être réutilisable.
- Objectifs :
  - Rendre interopérables les composants développés dans le cadre des architectures distribuées.
  - Conception d'architectures multi-niveaux associant :
    - une base de données,
    - un serveur d'applications,
    - et des serveurs Web.

Le standard Java EE (2/3)

- Objectifs (suite) :
  - Un composant conforme aux services techniques Java EE doit pouvoir s'exécuter dans n'importe quel environnement estampillé "Java EE".
- La réalité :
  - De nombreux logiciels exploitent une partie des services de Java EE.
  - Peu parmi eux ont passé la totalité des tests de conformité Java EE.
  - Le ralliement d'un éditeur à la norme Java EE nécessite souvent de nombreux mois de travail.
  - $\Rightarrow$ Implantation partielle de la norme Java EE.
- Tous les logiciels s'appuyant sur un serveur d'applications est concerné par la norme Java EE.

Le standard Java EE (3/3)

- Java EE, Java et les EJB :
  - Java EE capitalise les technologies poussés par SUN : JAVA, JSP.
  - Les EJB représentent un modèle de composants. Ils représentent donc l'un des points clefs de la technologie Java EE.
- Qui pilote Java EE :
  - Les briques de base ont été élaborées par Sun.
  - Il est possible aux éditeurs de soumettre des mises à jour des spécifications suivant un mécanisme bien défini.
  - La force de Java EE tient à son adoption par les grands éditeurs tels que BEA et IBM.
- Version actuelle de Java EE : Java EE 6
- Documentation : `http://java.sun.com/javaee/6/docs/tutorial/doc/`

Introduction aux Enterprise JavaBeans

Introduction aux Enterprise JavaBeans

Architecture d'un système d'informations

Prendre en compte :

- l'accès aux données,
- le traitement des données,
- la présentation des données,
- la connectivité réseau,
- la construction des transactions,
- la sécurité de l'application.

L'architecture en couche (1/3)

- Elle permet de maîtriser la conception d'un SI, ainsi que son évolutivité en séparant les rôles sous forme de couches logicielles.
- Une architecture classique est une architecture à 3 couches :
  - la couche présentation,
  - la couche métier,
  - la couche données.
- La couche présentation :
  - elle contient des composants qui réalisent l'interface graphique de l'application et gèrent les interactions avec l'utilisateur.
  - elle peut être développée avec Motif, les MFC ou les JFC (swing), ou encore une applet Java, un ActiveX ou une page HTML statique ou dynamique.

L'architecture en couche (2/3)

- La couche métier :
  - Elle contient des composants qui réalisent les règles de gestion du processus métier.
  - Exemple : pour une application bancaire, les composants peuvent êtres des objets modélisant des comptes, des contrats... Les objets sont écrits en C, C++, Java...
- La couche données :
  - Elle est utilisée par la couche métier pour stocker l'état des objets métiers sur un support de persistance (SGBD relationnel ou objet...).
  - La couche donnée peut changer ou évoluer indépendamment des autres couches et réciproquement.

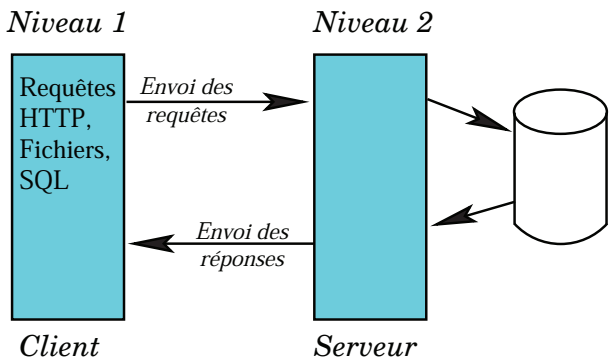
L'architecture en couche (3/3)

- Objectif principal de la séparation en couches :
  - assurer l'indépendance de la logique métier vis à vis des problèmes de visualisation et de stockage.
  - ⇒ Application plus modulaire, mieux structurée et plus facile à maintenir (évolutivité).
- On peut envisager des couches intermédiaires entre les couches précédentes :
  - une couche applicative interfaçant les couches présentation et métier (contrôleurs...),
  - une couche de services techniques interfaçant les couches métier et données.

L'architecture client/serveur (1/2)

- La séparation physique des couches est un autre problème.
- Dans une architecture à 2 niveaux, deux de ces couches sont regroupées et donc séparées de la troisième.
- Il s'agit notamment des applications client/serveur où les traitements et l'affichage sont effectués sur le poste client alors que les données sont stockées sur un serveur. Il s'agit de "clients lourds" au coût élevé de déploiement.
- Une autre méthode consiste à regrouper sur le serveur la logique métier (en partie ou en totalité) et le stockage des données.

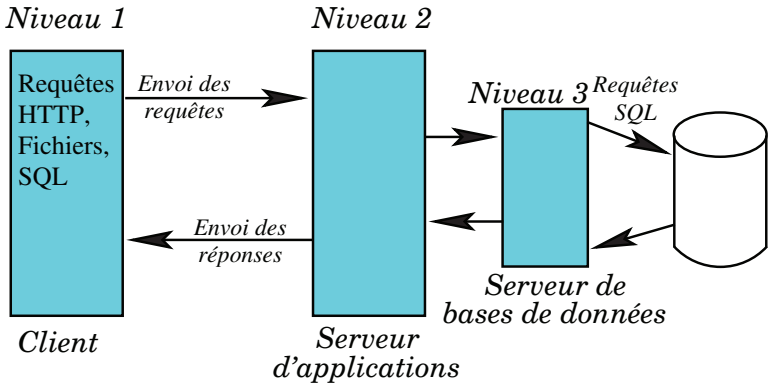
L'architecture client/serveur (2/2)



L'architecture 3 niveaux (1/3)

- Les 3 couches peuvent être complètement séparées physiquement :
  - On peut décomposer chacune de ces couches en sous-couches situées sur des machines ou des processus distincts.
  - Il s'agit alors d'une architecture n niveaux.
- C'est le cas d'une application Web :
  - la couche présentation est constituée de pages JSP et de servlets,
  - la logique métier est implémentée dans des beans ou objets hébergés sur un serveur d'application,
  - et la couche donnée consiste en une ou plusieurs bases de données.
- Intérêt de l'architecture 3 niveaux :
  - mieux supporter la montée en charge,
  - le partage des connexions aux ressources,
  - la facilité de déploiement, la sécurité.

L'architecture 3 niveaux (2/3)



L'architecture 3 niveaux (3/3)

- Le serveur devient le cœur du système.
- Toute la logique de l'application est implémentée par des composants hébergés sur ce serveur appelé serveur d'application.
- Il leur offre un environnement d'exécution, mais aussi un ensemble de services.
- La logique métier peut être assurer par plusieurs serveurs d'applications qui communiquent ensemble. Les objets métiers sont distribués.
- Par exemple, dans une application de e-commerce la logique nécessaire au traitement comptable est supportée par des composants situés sur un premier serveur alors que les composants qui gèrent la logique de gestion de stock se trouvent sur un deuxième serveur.

Les serveurs d'applications Java EE

- Malheureusement, ils n'étaient pas interopérables par manque de standard spécifiant le Middleware à utiliser, les services offerts, les interfaces pour y accéder...
- L'architecture Java EE est un standard proposé par Sun pour les serveurs d'applications Java.
- Au cœur de cette architecture, on trouve un Middleware standardisé, basé sur RMI/IIOP, et des objets métiers qui sont des objets Java distribués appelés EJB (Enterprise Java Bean).
- Un serveur d'application est un environnement complet, comprenant le conteneur d'EJB et le serveur Web.
- Pour assurer la portabilité, Sun a spécifié une suite de compatibilité consistant en environ 6000 tests que le serveur doit satisfaire pour obtenir la certification Sun.

L'architecture Java EE

- La technologie EJB fait partie d'une plate-forme plus large appelée Java EE.
- Cette plate-forme constitue une architecture pour le développement, le déploiement et l'exécution des applications distribuées.
- Ces applications requièrent des services techniques comme la gestion de transactions, la gestion de la sécurité, l'accès par les clients, l'accès aux bases de données.
- La plate-forme Java EE fournit ces services techniques.
- Le développeur peut se concentrer sur la logique métier au lieu de se disperser sur des problèmes techniques.

Les technologies Java EE (1/3)

Les technologies incluses dans Java EE, fournies sous formes d'API, permettent :

- La communication entre objets distribués avec RMI (Remote Method Invocation) et RMI/IIOP.
- La création d'objets distribués transactionnels avec des EJB (Enterprise JavaBeans).
- La recherche et la récupération dans un serveur de noms des références sur des objets distants avec JNDI (Java Naming and Directory Interface)
- L'accès aux bases de données avec JDBC (Java DataBase Connectivity)
- La gestion des transactions avec JTA (Java Transaction API) et JTS (Java Transaction Service).

Les technologies Java EE (2/3)

Elles permettent aussi :

- La communication asynchrone par messages entre objets distribués avec JMS (Java Message Service)
- La réalisation d'interfaces graphiques Web avec les pages JSP (JavaServer Pages) et les servlets.
- L'intégration des objets CORBA avec JavaIDL.
- L'envoi de courriers électroniques avec JavaMail.
- L'intégration des systèmes existants (CICS, progiciels) avec les connecteurs.
- La description du comportement des composants Java en XML.

Les technologies Java EE (3/3)

- La logique métier sera implémentée dans les EJB qui sont des composants transactionnels côté serveur et accessibles à distance par leurs clients.
- Les EJB s'exécutent dans des serveurs Java EE fonctionnant comme des serveurs intermédiaires dans un système client/serveur.
- Deux types de conteneurs sont utilisés : le conteneur Web et le conteneur EJB.
  - Le conteneur Web est un environnement d'exécution pour les pages JSP et les servlets qui constituent une passerelle entre l'interface utilisateur et les EJB implémentant la logique métier.
  - Le conteneur EJB est un environnement d'exécution pour les EJB. Le conteneur EJB héberge les composants métiers et leur fournit des services.

Le conteneur EJB

- Du côté serveur, l'accès aux différentes ressources s'avère complexe.
- Avec les EJB, la plupart des tâches se font par déclaration, évitant ainsi d'écrire du code spécifique pour gérer les transactions ou la persistance.
- Il s'agit de se concentrer sur le développement des composants en déléguant au conteneur EJB l'implantation des services techniques et la fourniture de ces services durant l'exécution.
- À un composant est associé un descripteur de déploiement, pour signifier par exemple qu'un bean soit persistant, sécurisé et accessible par plusieurs clients en même temps.
- Le serveur est responsable du traitement de ce descripteur et assure l'exécution de ces services.

Principales fonctionnalités d'un conteneur EJB (1/2)

Les principales fonctionnalités fournies par un conteneur d'EJB sont :

- La connectivité entre les clients et les EJB :
  - le conteneur gère les communications entre les clients et les EJB ;
  - après le déploiement d'un EJB dans un serveur d'applications, le client peut invoquer les méthodes de cet EJB comme si elles étaient situées dans la même machine virtuelle ;
  - les communications sont assurées par le Middleware sous-jacent de manière transparente.
- La gestion de la persistance :
  - les composants persistants peuvent choisir de déléguer leur persistance sur le support de persistance au conteneur.
- La gestion des transactions :
  - les composants persistants peuvent choisir de déléguer la gestion de leurs transactions au conteneur, qui implémente les mécanismes transactionnels nécessaires à la réalisation de la logique transactionnelle décrite par les beans.

Principales fonctionnalités d'un conteneur EJB (2/2)

- La gestion de la sécurité :
  - politiques de sécurité déclarées mais non codées par le développeur,
  - support de la sécurité basé sur l'API sécurité de Java,
  - méthodes liées à la sécurité implémentées par le conteneur,
  - utilisation des attributs de sécurité définis dans le descripteur de bean utilisé lors de la phase de déploiement.
- La gestion de la concurrence :
  - Les composants peuvent être invoqués par un seul client ou par plusieurs clients simultanément.
- La gestion du cycle de vie des composants :
  - création et la destruction des instances des composants.
- La gestion du pool de connexions :
  - connexion à une BD = coûteuse en termes de ressources,
  - nombre de connexions limité par le nombre de licences,
  - solution : le conteneur gère un pool de connexions.

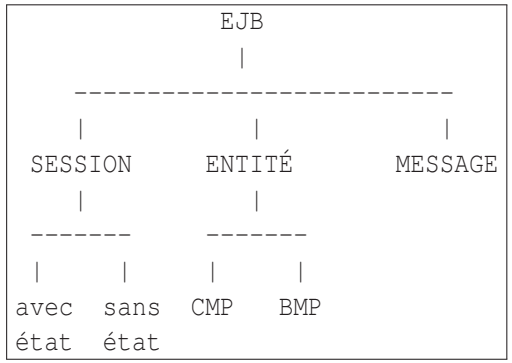
Les EJB (Enterprise JavaBeans) (1/3)

- Le développement de logiciel a vu apparaître dans les années 90 la notion de composant comme des morceaux de code standards, préconstruits, réutilisables et encapsulant de la logique métier.
- Les composants EJB sont conçus pour encapsuler la logique métier et éviter au développeur d'applications d'avoir à se préoccuper de tout ce qui a trait au système : transaction, sécurité, concurrence, communication, persistance, gestion d'erreurs...
- Un composant EJB est constitué d'une collection de classes Java et d'un fichier XML, fusionnés en une entité unique.
- Le conteneur prend en charge tout ce qui concerne le système. Cette séparation des tâches est le concept fondamental de cette technologie.

Les EJB (Enterprise JavaBeans) (2/3)

- Un composant EJB est conçu comme un ensemble réutilisable de logiques métiers et fonctionne avec tous les types de clients : servlets, JSP, application Java/RMI,...
- La spécification Enterprise JavaBeans 1.1 définit 2 types de beans : bean entité et bean session.
- La spécification Enterprise JavaBeans 2.0 introduit un troisième bean : le bean message. Il existe donc 3 types de beans :
  - les beans sessions,
  - les beans entités,
  - les beans messages.

Les EJB (Enterprise JavaBeans) (3/3)



Les beans sessions (1/2)

- Un bean session représente un processus. C'est une extension du processus du client dans le serveur d'applications Java EE.
- On distingue 2 types de beans sessions : les beans sessions sans état (stateless session) et les beans sessions avec état (statefull session).
- Un bean session ne peut avoir qu'un client à un moment donné.
- Pour un bean session sans état, plusieurs clients peuvent être associés au même bean successivement.
- Pour un bean session avec état, c'est le même client qui réalise toutes les invocations.
- Un bean session sans état représente un traitement fonctionnel, comme le calcul d'un itinéraire entre 2 points ou la demande d'un virement entre 2 comptes.

Les beans sessions (2/2)

- Un bean session avec état possède un état conversationnel, c'est à dire un état résultant des interactions avec un même client.
- Exemple de bean session avec état :
  - un panier sur un site de commerce électronique possédant deux attributs : un pour le nom du client et un pour les articles sélectionnés par ce client.
  - Ce bean peut posséder une méthode telle que `ajouterArticle()` que le client va invoquer pour ajouter un nouvel article dans le panier.
- L'état représenté par un bean session est privé et conversationnel. Le bean est accessible par un client unique.

Les beans entités (1/2)

- Un bean entité représente un objet métier persistant.
- Par exemple, un bean entité représente une commande, un article, un compte bancaire.
- Les attributs d'un bean entité peuvent être stockés dans une base de données ou tout autre support de persistance. Un bean entité peut être utilisé par plusieurs clients simultanément.
- La persistance d'un bean entité peut être gérée par le bean lui-même ou bien par le conteneur.
  - Dans le premier cas, les opérations de lecture et d'écriture sur le support de persistance doivent être codées dans le bean (par exemple du code SQL pour une BDR).
  - $\Rightarrow$ on parle de bean BMP (Bean Managed Persistence).

Les beans entités (2/2)

- Dans le second cas, la persistance est gérée par le conteneur et seules les fonctionnalités de recherches avancées doivent être codées dans le bean, les autres opérations de lecture et d'écriture sur le support de persistance sont effectués automatiquement par le conteneur.

  $\Rightarrow$ on parle de bean CMP (Container Managed Persistence).

- L'état représenté par un bean entité est partagé et transactionnel. Le bean constitue le point d'accés unique à ces données par différents clients.

Les beans messages

- Les beans sessions et entité sont des composants distribués invoqués par des clients de manière synchrone, c'est-à-dire par invocation de méthodes,
- Les beans messages sont des beans qui consomment des messages de manière asynchrone, par l'intermédiaire de Java Messaging Service (JMS).

Interfaces des Enterprise JavaBeans

- Les composants EJB possèdent des interfaces exposant leurs services disponibles aux clients.
- Les clients utilisent ces interfaces pour exécuter la logique encapsulée dans le bean.
- Il existent 2 sortes d'interfaces : les interfaces distances et les interfaces locales.
- Ces interfaces permettent de préciser quelles seront les méthodes métiers des Beans qui seront accessibles depuis l'intérieur ou l'extérieur d'un serveur d'un application.

Les objets distribués - base des EJB (1/3)

- Les objets distribués sont forts utiles puisqu'ils permettent de distribuer une application sur un réseau.
- Cependant, des exigences telles que les transactions et la sécurité deviennent indispensables dans des applications d'entreprise.
- Un objet distribué est un objet qui peut être appelé depuis un système distant, notamment :
  - depuis un client faisant partie du processus contenant l'objet (in-process),
  - depuis un client en dehors de ce processus (out-of-process),
  - ou bien depuis un client situé n'importe où sur le réseau.

Les objets distribués - base des EJB (2/3)

- Les modalités d'appel d'un objet distribué sont :
  1. Le client appelle un stub, qui est un objet proxy côté client.
     - Le stub masque au client la communication réseau.
     - Le stub envoie des appels sur le réseau au moyen de sockets, en manipulant les paramètres comme il convient dans leur représentation réseau.
  2. Le stub appelle un skeleton sur le réseau, qui est un objet proxy côté serveur.
     - Le skeleton masque la communication effectuée au niveau réseau à destination de l'objet distribué.
     - Le skeleton est en mesure de recevoir les appels sur une socket et de manipuler les paramètres dans leur représentation réseau.
     - Le skeleton délègue l'appel à l'objet distribué.
  3. L'objet distribué accomplit sa tâche et renvoie le contrôle au skeleton, qui le renvoie à son tour au stub pour ce que ce dernier le renvoie au client.

Les objets distribués - base des EJB (3/3)

- Le stub et l'objet distribué implémentent la même interface, appelée interface distante.
  - Un client qui appelle une méthode de l'objet distribué appelle en fait une méthode du stub.
  - On parle de transparence locale/distante.
- Différentes technologies permettent d'utiliser des objets distribués telles que CORBA de OMG, DCOM de Microsoft et Java RMI-IIOP de Sun.

Les Middlewares

- Les objets distribués nécessitent une infrastructure technique permettant d'assurer la communication entre ces objets.
- Cette infrastructure doit fournir aux objets un accès aux services transversaux comme les services de nommage, de gestion des transactions, de gestion de persistance, de gestion de sécurité...
- de manière transparente pour le développeur, c'est à dire sans qu'il ait besoin de coder des mécanismes pour assurer ces services.

$\Rightarrow$ Ce type d'infrastructure est appelée Middleware.

- D'autres services sont nécessaires :
  - optimisation des accès aux ressources (pool de connexions...),
  - mécanisme d'activation/désactivation des objets,
  - mécanisme de répartition de charge et de tolérance aux pannes.

Middleware explicite

- Dans une programmation d'objets distribués classique, telle que la programmation CORBA, il existe un Middleware standard utilisé dans le code (qui appelle l'API du Middleware).
- On obtient ainsi des transactions en écrivant sur une API de transaction.
- Ce Middleware est dit explicite, puisqu'il faut écrire sur une API pour l'obtenir.
- Par exemple, un objet distribué de compte bancaire, capable de transférer des fonds d'un compte à un autre utiliserait le code suivant :

```
transfert(Compte a, Compte b, long montant){
// 1: appelle l'API Middleware pour effectuer un contrôle de sécurité
// 2: appelle l'API Middleware pour lancer une transaction
// 3: appelle l'API Middleware pour charge les lignes d'une base de données
// 4: extrait le solde d'un compte et l'ajoute à un l'autre
// 5: appelle l'API Middleware pour enregistrer les lignes dans base de données
// 6: appelle l'API Middleware pour clore la transaction}
```

- L'obtention des services Middleware et la logique métier sont étroitement liés.
- Cela se traduit par des difficultés d'écriture, de maintenance et de dépendance au Middleware.

Middleware implicite (1/2)

- L'écriture de l'objet distribué contient essentiellement la logique métier :

```
transfert(Compte a, Compte b, long montant){
// 1: extrait le solde d'un compte et l'ajoute à un l'autre
}
```

- La déclaration des services Middleware nécessaires (transactions, persistance, sécurité) à l'objet distribué se fait dans un fichier de description distinct.
- Un outil associé au Middleware permet de générer un objet "intercepteur d'appels".
- Cet objet intercepte les appels clients, exécute le Middleware nécessaire (transaction, sécurité, persistance) puis délègue ces appels à l'objet distribué.

Middleware implicite (2/2)

- Un Middleware implicite (ou déclaratif) apporte une facilité d'écriture, de maintenance et d'indépendance du support.
- Dans l'architecture Java EE, c'est le conteneur EJB qui intercepte les appels pour exécuter automatiquement les services du Middleware et délègue à l'instance du bean.
  - Le conteneur ≈ couche d'indirection entre le code client et le bean, utilisation d'un intercepteur d'appels nommé objet EJB (`EJBObject`).
  - Ces objets sont dépendants du conteneur utilisé. Le client traite donc avec l'objet EJB et non directement avec le bean.
  - Il lui permet d'obtenir une référence à cet objet EJB.
  - Le client fait appel à une fabrique d'objets EJB. Celle-ci est chargée d'instancier et de détruire les objets EJB. Cette fabrique est appelée objet d'accueil ou `EJBHome`.
- Le composant ou bean EJB est donc un composant côté serveur entièrement géré et distribué par le conteneur.

Exemple d'application Java EE

Un exemple type d'application basée sur l'architecture Java EE est celui d'une application de e-commerce :

- Le client se connecte sur le site Web d'un magasin, consulte un catalogue des articles disponibles, en choisit certains qu'il met dans un panier électronique et règle ses achats.
- Du côté du serveur Web, les pages JSP et servlets utilisent :
  - des beans entités qui correspondent aux objets métiers comme les articles, clients, commandes, factures et des beans sessions avec état représentant les paniers des clients.
  - des beans sessions sans état qui peuvent implémenter des traitements comme la navigation dans un catalogue ou la demande d'un devis pour le contenu d'un panier.

Écriture d'un premier Bean

Écriture d'un premier Bean

Écriture d'un premier Bean

- Le développement d'un bean est relativement complexe et nécessite de bien comprendre les différents éléments mis en œuvre ainsi que les différentes étapes du processus de création du composant.
- Le composant développé ici sera un bean session sans état (stateless session bean) qui renvoie la chaîne de caractères "Hello, World".

Développement d'un composant EJB

- Une application Java EE commence par la création de composants Java EE.
- Ces composants sont ensuite regroupés en modules auxquels on associe des descripteurs de déploiement.
- Ces modules Java EE peuvent ensuite être déployés comme application à part entière ou être assemblés avec un descripteur de déploiement Java EE et être déployés en tant qu'application Java EE.

Création de composants applicatifs

- Durant cette phase, nous modélisons les règles métiers sous la forme de composants applicatifs.
- Ces composants applicatifs seront groupés selon leur type (Web, EJB) avec un descripteur de déploiement pour donner un module Java EE.
- Ces modules Java EE composeront l'application Java EE.

Note : un module Java EE peut être déployé seul et être considéré comme une application Java EE valide.

Assemblage de l'application

- Une application Java EE peut être constitué d'un ou plusieurs module Java EE et un descripteur d'application Java EE.
- L'application Java EE est packagée dans un fichier ayant l'extension .ear (Enterprise Archive).

Note : Un fichier EAR est un ficher JAR sauf qu'il contient un fichier application.xml décrivant l'application.

Déploiement

- Le déploiement d'applications consiste à installer et à personnaliser des modules empaquetés sur une plate-forme Java EE.
- Ce processus se compose de deux étapes :
  - Installation : On installe l'application sur le serveur Java EE.
  - Configuration : On paramètre l'application pour qu'elle s'intègre à l'infrastructure sur laquelle elle vient d'être installée (mot de passe pour la base de données, factory de connexions, création des utilisateurs, définition des droits...).

Écriture d'un premier Bean

De manière plus détaillée :

1. Écriture des différents fichiers .java qui composent le bean :
   - les interfaces du bean,
   - les interfaces d'accueil,
   - le fichier de classe du bean et les classes auxiliaires nécessaires
2. Écriture du descripteur de déploiement
3. Compilation en fichiers .class
4. Création du fichier EJB-jar contenant le descripteur de déploiement et les fichiers .class à l'aide de l'utilitaire jar
5. Déploiement du fichier EJB-jar dans le conteneur (en général à l'aide d'un outil approprié)
6. Configuration du conteneur à l'aide d'un fichier de configuration
7. Lancement du serveur (chargement du fichier EJB-jar)
8. Écriture d'un fichier client java, compilation et exécution pour tester les fonctionnalités offertes par l'API du bean.

Spécification des interfaces de l'EJB

- Les beans sont des composants protégés, dans le sens où le client ne peut avoir accès direct à un bean.
- L'accès se fait toujours par l'intermédiaire d'interfaces rendues disponibles par le conteneur. Il existe 2 sortes d'interfaces :
  - les interfaces métiers (méthodes spécifiques à l'application),
  - les interfaces de gestion du cycle de vie (Home)(méthodes génériques).
- De plus, chacune de ces catégories d'interfaces se décline en interface locale et interface distante.

Interface métier (Object) distante (1/2)

- Les interfaces métiers définissent les services que le bean rend au client.
- L'interface distante définit les méthodes métiers qui sont accessibles par des composants situés dans la même machine virtuelle que le bean ou dans une autre JVM.

```
package hello;

import java.rmi.*;
import javax.ejb.*;

public interface Hello extends EJBObject{
  public String hello() throws RemoteException;
}
```


Interface métier (Object) distante (2/2)

- Cette interface étend `EJBObject`. Par conséquent, l'objet EJB génèré par le conteneur et qui implémente cette interface distante contient toutes les méthodes définies par l'interface `EJBObject`.
- La méthode métier ici est `hello()`. Cette méthode doit être implémentée dans la classe du bean.
- L'interface distante étant une interface RMI-IIOP (étend l'interface Remote), elle doit lever une exception distante pour cette méthode.
- Il s'agit là de la seule différence entre la méthode `hello()` de l'interface distante et celle de la méthode `hello()` du bean.

Interface métier locale (1/2)

- L'interface métier locale définit les méthodes métiers accessibles uniquement par des composants (clients locaux) situés dans la même JVM que le bean.
- Dans l'exemple, l'interface définit la méthode métier `hello()`.

```
package hello;

public interface HelloLocal extends EJBLocalObject{
  public String hello();
}
```

- Elle définit donc les mêmes méthodes que l'interface distante. Il n'y a pas de `RemoteException`.

Interface métier locale (2/2)

- L'interface locale a été ajoutée dans la version 2.0 pour améliorer les performances en évitant de passer par le Middleware réseau pour deux composants se trouvant dans le même conteneur.
- L'interface locale trouve son intérêt par rapport à des classes java dans le fait que le bean doit rester isolé de ses clients.
- Cela permet au conteneur, que le client soit local ou distant, de prendre en charge différents services comme la gestion de la sécurité ou la gestion des transactions.
- Il est toujours possible d'utiliser en local l'interface distante !

Interface Home distante (1/2)

- Les interfaces Home gère le cycle de vie du bean, c'est-à-dire définissant la manière dont le bean est créé, recherché et supprimé.
- Elles peuvent être également locale ou distante.
- L'interface Home possède des méthodes permettant de créer et de détruire les objets EJB.
- L'implantation de l'interface Home est l'objet Home, génère automatiquement par les outils du conteneur.

```
package hello;

import java.rmi.*;
import javax.ejb.*;

public interface HelloHome extends EJBHome{
  Hello create() throws RemoteException, CreateException
}
```


Interface Home distante (2/2)

- La méthode `create()` est une méthode utilisée par les clients pour obtenir une référence à un objet EJB. Elle permet également d'initialiser un bean.
- La méthode `create()` lève des exceptions `RemoteException` et `CreateException`. La première est requise car l'objet Home implémente est un objet distant.
- L'interface Home étend l'interface `EJBHome`.

Interface Home locale

- L'interface locale définit des méthodes de gestion du cycle de vie (création, suppression, recherche) accessibles uniquement par des clients situés dans la même JVM.

```
package hello;

public interface HelloLocalHome extends EJBLocalHome{
  HelloLocal create() throws CreateException;
}
```


Interface Home locale (2/2)

- Techniquement, les différences entre l'interface distante et l'interface locale sont :
  - l'interface locale étend `EJBLocalHome`. `EJBLocalHome` n'étend pas l'interface Remote. Par conséquent, l'implantation générée n'est pas un objet distant.
  - l'interface locale ne lève pas d'exception `RemoteException`.
- Les interfaces métiers et Home, locales ou distantes, sont communes aux beans sessions et entité.
- Les beans entités possèdent, en outre, une classe représentant la clé qui permet d'identifier le bean de manière unique dans la base de données.
- Cette clé est appelée clé primaire et la classe est appelée classe de clé primaire.

La classe du bean (1/3)

- La classe d'implantation d'un bean session implémente l'interface `SessionBean`.
- Celle d'un bean entité implémente l'interface `EntityBean`.
- Celle d'un bean message implémente l'interface `MessageDrivenBean`.
- Ces 3 interfaces héritent de l'interface `EnterpriseBean`.
- Un bean session (ou entité) peut supporter l'une ou les deux interfaces métiers et l'une ou les deux interfaces Home selon que le bean sera accessible à distance ou localement.
- La classe d'implantation ne doit pas déclarer qu'elle implémente ces interfaces, mais doit avoir des méthodes qui correspondent aux méthodes des interfaces métiers et Home ! !

La classe du bean (2/3)

- Dans l'exemple, la classe d'implantation s'appelle HelloBean.
- Elle implémente la méthode métier `hello()`, définie dans les interfaces métiers locales et distantes ainsi que la méthode `ejbCreate()` qui correspond à la méthode `create()` des interfaces Home locale et distante.
- Les méthodes `ejbRemove()`, `ejbPassivate()`, `ejbActivate()` participent à la gestion du cycle de vie du bean.
- Pour le bean hello, il n'y a rien à faire de spécial. Pour des beans plus complexes, ces méthodes permettent de gérer les ressources manipulées par le bean.

La classe du bean (3/3)

```
package hello;

import java.rmi.*;
import javax.ejb.*;

public class HelloBean implements SessionBean{
  public String hello(){
    return "Hello, World";
}

  public void ejbCreate(){}
  public void ejbRemove(){}
  public void ejbActivate(){}
  public void ejbPassivate(){}
  public void setSessionContext(SessionContext ctx){}
}
```


Déploiement du Bean

- Une fois le bean complètement défini, il faut le déployer dans un serveur d'applications Java EE.
- Pour cela, il faut assembler les éléments du bean dans un fichier JAR qui va contenir les éléments suivants :
  - les interfaces du bean,
  - la classe d'implantation et toutes les classes auxiliaires nécessaires,
  - le descripteur de déploiement.
- Le descripteur de déploiement spécifie de manière déclarative les attributs des beans, plutôt que de programmer cette fonctionnalité dans le bean lui-même.
- C'est un document XML nommé ejb-jar.xml.

Le descripteur de déploiement (ejb-jar.xml)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
 "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
  <description>Descripteur de déploiement de HelloWorld</description>
  <display-name>HelloWorld</display-name>
  <enterprise-beans>
    <session>
      <description>Descripteur de déploiement de HelloWorld</description>
      <display-name>Hello</display-name>
      <ejb-name>HelloEJB</ejb-name>
      <home>hello.HelloHome</home>
      <remote>hello.Hello</remote>
      <ejb-class>hello.HelloBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
    <method>
      <ejb-name>HelloEJB</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```


Le déploiement (1/2)

- Il est assuré par le fichier EJB-jar (archive contenant tous les fichiers du Bean), ici Hello.jar.
- Il peut être créé manuellement par `jar cvf Hello.jar *` dans le répertoire contenant les fichiers .class et le fichier XML.
- Sa structure est :

```
META-INF/MANIFEST.MF
META-INF/ejb-jar.xml
hello/HelloBean.class
hello/HelloHome.class
hello/HelloLocalHome.class
hello/HelloLocal.class
hello/Hello.class
```


Le déploiement (2/2)

- Si les classes sont paquetagées (par exemple `package hello;`) il faut disposer les fichiers `.class` dans un sous-répertoire `hello` de même niveau que `META-INF`.
- Le fichier MANIFEST.MF répertorie les fichiers contenus dans le fichier EJB-jar. Il est normalement généré automatiquement par l'utilitaire jar.
- Le déploiement effectif dépend du conteneur d'EJB. Lors de ce déploiement, le fichier EJB-jar est vérifié. Puis, un objet EJB est généré ainsi que les stubs et skeletons RMI nécessaires.
- Avec le serveur JBoss, il suffit de copier le fichier EJB-Jar dans le répertoire `$JBOSS_HOME$/server/default/deploy`.
- Certains outils de développement tels que NetBeans ou Eclipse pourront faire le déploiement de façon automatique.

Le client

- Une fois le bean déployé, il devient accessible aux clients.
- Le client doit commencer par chercher l'interface Home d'un bean, ce qui lui permet ensuite de récupérer une référence à l'interface métier.

```
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import java.util.Properties;
import hello.*;

public class HelloClient{
  public static void main(String[] args) throws Exception {
    try{
      Properties props = System.getProperties(); // Récupération des varaibles d'environnement
      props.put(Context.INITIAL_CONTEXT_FACTORY, "org.jnp.interfaces.NamingContextFactory");
      props.put(Context.URL_PKG_PREFIXES, "org.jboss.naming:org.jnp.interfaces");
      props.put(Context.PROVIDER_URL, "jnp://localhost:1099");  // Configuration de l'adresse pour le JNDI
      Context ctx = new InitialContext();
      Object ref = ctx.lookup("HelloEJB"); // Récupération de la référence dans le JNDI
      HelloHome home = (HelloHome)PortableRemoteObject.narrow(ref,HelloHome.class);
      Hello hello = home.create();         // Demande de création d'un objet sur le serveur d'application
      System.out.println(hello.hello());   // Appel de la méthode méier
      hello.remove();                      // Relachement de l'objet sur le serveur d'application
    }catch(Exception e){
      e.printStackTrace();
      System.exit(1);
    }
  }
}
```

Les beans sessions

Les beans sessions

Introduction

- Un bean session est un composant fournissant un processus métier.
- Il implémente la logique métier, les règles métiers comme le traitement d'une commande, le traitement de demande de virement bancaire...
- C'est la prolongation du processus client dans un serveur d'applications Java EE.
- Le client peut être une application Java autonome, une applet, une servlet ou un autre bean session ou entité. Il sollicite les services du bean via son interface métier.
- Deux types de beans sessions : avec état ou sans état.

Bean session **sans** état

- Un bean session sans état est une collection de services, chacun représenté par une méthode.
- Le bean ne préserve pas d'état d'un appel à l'autre.
- Quand on invoque une méthode sur un bean sans état, il exécute la méthode et renvoie le résultat ; il ne se préoccupe pas de savoir quelle méthode a été invoquée avant ou sera invoquée après.
- Le client garde une référence sur l'objet EJB mais le bean est libre de servir des invocations de méthodes en provenance d'autres objets EJB.
- Le client voit le bean via l'objet EJB qui a une durée de vie liée à la session de l'utilisateur : instance de la classe d'implantation peut exister avant et après la création de l'objet EJB.
- Exemples classiques de beans sessions sans état : services de calculs, de recherche d'information dans une BD...

Bean session **avec** état

- Un bean session avec état est une extension du client dans le serveur d'application.
- Il effectue des opérations pour le compte d'un client et maintient un état propre complémentaire de l'état du client.
- Cet état qui est matérialisé par diverses variables d'instances du bean est conservé entre les différents appels effectués par le client lors de sa conversation avec le bean.
- On l'appelle état conversationnel.
- Il peut être lu ou modifié par les méthodes du bean.
- L'exemple classique est celui du panier électronique.

Classe d'implantation d'un bean session (1/2)

- La classe d'implantation d'un bean session implémente l'interface `SessionBean`. Cette interface hérite de l'interface `EnterpriseBean`.
- Un bean session peut supporter l'une ou les deux interfaces métier et l'une ou les deux interfaces Home selon que le bean sera accessible à distance et/ou localement.
- La classe d'implantation ne doit pas déclarer qu'elle implémente ces interfaces, mais doit avoir des méthodes qui correspondent aux méthodes des interfaces métiers et Home !
- Cette classe doit être publique, non abstraite et non finale.
- Elle doit avoir un constructeur public sans argument et ne doit pas redéfinir la méthode `finalize()`.

Classe d'implantation d'un bean session (2/2)

- Un bean session sans état doit implémenter la méthode de création `ejbCreate()` sans argument alors qu'un bean session avec état doit implémenter une méthode de création `ejbCreateXxxx()`.
- Les méthodes de création doivent être publiques, non statiques, non finales.
- Une méthode de création lève l'exception `CreateException` si un paramètre d'entrée n'est pas valide.
- Dans le cas d'une interface distante, les arguments et la valeur de retour doivent être compatibles avec RMI.
- Les méthodes de l'interface `SessionBean` définies par la classe d'implantation seront invoquées par le conteneur pour informer le bean de son parcours dans son cycle de vie.

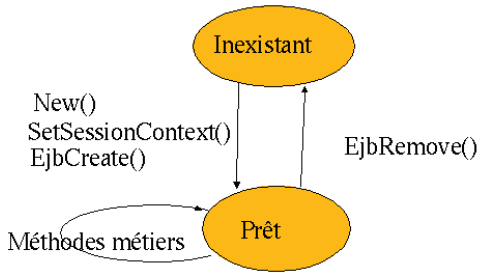
Cycle de vie d'un bean

- Un serveur d'applications est susceptible de monter en charge, c'est à dire de servir un grand nombre de clients avec des ressources limitées.
- Il chargera en mémoire un nombre limité de beans.
- Certains seront désactivés (passivation) et swappés et inversement (activation).
- Le serveur informe le bean au cas où il doit le désactiver pour que celui-ci libère les ressources acquises avant de passer à l'état passif.
- Idem pour la réactivation pour restaurer les ressources. C'est le rôle des méthodes `ejbActivate()` et `ejbPassivate()` qui sont invoquées par le conteneur.
- Les beans sessions ont donc un cycle de vie différents selon le cas (sans état et avec état).

Cycle de vie d'un bean session sans état (1/2)

- Un même bean peut être utilisé par le conteneur pour servir des requêtes provenant de différents clients, une requête à la fois, les beans n'étant pas multithreadés.
- Le conteneur crée un certain nombre de beans qu'il met dans un pool et les répartis sur les différentes requêtes.
- Il peut créer éventuellement de nouvelles instances si la charge augmente. Ou en supprimer pour les raisons inverses.
- On n'a pas de notion d'activation ou de passivation : les méthodes `ejbPassivate()` et `ejbActivate()` doivent être vides.
- Un bean session sans état n'étant jamais passivé, son cycle de vie se réduit à 2 états : inexistant et prêt pour recevoir des invocations.

Cycle de vie d'un bean session sans état (2/2)



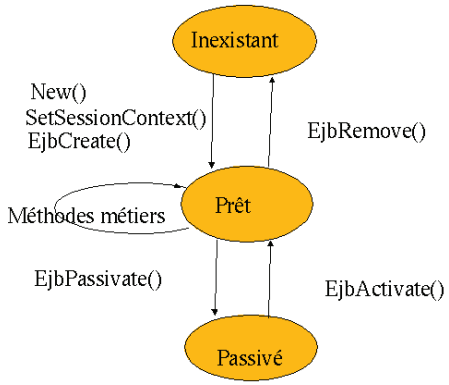
L'appel à la méthode `create()` fournit une référence sur un objet EJB.

Lors de l'appel à une méthode métier, une instance est sélectionnée pour servir la requête.

Cycle de vie d'un bean session avec état (1/4)

- Un bean session avec état est associé à un seul client. Il est activé ou passivé par le conteneur.
- Lors d'une passivation, le conteneur sérialise le bean pour être sauvegardé.
  - En fait seules les variables d'instances sérialisables et les références vers les objets du conteneur (comme `SessionContext`) sont sauvegardés.
  - Les autres variables d'instances sont à la charge du programme (fermeture d'une connexion puis restauration lors de l'activation).
- `ejbActivate()` est presque identique à `ejbCreate()` : acquisition de connexions aux ressources nécessaires.
- `ejbPassivate()` est presque identique à `ejbRemove()` : libération des connexions aux ressources.

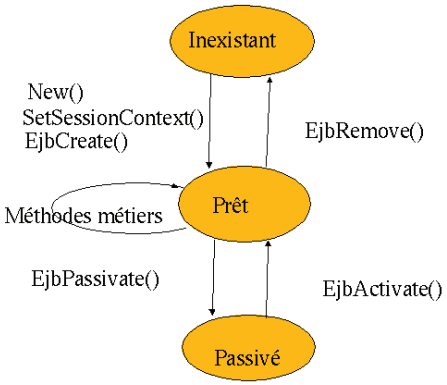
Cycle de vie d'un bean session avec état (2/4)



Initialisation du cycle de vie :

- Le client initie le cycle de vie en invoquant `create()`.
- Le conteneur crée une instance du bean et invoque les méthodes `setSessionContext()` et `ejbCreate()`.
- Le bean est alors prêt à recevoir les appels de méthodes métiers du client.

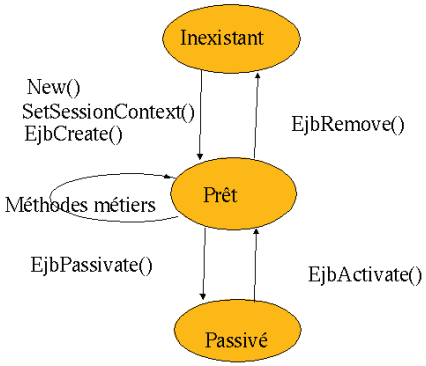
Cycle de vie d'un bean session avec état (3/4)



Désactiver le bean :

- Quand le bean est dans l'état prêt, le conteneur peut désactiver ou passiver le bean en le supprimant de la mémoire vers un stockage secondaire (par exemple, le bean le moins utilisé).
- Le conteneur invoque `ejbPassivate()` juste avant de passiver le bean.
- Si le client invoque une méthode du bean dans l'état passif, le conteneur l'active en le rechargeant puis invoque la méthode `ejbActivate()`.

Cycle de vie d'un bean session avec état (4/4)



Fin du cycle de vie :

- Le cycle de vie se termine quand le client invoque `remove()`.
- Le conteneur invoque alors `ejbRemove()` et dé-référence le bean qui pourra être supprimé par le garbage collector.

Contrôle du cycle de vie :

- Le développeur ne contrôle le cycle de vie que sur les méthodes `createXxxx()`, `remove()` et les méthodes métiers.
- Toutes les autres méthodes sont invoquées par le conteneur.

Développement d'un bean session sans état

- Un bean session implémente des méthodes métiers.
- Certaines de ces méthodes sont visibles de clients distants en étant déclarées dans une interface métier distante et utilisées par les clients.
- Une interface Home distante complète le dispositif.
- Pour le bean Calc, cela donne la classe d'implantation CalcBean, l'interface métier distante Calc et l'interface Home distante CalcHome.
- On peut compléter par des interfaces locales (CalcLocalHome et CalcLocal).

L'interface métier distante

- L'interface déclare les méthodes `add()` et `mult()`.

```
import javax.ejb.*;
import java.util.*;
import java.rmi.*;

public interface Calc extends EJBObject {
  public double add(double val1, double val2) throws RemoteException;
  public double mult(double val1, double val2) throws RemoteException;
}
```

- Les méthodes doivent être conformes aux conventions RMI :
  - paramètres compatibles avec RMI/IIOP,
  - méthode distante générant une exception `RemoteException`,
  - et modificateur non static ou final.
- Calc hérite de l'interface `EJBObject`.

L'interface Home distante

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

public interface CalcHome extends EJBHome {
  public Calc create() throws CreateException, RemoteException;
}
```


La classe d'implantation

```
import javax.ejb.*;

public class CalcBean implements SessionBean {

  SessionContext sessionContext;

  public void ejbCreate() { }
  public void ejbRemove() { }
  public void ejbActivate() { }
  public void ejbPassivate() { }

  public void setSessionContext(SessionContext sessionContext) {
    this.sessionContext = sessionContext;
}

  public double add(double val1, double val2) {
    return val1 + val2;
}
  public double mult(double val1, double val2) {
    return val1 * val2;
}
}
```


Le client

```
import javax.naming.*;
import javax.rmi.*;
import javax.ejb.*;
import java.util.Properties;

public class CalcClient {

  public static void main(String[] args) {
    try {
      Properties props = System.getProperties();
      props.put(Context.INITIAL_CONTEXT_FACTORY, "org.jnp.interfaces.NamingContextFactory");
      props.put(Context.URL_PKG_PREFIXES, "org.jboss.naming:org.jnp.interfaces");
      props.put(Context.PROVIDER_URL, "jnp://localhost:1099");
      Context ctx = new InitialContext();
      Object ref = ctx.lookup("CalcEJB");
      CalcHome home = (CalcHome) PortableRemoteObject.narrow(ref, CalcHome.class);
      Calc calc = home.create();

      double somme = 0;
      somme = calc.add(5.643, 8.2921);
      System.out.println(somme);

      calc.remove();
} catch (Exception e) { e.printStackTrace(); }
}
}
```


Développement d'un bean session avec état

- C'est similaire au cas de bean session sans état.
- L'exemple est celui du panier électronique.
- Il représente à un moment donné l'ensemble des articles que le client envisage d'acheter.
- Il y ajoute ou supprime des articles et en demande la liste, puis achète le contenu du panier.
- Chaque client doit posséder son propre panier.
- Une fois que l'achat est fait, le panier est supprimé.
- Un bean session avec état, dit conversationnel, permet d'implémenter un panier.

L'interface métier distante

```
import javax.ejb.*;
import java.util.*;
import java.rmi.*;

public interface Panier extends EJBObject {
  public void ajouterArticle(int idArticle) throws RemoteException;
  public void supprimerArticle(int idArticle) throws RemoteException;
  public Vector listerArticles() throws RemoteException;
  public void setNom(String nomClient) throws RemoteException;
  public String getNom() throws RemoteException;
}
```

Les méthodes doivent être conformes aux conventions RMI :

- paramètres compatibles avec RMI/IIOP,
- méthode distante générant une exception `RemoteException`,
- et modificateur non static ou final.

L'interface Home distante

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

public interface PanierHome extends EJBHome {
  public Panier create()
    throws RemoteException, CreateException;
  public Panier createAvecNom(String nomClient)
    throws RemoteException, CreateException;
}
```

- Lorsque le client invoque une méthode `createXxxx()` sur une interface Home, un bean et son objet EJB sont créés et associés au client.
- Le conteneur invoque alors la méthode `ejbCreateXxxx()` correspondante sur le bean pour qu'il s'initialise.

La classe d'implantation (1/3)

- Elle définit deux variables d'instance : le nom du client et le vecteur d'articles.
- L'initialisation de ces variables se fait lorsque le conteneur invoque la méthode `ejbCreateXxxx()` suite à l'invocation de la méthode `createXxxx()` de la part du client.
- Lorsque le client utilise la méthode `create()` sans argument, le conteneur invoque la méthode `ejbCreate()` sans argument sur le bean.
- Lorsque le client invoque la méthode `createAvecNom()`, le conteneur invoque la méthode ejbCreateAvecNom() sur le bean.
- Cette classe fournit l'implantation des méthodes métiers `ajouterArticle()`, `supprimerArticle()`, `listerArticle()`, `getNomClient()` et `setNomClient()`.
- Elle ne fait aucun traitement de passivation et d'activation : les données de la classe étant de type `String` et `Vector`, elles seront automatiquement sérialisées lors d'une passivation et désérialisées lors d'une activation.

La classe d'implantation (2/3)

```
import javax.ejb.*;
import java.rmi.*;
import javax.util.*;

public class PanierBean implements SessionBean {
  SessionContext sessionContext;
  Vector articles;
  String nomClient;

  public void ejbCreate() throws CreateException {
    articles = new Vector();
}

  public void ejbCreateAvecNom(String nomClient) throws CreateException {
    this.nomClient = nomClient;
    articles = new Vector();
}

  public void ejbRemove() { }
  public void ejbActivate() { }
  public void ejbPassivate() { }

  public void setSessionContext(SessionContext sessionContext) {
    this.sessionContext = sessionContext;
}
```

La classe d'implantation (3/3)

```
  public void ajouterArticle(int idArticle) {
    articles.add(new Integer(idArticle));
}

  public void supprimerArticle(int idArticle){
    articles.remove(new Integer(idArticle));
}

  public Vector listeArticle(){
    return articles;
}

  public void setNom(String nomClient) {
    this.nomClient = nomClient;
}

  public String getNom() {
    return nomClient;
}
}
```


Le client

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

public class PanierClient {
  public static void main(String[] args) throws Exception {
    try {
      Properties props = System.getProperties();
      props.put(Context.INITIAL_CONTEXT_FACTORY, "org.jnp.interfaces.NamingContextFactory");
      props.put(Context.URL_PKG_PREFIXES, "org.jboss.naming:org.jnp.interfaces");
      props.put(Context.PROVIDER_URL, "jnp://localhost:1099");
      Context ctx = new InitialContext();
      Object ref = ctx.lookup("PanierEJB");
      PanierHome panierHome = (PanierHome) PortableRemoteObject.narrow(ref, PanierHome.class);
      Panier monPanier = panierHome.createAvecNom("Dupont");
      monPanier.ajouterArticle(65);
      monPanier.ajouterArticle(53);
      monPanier.ajouterArticle(23);
      monPanier.ajouterArticle(18);
      monPanier.supprimerArticle(65);

      Vector mesArticles = monPanier.listerArticles();
      System.out.println ("Il y a "+mesArticles.size()+" article(s) dans le panier !");
      Enumeration e = mesArticles.elements();
      while (e.hasMoreElements()) {
      System.out.println((Integer)e.nextElement());
}
    monPanier.remove();
} catch (Exception e) { e.printStackTrace();}
}
}
```


Collaboration entre beans

- Une application basée sur les EJB met en jeu un certain nombre de beans qui vont collaborer ensemble.
- Chaque bean a une fonction bien définie, mais il peut demander des services à d'autres beans.
- L'exemple suivant illustre de manière simple une telle collaboration.
- Il s'agit d'un service de cumul pour des client Web.
- Chaque fois que le client fournit une valeur, celle-ci est ajoutée au contenu courant d'un accumulateur dédié au client.
- Cet accumulateur va utiliser à son tour les services d'un bean Calc pour effectuer l'addition.

Les beans entités

Les beans entités

Introduction (1/3)

- Un bean entité permet d'implémenter des objets ou composants modélisant une réalité métier comportant des données et non seulement des services.
  - Par exemple, des comptes ou des commandes sont des objets métiers.
  - Ces objets ont une existence indépendamment de processus qui les utilisent : le compte continue d'exister entre deux opérations bancaires.
  - Le bean entité est une implantation complète du concept : le bean qui modélise un compte va avoir autant d'instances que la banque a de comptes.
  - Chaque instance contient les données associés au compte avec sa propre identité. À un instant donné, toutes les instances ne sont pas en mémoire.
  - Le conteneur instancie au moment voulu un bean en récupérant les données stockées dans une base de données.

Introduction (2/3)

- Lors des mises à jour, le conteneur a la responsabilité d'enregistrer les données dans la base.
- En cas d'accès simultanés au même bean, le conteneur gère la concurrence d'accès directement au niveau du bean entité.
- Toute utilisation de l'objet métier, comme le compte, se fera par l'intermédiaire du bean.
- Le serveur offre les mêmes garanties en ce qui concerne les transactions, la sécurité,...
- Tout comme les bases de données centralisées ont permis de réutiliser les données entre plusieurs applications, les beans permettent de réutiliser les données et la logique associée entre plusieurs applications.

Introduction (3/3)

- De plus, l'aspect réparti et standardisé des EJB masque l'hétérogénéité des différentes bases de données ou systèmes sous-jacents.
- Les EJB sont un élément structurant dans le développement des applications.
- Une application Java EE met en œuvre la séparation de couches :
  - niveau affichage (application autonome, servlets, JSP),
  - niveau applicatif (programme appelants et bean session avec état),
  - niveau métier (beans entités ou beans sessions sans état).

Mapping entre bean et base de données

- Le mapping entre un bean et une base de données consiste à déterminer comment les attributs du bean seront enregistrés dans la base.
- Au plus simple, les attributs d'un bean correspondent aux attributs d'un tuple d'une table.

Types de beans entités

- La persistance des données est gérée de 2 manières.
- La synchronisation avec le support (la base) peut être gérée par le conteneur (Container Managed Persistence : CMP) ou par le bean (Bean Managed Persistence : BMP).
- Dans le premier cas, il suffira de déclarer les caractéristiques de l'accès à la base pour être pris en charge par le conteneur.
- Dans le second cas, il faudra coder les différents traitement nécessaires à la communication avec le support de persistance directement dans le bean (JDBC).
- La synchronisation avec le support (recherche, mise-à-jour) est faîte au niveau du conteneur pour faire le mapping de manière transparente.
- L'utilisateur du bean ne sait pas s'il travaille en CMP ou BMP. Par contre du point de vue développement, le choix est important.

Développement d'un bean entité CMP (1/2)

- Les beans entités, comme les beans sessions, sont définis par des interfaces métiers et Home.
- Le code est développé dans une classe d'implantation.
- La fabrique de beans (`EJBHome`) a un rôle important.
- Par exemple, dans le cas d'un compte bancaire :
  - La demande de création d'un bean entité modélisant un compte correspond à l'ouverture d'un compte (insertion d'un tuple dans la base).
  - Si on veut travailler avec un compte déjà existant, des méthodes adaptées (Finder) permettent de retrouver le bean associé. La fabrique possède quatre sortes de méthodes :
    1. les méthodes de création,
    2. les méthodes de recherche,
    3. les méthodes de suppression,
    4. les méthodes métiers non associées à une instance du bean : ce cas permet d'associer approche objet et approche service.

Développement d'un bean entité CMP (2/2)

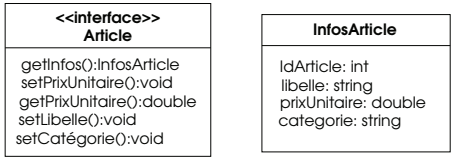
- Le développeur de bean doit donc fournir :
  - l'interface Home qui gère le cycle de vie.
  - l'interface métier qui définit les services métiers.
  - la classe du bean qui implémente les services métiers et les méthodes de contrat entre le bean et le conteneur.
  - une classe de clé primaire.
  - des informations dans le descripteur de déploiement.
- L'exemple suivant est celui d'un site e-commerce de vente d'articles.
  - Les articles sont modélisés par un bean Article.
  - Les articles appartiennent à des catégories d'articles modélisée par une classe Catégorie.
  - Les articles vont être implémentés sous forme d'un bean entité.
  - Pour l'instant, l'attribut catégorie sera directement mis dans le bean.

L'interface métier (1/4)

- L'interface métier hérite de `EJBObject`. À ce titre, elle hérite des méthodes `getEJBHome()`, `getHandle()`, `getPrimaryKey()`, `isIdentical()` et `remove()`.
- La méthode `getPrimaryKey()` ne s'applique qu'aux beans entités. Elle retourne la clé primaire (type Object à transtyper : cast ou méthode `narrow()`).
- La méthode `isIdentical()` compare 2 beans. Ils sont identiques s'ils sont de même classe et ont la même clé.
- La méthode `remove()` déclenche la suppression complète de l'objet tant en mémoire qu'en base de données.
- L'interface possède en outre les méthodes du bean visibles au client : accesseurs et méthodes métiers.

L'interface métier (2/4)

- Par exemple, pour les articles, on a :



- On introduit une classe InfosArticle, sérialisable, qui regroupe les données d'un article.
- La méthode getInfos() retourne une instance de cette classe.
- Cela permet d'obtenir tous les attributs de l'article en un seul appel, ce qui est utile lors d'appels distants.
- Le prix unitaire est obtenu de même.
- Les autres méthodes permettent de modifier le libellé, la catégorie et le prix. Ces choix sont dépendants des besoins de l'application.

L'interface métier (3/4)

```
import java.util.*;
import javax.ejb.*;
import java.rmi.RemoteException;

public interface Article extends EJBObject{
  public double getPrixUnitaire() throws RemoteException;
  public void setCategorie(String categorie) throws RemoteException;
  public void setLibelle(String libelle) throws RemoteException;
  public void setPrixUnitaire(double prixUnitaire) throws RemoteException;
  public InfosArticle getInfos() throws RemoteException;
}
```


L'interface métier (4/4)

- La classe InfosArticle est :

```
public class InfosArticle implements java.io.Serializable {
  public final Integer idArticle;
  public final String libelle;
  public final double prixUnitaire;
  public final String categorie;

  public InfosArticle(Integer argIdArticle, String argLibelle,
                      double argPrixUnitaire, String argCategorie) {
    super();
    idArticle = argIdArticle;
    libelle = argLibelle;
    prixUnitaire = argPrixUnitaire;
    categorie = argCategorie;
}
}
```

- Toutes les méthodes métiers de l'interface métier distante doivent disposer d'une clause `throws` qui contient obligatoirement `RemoteException`.

L'interface Home (1/4)

```
import java.util.*;
import javax.ejb.*;
import java.rmi.RemoteException;

public interface ArticleHome extends EJBHome {
  public Article create(Integer idArticle, String libelle, double prixUnitaire, String categorie)
    throws RemoteException, CreateException;
  public Article createSansID(String libelle, double prixUnitaire, String categorie)
    throws RemoteException, CreateException;
  public Collection findByCategorie(String categorie)
    throws RemoteException, FinderException;
  public Article findByPrimaryKey(Integer id)
    throws RemoteException, FinderException;
  public Integer getNbreArticleDeCategorie(String categorie)
    throws RemoteException;
}
```

- Les méthodes de création :
  - La création correspond à l'insertion dans une base de données.
  - Les méthodes de création sont de la forme `createXxxx()`.
  - La méthode `create()` doit créer un article à partir des éléments fournis en paramètres.
  - La méthode `createSansId()` fait la même chose mais détermine elle-même l'identifiant de l'article.

L'interface Home (2/4)

- Les méthodes de recherche :
  - Elles permettent de retrouver des beans déjà existants.
  - Elles s'appellent Finder. Elles sont de la forme `findXxxx()`.
  - Elles sont de 2 types : celles qui retourne au plus un bean et celles qui en retourne plusieurs.
  - Celles qui retourne au plus un bean ont pour type de retour l'interface métier, les autres l'interface `Collection`.
  - Dans tous les cas elles doivent lever l'exception FinderException et `RemoteException`.
  - L'exception `ObjectNotFound` (sous classe de `FinderException`) est levée si aucun bean n'est retourné dans le premier cas et une collection vide dans le second cas.
  - La méthode `findByPrimaryKey()` est obligatoire.
  - Dans les beans CMP, l'implantation des méthodes de recherche est effectuée par le conteneur.

L'interface Home (3/4)

```
<entity>
  <display-name>ArticleBean</display-name>
  <ejb-name>ArticleBean</ejb-name>
  <home>fr.orbycom.ejbcmp.ArticleHome</home>
  <remote>fr.orbycom.ejbcmp.Article</remote>
  <ejb-class>fr.orbycom.ejbcmp.ArticleBean</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>java.lang.Integer</prim-key-class>
  <reentrant>False</reentrant>
  <cmp-version>2.x</cmp-version>
  <abstract-schema-name>ArticleSN</abstract-schema-name>
  ...
  <query>
    <description></description>
    <query-method>
      <method-name>findByCategorie</method-name>
      <method-params>
        <method-param>java.lang.String</method-param>
      </method-params>
    </query-method>
    <ejb-ql>
    select object(a) from ArticleSN a where a.categorie = ?1
    </ejb-ql>
  </query>
</entity>
```


L'interface Home (4/4)

- Dans le cas de la méthode `findByCategorie()`, il faut lui indiquer la requête à effectuée à l'aide d'EJB QL (EJB 2.0) dans le descripteur associé.
- La méthode `findCategorie()` retourne le bean entité Article dont la catégorie est égale au premier paramètre de la fonction.
- `ArticleSN` est ici le nom logique que l'on a donné au bean Article. Ce nom est défini dans la balise <abstract-schema-name>.

- Les méthodes sans entité (EJB 2.0) :
  - Elles simplifient certains traitements.
  - Ce sont des méthodes de type service qui ne s'exécutent pas sur un bean précis.
  - C'est le cas de la méthode `getNbreArticleDeCategorie()`.

La classe d'implantation (1/2)

- La classe d'implantation du bean doit implémenter l'interface `EntityBean` (déclare des méthodes dites de rappel, callback) qui sont invoquées automatiquement par le conteneur.
- Le bean étant géré par le conteneur, la classe d'implantation doit être abstraite (EJB 2.0).
- Cela permet au conteneur de sous-classer la classe d'implantation.
- Cette classe contient le code qui est exécuté suite à l'appel de méthodes sur les interfaces Home et métier.
- C'est le conteneur qui déclenche ces appels.

La classe d'implantation (2/2)

- La classe d'implantation est composée de :
  - méthodes de gestion des champs persistants.
  - méthodes métiers déclarées dans les interfaces métiers.
  - constructeur sans paramètre (appelé par le conteneur).
  - méthodes liés aux méthodes de création déclarées dans les interfaces Home.
  - méthodes sans entités déclarées dans les interfaces Home.
  - méthodes internes d'accès aux données, appelées méthodes Select.
  - méthodes de rappel, appelées par le conteneur.
- Détaillons tout cela...

Les méthodes de gestion des champs CMP (1/2)

- À chaque champ correspond un accesseur get() et un accesseur set(). Le descripteur spécifie la correspondance des accesseurs avec les champs.

```
<enterprise-beans>
  <entity>
    <display-name>ArticleBean</display-name>
    <ejb-name>ArticleBean</ejb-name>
    <home>fr.orbycom.ejbcmp.ArticleHome</home>
    ...
    <cmp-version>2.x</cmp-version>
    <abstract-schema-name>ArticleSN</abstract-schema-name>
    <cmp-field>
      <field-name>libelle</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>categorie</field-name>
    </cmp-field>
    <cmp-field>
    <field-name>idArticle</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>prixUnitaire</field-name>
    </cmp-field>
  </entity>
```


Les méthodes de gestion des champs CMP (2/2)

- La règle est qu'à chaque champ xy correspond les accesseurs getXy() et setXy().
- Lors du déploiement, la classe du bean est sous-classée et les méthodes abstraites générées.
- Dans EJB 1.1, la classe du bean était concrète et les attributs persistants apparaissaient directement dans la classe du bean sous forme d'attributs java.

Les méthodes métiers

- Ces méthodes implémentent les méthodes définies dans l'interface métier.
- Dans EJB 2.0, `RemoteException` est proscripte (mais tolérée pour des raisons de compatibilité) en dehors de l'interface métier distante.
- Les exceptions applicatives doivent utiliser une exception héritant ni de `RemoteException` ni de RuntimeException.

Les méthodes de création (1/2)

- Pour chaque méthode de création `createXxxx()` des interfaces Home, on doit écrire 2 méthodes nommées respectivement `ejbCreateXxxx()` et `ejbPostCreateXxxx()` avec les mêmes paramètres.
- Les méthodes `ejbCreateXxxx()` doivent retourner un paramètre de type clé primaire. `ejbPostCreateXxxx()` ne retourne rien. `ejbCreateXxxx()` doit comporter la clause throws `CreateException`. `ejbPostCreateXxxx()`. Cette clause n'est pas obligatoire pour `ejbPostCreateXxxx()`.
- La méthode `ejbCreateXxxx()` doit alimenter les champs du bean gérés par le conteneur à partir des informations passées en paramètre et en utilisant les `get()` du bean.
- La méthode `ejbCreateXxxx()` doit retourner null.

Les méthodes de création (2/2)

- Le conteneur se charge ensuite de retrouver la clé primaire à partir des attributs, de récupérer les valeurs des champs CMP et de les stocker sur le support de persistance.
- La méthode `ejbPostCreate()` correspondante est appelée après que le support de persistance a été mis à jour.
- Cette méthode peut être codée avec des traitements spécifiques pour finaliser la création du bean.
- Les méthodes `getPrimaryKey()`, `getEJBObject()` ne sont pas accessibles dans les méthodes `ejbCreateXxxx()` mais le sont dans les méthodes `ejbPostCreate()`.

Les méthodes sans entités

- Définies dans l'interface Home, elles s'implantent dans la classe du bean sous la forme `ejbHomeXxxx()`.
- Il faut veiller à les optimiser.

Les méthodes Select (1/5)

- Il arrive qu'un bean entité ait besoin d'accéder à un autre bean, de la même classe ou non, pour effectuer des traitements.
- La logique EJB standard parvient plus ou moins bien à résoudre ce problème en terme de performance.
- Ainsi la méthode `getNbreArticleDeCategorie()` du bean Article nécessite d'obtenir l'interface Home du bean Article, appeler la méthode de recherche des beans par catégorie, puis de compter le nombre d'articles trouvés. Cela est très coûteux.
- Il est préférable de déclarer une méthode abstraite appelée Select et nommée `ejbSelectXxxx()`, à laquelle est associée une requête EJB QL et renvoyant une `Collection` sur un attribut, l'identifiant, ce qui améliore les performances.

Les méthodes Select (2/5)

- On déclare la méthode `ejbSelectIdArticlesDeCategorie()` dans le bean Article. Cette méthode n'a pas d'implantation car c'est le conteneur qui la fournit.

```
public abstract Collection ejbSelectIdArticlesDeCategorie(String categorie)
  throws FinderException;
```


Les méthodes Select (3/5)

- avec l'ajout suivant dans le descripteur :

```
<query>
  <query-method>
    <method-name>
    ejbSelectIdArticlesDeCategorie
    </method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-name>
  </query-method>
  <ejb-ql>
  select a.idArticle from ArticleSN a where a.categorie = ?1
  </ejb-ql>
</query>
```


Les méthodes Select (4/5)

- Cela permet de simplifier la méthode ejbHomeGetNbreArticleDeCategorie() :

```
public Integer ejbHomeGetNbreArticleDeCategorie(String categorie) {
  int res = 0;
  try {
    Collection lesIdArticles = null;
    // Implémentation avec select
    lesArticles = ejbSelectIdArticlesDeCategorie(categorie);
    res = lesArticles.size();
} catch (FinderException fe) {
    throw new EJBException(fe);
}
  return new Integer(res);
}
```

- On remarque que le fonctionnement des méthodes de recherche Finder et des méthodes Select sont similaires.

Les méthodes Select (5/5)

- Par contre leur utilisation est différente :
  - Les méthodes Finder sont déclarées dans les interfaces Home et sont à l'usage de l'utilisateur du bean.
  - Les méthodes Select sont déclarées dans la classe du bean et utilisées de manière interne. Elles constituent une partie de l'implantation du bean.
- Les méthodes Select peuvent retourner un attribut du bean, une collection du même attribut d'une collection de beans, une interface métier ou une collection d'interfaces métiers.
- Les méthodes Finder ne peuvent retourner qu'une interface ou une collection d'interfaces métiers.

Les méthodes de rappel (1/5)

- Les méthodes de rappel de l'interface `EntityBean` sont utilisées par le conteneur pour communiquer avec le bean.
- Ces méthodes indiquent au bean CMP ce que le conteneur a fait ou va faire pour synchroniser les données avec le support de persistance.
- Il y a 7 méthodes : `ejbActivate()`, `ejbPassivate()`, `ejbRemove()`, `ejbLoad()`, `ejbStore()`, `setEntityContext()` et `unsetEntityContext()`.
- Elles correspondent à des changements d'état du bean.

Les méthodes de rappel (2/5)

- La méthode `setEntityContext()` :
  - Elle est appelée lorsque le bean est créé.
  - Le bean n'est pas encore associé à une entité particulière (accès sans signification).
  - Cette méthode peut allouer des ressources non spécifiques à une entité particulière.
  - Une opération classique de cette méthode est d'enregistrer la référence du contexte passé en paramètre dans une variable d'instance afin de pouvoir y accéder lors de la vie de l'instance.
- La méthode `ejbActivate()` :
  - Elle est appelée par le conteneur quand le conteneur prend une instance du pool pour l'associer à une entité spécifique.
  - À cet instant, on connaît l'identité de l'objet (clé primaire), mais pas son contenu.
  - Elle ne peut donc accéder aux champs persistants du bean.
  - L'identité peut être obtenue par les méthodes `getPrimaryKey()` ou `getEJBObject()`.

Les méthodes de rappel (3/5)

- La méthode `unsetEntityContext()` :
  - Elle est appelée par le conteneur avant de supprimer l'instance de la mémoire, pour relâcher des ressources par exemple acquises dans la méthode setEntityContext().
- La méthode `ejbLoad()` :
  - Elle est appelée après que le conteneur ait alimenté les champs persistants.
  - Dans les beans CMP, il faut recalculer toutes les valeurs non persistantes.
  - `ejbLoad()` indique qu'une synchronisation vient d'être effectuée en lecture, donc tous les champs doivent être mis à jour : le conteneur s'occupe des champs CMP, le développeur s'occupe des autres.

Les méthodes de rappel (4/5)

- La méthode `ejbStore()` :
  - Elle est appelée lors de la synchronisation entre le bean en mémoire et le support de persistance.
  - Il est possible d'effectuer des opérations avant la persistance (par exemple compression de texte).
- La méthode `ejbRemove()` :
  - Elle est appelée en réponse à l'invocation par le client d'une méthode `remove()` sur une interface Home ou métier du bean.
  - L'instance est dans l'état prêt quand la méthode est appelée, elle entrera dans le pool une fois la méthode exécutée.
  - Cette méthode est appelée avant que la suppression sur le support de persistance soit automatiquement effectuée par le conteneur.
  - Il ne s'agit pas que d'une suppression du bean en mémoire.
  - Le bean est dans l'état prêt, ce qui signifie que l'on a encore accès à tous les champs persistants, ainsi qu'à l'identité du bean.

Les méthodes de rappel (5/5)

- La méthode `ejbPassivate()` :
  - Elle est appelée par le conteneur quand il décide de dissocier l'instance d'une entité particulière et de remettre cette instance dans le pool.
  - Il est possible de désallouer des ressources non utilisées par les instances du pool et qui ont été allouées dans les méthodes `ejbActivate()` et `ejbCreateXxxx()`.

Le source suivant reprend l'ensemble des méthodes vues :

```
import java.util.*;
import javax.ejb.*;
import javax.naming.*;

public abstract class ArticleBean implements javax.ejb.EntityBean {
  EntityContext ec = null;

  public ArticleBean() { }

  // accesseurs(1)

  public abstract String getCategorie();
  public abstract Integer getIdArticle();
  public abstract String getLibelle();
  public abstract double getPrixUnitaire();
  public abstract void setCategorie(String categorie);
```


La clé primaire

Les beans entités ont un identifiant matérialisé par une clé primaire.
Techniquement la clé primaire est sérialisable (type primitif, String ou classe ad-hoc).
Elle est indiquée dans le descripteur par les balises <prim-key-class> et <primkey-field>.

```
<entity>
  <display-name>ArticleBean</display-name>
  <ejb-name>ArticleBean</ejb-name>
  <home>fr.orbycom.ejbcmp.ArticleHome</home>
  <remote>fr.orbycom.ejbcmp.Article</remote>
  <ejb-class>fr.orbycom.ejbcmp.ArticleBean</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>java.lang.Integer</prim-key-class>a
  <reentrant>False</reentrant>
  <cmp-version>2.x</cmp-version>
  ...
  <cmp-field>
    <description>no description</description>
    <field-name>idArticle</field-name>
  </cmp-field>
  ...
  <primkey-field>idArticle</primkey-field>
</entity>
```


Le descripteur de déploiement

- Le descripteur de déploiement est un fichier XML donnant des informations sur le bean entité. Il permet lors du déploiement de générer le code spécifique au conteneur.
- Les informations présentes dans le conteneur sont réparties ainsi :
  - la partie enterprise-beans : informations strictement liées au bean.
  - la partie relationships : informations sur la gestion des relations par les CMP.
  - la partie assembly-descriptor : informations sur la gestion des appels par le conteneur : sécurité...

Déploiement d'un bean entité

Le déploiement se fait à l'aide d'outils spécifiques du serveur d'applications utilisé.

Utilisation d'un bean entité :

- En partant de l'existence du bean Article, on va l'utiliser dans un programme client qui crée un article, le retrouve et l'affiche.
- Il est utile d'encapsuler les fonctionnalites de base dans des méthodes plus orientées client.

Création d'un bean

```
public Article creerArticle(int numeroArticle,
                            String libelle,
                            double montantUnitaire,
                            Categorie categorie) {
  ArticleHome ah;
  Article article = null;
  try {
    Context ic = new InitialContext();
    ah = (ArticleHome) ic.lookup("java:comp/env/ejb/Article");

    article = ah.create(new Integer(numeroArticle),libelle, montantUnitaire, categorie);
    afficherInfosArticle(article.getInfos());
  } catch (Throwable th) {
    System.out.println("Erreur dans creerArticle : " + th);
  }
  return article;
}
```

avec `afficherInfosArticle()` :

```
public void afficherInfosArticle(InfosArticle infos) {
  System.out.println("voici les infos sur l'article : " + infos.idArticle);
  System.out.println(" id : " + infos.idArticle);
  System.out.println(" libelle : " + infos.libelle);
  System.out.println(" prix unitaire : " + infos.prixUnitaire);
  System.out.println(" categorie : " + infos.codeCategorie);
}
```


Recherche d'un bean (1/2)

La recherche de bean se fait par l'utilisation des méthodes Finder. La méthode suivante recherche un Article sur sa clé primaire puis l'affiche :

```
public Article rechercherArticle(int numeroArticle) {
  ArticleHome ah;
  Article article = null;
  try {
    Context ic = new InitialContext();
    ch = (ArticleHome) ic.lookup("java:comp/env/ejb/Article");
    article = ah.findByPrimaryKey(new Integer(numeroArticle));

    InfosArticle ia = article.getInfos();
    afficherInfosArticle(ia);
  } catch (Throwable th) {
    System.out.println("GereCommande.creerArticle : " + th);
  }
  return article;
}
```


Recherche d'un bean (2/2)

La méthode suivante utilise la `Collection` renvoyée par la recherche sur la catégorie :

```
public void rechercherArticlesParCategorie(String categorie) {

  ArticleHome ah;
  Article article;
  Collection col;
  try {
    Context ic = new InitialContext();
    ah = (ArticleHome) ic.lookup("java:comp/env/ejb/Article");
    col = ah.findByCategorie(categorie);

    Iterator i = col.iterator();
    InfosArticle ip = null;
    while (i.hasNext()) {
      article = (Article) i.next();
      afficherInfosArticle(article.getInfos());
    }
  } catch (Throwable th) {
    System.out.println("Erreur dans rechercherArticlesParCategorie : " + th);
  }
}
```

Le client

```
public static void main(java.lang.String[] args) {

  Article article1, article2, article3, article4;

  article1 = creerArticle(1,"Les misérables", 21, "LIVRE");
  article2 = creerArticle(2,"Céline Dion au stade de France", 120, "CD");
  article3 = creerArticle(3,"Je l'aime à mourir", 28, "LIVRE");
  article4 = creerArticle(4,"La mer", 38, "LIVRE");

  // Recherche de l'article 3
  rechercherArticle(3);

  // Recherche de la catégorie
  rechercherArticlesParCategorie("CD");
}
```


Les beans messages

Les beans messages

Qu'est-ce qu'un bean message ?

- En anglais, Message-Driven Bean (MDB).
- Ils ont été introduits dans la spécification EJB 2.0 pour prendre en charge le traitement asynchrone de messages.
- Ils permettent aux applications Java EE de traiter (essentiellement) des messages JMS (Java Message Service), et également d'autres types de messages, en se comportant comme un écouteur (listener) de messages JMS qu'ils traitent de manière asynchrone.
- Le bean message implémente l'interface `javax.jms.MessageListener` qui lui permet de réagir aux messages JMS reçus via la méthode `onMessage()`.

Des ressemblances avec les beans sessions sans état

- Ils ne conservent aucun état, aucune donnée d'un client spécifique.
- Toutes les instances de beans messages sont équivalentes.
- Le conteneur EJB peut envoyer des messages à n'importe laquelle d'entre elles, et ces messages peuvent être en concurrence et traités via un pool d'instances géré par le conteneur.
- Un bean message peut envoyer des messages à de multiples clients.

Caractéristiques des beans messages

- Ils s'exécutent à la réception d'un message provenant d'un client.
- Ils ont une durée de vie plutôt courte.
- Ils sont invoqués de manière asynchrone.
- Ils ne représentent pas directement des données partagées dans une base de données mais peuvent y accéder et les modifier.
- Ils sont sans état.
- Ils ne peuvent être exécuter dans une transaction existante (spécification EJB 2.0), mais peuvent créer une nouvelle transaction dans la méthode `onMessage()`, durant laquelle plusieurs opérations pourront être effectuées.

Différences avec les beans sessions sans état

- Le client n'utilise pas d'interface pour accéder au bean message.
- Le bean est constitué d'une seule et unique classe.
- L'utilisation des beans messages est préférée à celle des beans sessions sans état lorsque les messages doivent être reçus de manière asynchrone afin de ne pas saturer les ressources côté serveur.

API JMS (Java Message Service) - Généralités (1/2)

- L'API JMS n'est pas précisément une API mais une spécification de Sun pour la gestion des messages de la plate-forme Java EE.
- Elle est composée essentiellement d'interfaces qui simplifient le travail du développeur pour l'envoi ou la réception de messages.
- Quatre acteurs interviennent lors de l'utilisation de JMS dans un système :
  - Le message lui même, qui est transmis entre 2 applications.
  - Les applications qui fournissent et reçoivent le message, par exemple un client extérieur au serveur d'applications et un EJB ou tout applicatif Java EE.

API JMS (Java Message Service) - Généralités (2/2)

- Un provider JMS qui fournit des outils d'administration de messages et qui est chargé de retrouver les messages qui lui sont adressés pour pouvoir les délivrer.
- Les objets administrés qui sont les ressources à rechercher dans le serveur d'applications ou dans l'annuaire JNDI du provider. On distingue 2 types d'objets :
  - les ConnectionFactory qui sont des « usines à connexion » servant à se connecter au provider,
  - et les « Destination » qui sont soit des « Queue » soit des « Topic ».

API JMS - Deux modes de connexion

- Le mode **Point à Point** (Point to Point) dans lequel on a un seul destinataire pour chaque message.
  - Le message est envoyé dans une file d'attente (Queue) et est supprimé du provider JMS dès qu'il a été « consommé » (ou qu'il a expiré).
  - Le producteur et le consommateur du message n'ont pas besoin d'être connectés au même moment.
- Le mode **Publication/Souscription** (Publish/Subscribe) s'apparente au fonctionnement d'un serveur de news.
  - Un message est publié sur un « Topic » et les consommateurs de ce message doivent s'incrire aux Topics qui les intéressent.
  - Le message est supprimé du provider lorsqu'il a été lu et acquitté par tous les souscripteurs du Topic. Il peut donc avoir plusieurs destinataires.

API JMS - Deux modes de consommation

- Les messages en provenance d'une Queue ou d'un Topic peuvent être reçus de 2 façons :
  - De manière synchrone, l'application est bloquée lors de l'appel à la méthode `receive()`. Une version surchargée de cette méthode permet de rendre la main après l'écoulement d'un *timeout*.
  - De manière asynchrone où l'applicatif est informé de l'arrivée d'un message via un `MessageListener`. Ce *listener* lance un thread qui attends les messages et exécute une méthode dès leurs arrivées.

Développement d'un bean message (1/2)

- La classe principale du bean message doit implémenter les deux interfaces `MessageListener` et `MessageDrivenBean`.
- L'interface `MessageListener` :
  - elle permet d'implémenter la méthode `onMessage (Message)` déclenchée lors de la réception d'un message.
  - le message passé en argument peut être de type `BytesMessage`, `ObjectMessage`, `TextMessage`, `StreamMessage` ou `MapMessage`. Pour connaître le type du message reçu, il faut utiliser l'opération `instanceof`.

Développement d'un bean message (2/2)

- L'interface `MessageDrivenBean` :
  - elle permet d'implémenter les méthodes `setMessageDrivenContext(MessageDrivenContext)` et `ejbRemove()`.
  - la méthode `setMessageDrivenContext(MessageDrivenContext)` est appelée par le container à la création du Bean.
  - la méthode `ejbRemove()` est appelée par le container à la destruction du Bean.
- Il faut toujours implémenter la méthode `ejbCreate()` et définir un descripteur de déploiement au format XML.

La classe du bean message (1/2)

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

public class MessageBean implements MessageDrivenBean, MessageListener {

  protected MessageDrivenContext ctx;

  //association de cette instance du bean à un context particulier
  public void setMessageDrivenContext(MessageDrivenContext ctx) {
    this.ctx = ctx;
  }

  //Initialisation du bean
  public void ejbCreate() {
    System.out.println("ejbCreate()");
  }
```


La classe du bean message (2/2)

```
  //Méthode de réception des messages
  public void onMessage(Message msg) {
    TextMessage tm = (TextMessage) msg;

    try {
      String text = tm.getText();
      System.out.println("Message Recu : " + text);
    }catch(JMSException e) {
      e.printStackTrace();
    }
  }

  //Destruction du bean
  public void ejbRemove() {
    System.out.println("ejbRemove()");
  }
}
```


Le descripteur de déploiement

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
 "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">

<ejb-jar>
  <enterprise-beans>
    <message-driven>
      <ejb-name>MessageBean</ejb-name>
      <ejb-class>MessageBean</ejb-class>
      <transaction-type>Container</transaction-type>
      <message-driven-destination>
        <destination-type>javax.jms.Topic</destination-type>
      </message-driven-destination>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

Le client

```
import javax.naming.*;
import javax.jms.*;
import java.util.*;

public class Client {

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

    // 1: Recherche d'une fabrique de connexions via le JNDI
    TopicConnectionFactory  factory = (TopicConnectionFactory) ctx.lookup("ConnectionFactory");

    // 2: Utilisation de la fabrique de connexions pour créer une connexion JMS
    TopicConnection connection = factory.createTopicConnection();

    // 3: Utilisation de la connection pour créer une session
    TopicSession session = connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);

    // 4: Recherche du sujet (topic) via le JNDI
    Topic topic = (Topic) ctx.lookup("topic/MessageBean");

    // 5: Création d'un producteur de message
    TopicPublisher publisher = session.createPublisher(topic);

    // 6: Création d'un  message texte et publication
    TextMessage msg = session.createTextMessage();
    for (int i=0;i<10;i++){
      msg.setText("Envois message : " + i);
      publisher.publish(msg);
    }
  }
}
```


La norme 3.0

La norme 3.0

Introduction

- La norme EJB 3.0 constitue une évolution importante dans le domaine des EJB.
- Elle constitue une partie importante de la norme Java 2 EE 5.
- Un nouveau modèle de gestion de la persistance des données inspiré d'Hibernate de Jboss.
- Une simplification très importante des développements.
  - Utilisation des annotations introduites dans la version 5.0 du langage Java.
  - Adoption du modèle de programmation POJO.
  - Suppression des descripteurs de déploiement.
  - Suppression des interfaces métiers et des interfaces Home.
- La norme EJB 3.0 intègre aussi une approche Programmation Orientée Aspect (AOP) en introduisant la notion d'intercepteur (Interceptor).

Objectifs de la norme 3.0

- Simplification de la conception des Beans coté développeur.
- Simplification de la définition des interfaces, suppression d'un certain nombre de pré-requis présents dans la norme 2.1 (absence d'héritage de super classes ou interfaces).
- Simplification de la création du Bean.
- Simplification des API pour l'accès à l'environnement du Bean : définition par simple injection dépendante.
- Introduction des annotations en Java en lieu et place du descripteur de déploiement.
- Simplification concernant la persistance d'objet par l'utilisation facilitée de mapping objet/relationnel basée sur l'utilisation directe de classes Java et non de composants persistants.

Les annotations

- Métadonnées permettant à certains outils de générer des constructions additionnelles à la compilation ou à l'exécution.
- Elles simplifient considérablement l'écriture de programmes.
- Elles permettent de se passer du descripteur de déploiement. Néanmoins, il est toujours possible de l'utiliser.
- Elle permettent de spécifier les interfaces locales ou distantes du Bean avec uniquement les mots clefs `@Remote` et `@Local`.
- Elle permettent de définir un Bean session avec ou sans état avec uniquement les mots clefs `@Statefull` ou `@Stateless`.

Hello World avec les EJB 3.0 (1/4)

- Définition de l'interface Hello :
  - Désormais, cette interface est la seule à définir.
  - Interface extrêmement simplifiée : pas besoins d'avoir recours à de quelconques héritages.

```
package hello;

public interface Hello {
    public String hello (String msg);
}
```


Hello World avec les EJB 3.0 (2/4)

- Définition du Bean :
  - Cette classe est aussi très simplifiée.
  - Elle n'hérite pas de classes spécifiques.
  - Elle doit pallier à la suppression des interfaces distantes et locales en utilisant des annotations.
  - On utilise une annotation pour spécifier qu'il s'agit d'un bean Session sans état.

```
package hello;

import javax.ejb.*;

@Stateless
@Remote (Hello.class)
public class HelloBean implements Hello {
    public String hello (String msg){
        System.out.println ("Message reçu : "+msg);
        return "Hello "+msg;
    }
}
```

Hello World avec les EJB 3.0 (3/4)

- Définition du descripteur de déploiement : il n'en reste presque rien ce qui simplifie son écriture

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<ejb-jar xmlns="http://java.sun.com/xmlns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                             http://java.sun.com/xml/ns/j2ee/ejb-jar_3_0.xsd"
         version="3.0">
  <enterprise-beans>
  </enterprise-beans>
</ejb-jar>
```


Hello World avec les EJB 3.0 (4/4)

- Définition du client : lui aussi est simplifié par rapport au même client dans les EJB 2.1.

```
import javax.naming.Context;
import javax.naming.InitialContext;
import hello.*;
import java.util.*;

public class HelloClient{
    public static void main(String[] args) throws Exception {
        Properties props = System.getProperties();
        props.put(Context.INITIAL_CONTEXT_FACTORY,
                  "org.jnp.interfaces.NamingContextFactory");
        props.put(Context.URL_PKG_PREFIXES,
                  "org.jboss.naming:org.jnp.interfaces");
        props.put(Context.PROVIDER_URL, "jnp://localhost:1099");

        Context ctx = new InitialContext(props);
        Hello hello = (Hello) ctx.lookup("HelloBean/remote");

        System.out.println(hello.hello("World"));
    }
}
```


Cycle de vie des Beans 3.0 (1/3)

- Disparition des interface Home : les références sur les beans se font directement.
- Utilisation des méthodes "callback" pour la gestion du cycle de vie.
- Les méthodes callback peuvent être définies :
  - soit directement dans le bean : la signature de la méthode doit être `public void nom_methode ()`,
  - soit dans une classe Listener Callback qui ressemble à une interface home.
- La classe Listener Callback :
  - toutes les méthodes sont définies dans la classe.
  - les méthodes prennent en argument le bean correspondant.
  - il suffit d'importer cette classe dans la classe du bean CallbackListener.
  - la signature de la méthode doit être `public void nom_methode (Object)` où `Object` est le type du bean, NomDeLaClasseCallback.

Cycle de vie des Beans 3.0 (2/3)

- Les méthodes "callback" doivent être annotées avec les annotations de callback :

```
@PostConstruct
@PreDestroy
@PostActivate
@PrePassivate
@PrePersist
@PostPersist
@PreRemove
@PostRemove
@PreUpdate
@PostLoad
@CallBackListener(String classname)
@EntityListener(String classname)
```

- Exemple 1 : Définition des méthodes "callback" directement dans le bean.

```
@Stateful
public class MonBean {
    private float total;
    private Vector productCodes;
    public int desMethods(){...};
    ...
    @PreDestroy public void uneMethodeCallback() {...};
    ...
}
```

Cycle de vie des Beans 3.0 (3/3)

- Exemple 2 : Définition des méthodes "callback" en dehors du bean.

Tout d'abord, on définit une classe Listener qui contiendra les méthodes "callback".

```
public class MonBeanListener {
    @PreDestroy public void uneMethodeCallback() {...};
    ...
}
```

Puis, on fait appel au "callback listener" dans le bean.

```
@CallbackListener MonBeanListener
@Stateful
public class MonBean {
    private float total;
    private Vector productCodes;
    public int desMethods(){...};
    ...
}
```


L'injection de dépendance (1/2)

- Elle permet à un bean d'acquérir des références sur d'autres ressources ou d'autres objets de l'application.
- Auparavant, il fallait avoir recours aux fichiers XML (avec `ejb-ref`) et se servir du JNDI pour accéder à une ressource :
  - Pour accéder à un EJB, il était obligatoire de passer par son interface Home pour récupérer une référence sur son interface et utiliser ses méthodes métiers.
- Maintenant, c'est le conteneur qui injecte ces références avant que toute méthode liée au cycle de vie de l'EJB ou méthode métier ne soit appelée.
- Le conteneur se charge d'initialiser toutes les ressources dont le bean a besoin.

L'injection de dépendance (2/2)

- Exemple de récupération d'une ressource de type base de données :

```
// On récupère une ressource maBD et on l'affecte à clientBD
// Le type de maBD est déduit de la variable clientBD
@Ressource (name="maBD")
public SourceDesDonnees clientBD;
```

- Exemple de récupération de la référence d'un autre EJB :

```
@EJB
public AddressHome addressHome;
```

- On peut aussi passer par les méthodes `setXXXX` pour récupérer une ressource :

```
// récupère la ressource de nom clientBD et de type SourceDesDonnees
@Ressource(name="clientBD")
public void setSourceDesDonnees(SourceDesDonnees clientBD){
  this.sdd = clientBD;
}

// récupère la ressource de nom maBD et de type SourceDesDonnees
@Ressource
public void setClientBD(SourceDesDonnees maBD){
  this.clientBD = maBD;
}
```


Les méthodes d'interception

- Appelées méthodes "interceptors", elles seront exécutées à chaque appel d'une méthode métier définissant ces méthodes.
- Elles sont de la forme :

```
public Object <methodName>(javax.ejb.InvocationContext)
    throws Exception
```

- Toute méthode interceptor doit prendre en argument l'interface suivante :

```
public interface InvocationContext {
  // retourne le bean cible de la méthode interceptor
  public Object getTarget ();
  // retourne la méthode déclenchant l'interceptor
  public Method getMethod ();
  // retourne les paramètres de la méthode métier
  public Object[] getParameters();
  public void setParameters (Object[] params);
  public java.util.Map<String, Object> getContextData();
  // exécute la méthode interceptor suivante ou
  // la méthode annotée @AroundInvoke
  public Object proceed() throws Exception;
}
```

Exemple de méthodes interceptor

```
@Stateless
@Interceptor("TestInterceptor")
public class LoginBean {
  @AroundInvoke
  public Object testInterceptor (InvocationContext invContext) throw Exception {
    invContext.proceed ();
  }
}

public class TestInterceptor {
  @AroundInvoke
  public Object myInterceptor (InvocationContext invContext) throws Exception{
    invContext.proceed ();
  }
}
```


Beans sessions sans état

- La classe doit être annotée avec `@Stateless (import javax.ejb.Stateless;)`
- Ces beans supportent les callback d'évènements : `@PostConstruct`, `@PreDestroy`, mais aussi `@PostActivate` et `@PrePassivate` qui correpondent aux méthodes `ejbActivate` et `ejbPassivate` présentes dans les EJB 2.x.
- La méthode callback `@PostConstruct` intervient après toutes les injections de dépendances effectuées par le conteneur et avant le premier appel d'une méthode métier.
- La méthode callback `@PreDestroy` est appelée au moment où l'instance du bean est détruite.

Beans sessions sans état (Exemple) (1/2)

L'interface distante du Bean :

```
package calc;

import javax.ejb.Remote;

@Remote public interface Calc {
    public double add(double val1, double val2);
    public double mult(double val1, double val2);
}
```

La classe du Bean :

```
package calc;

import javax.ejb.Stateless;

@Stateless public class CalcBean implements Calc {

    public double add(double val1, double val2) {
        return val1 + val2;
    }

    public double mult(double val1, double val2) {
        return val1 * val2;
    }
}
```


Beans sessions sans état (Exemple) (2/2)

Le client du bean :

```
import javax.naming.Context;
import javax.naming.InitialContext;
import calc.*;
import java.util.*;

public class CalcClient{
    public static void main(String[] args) throws Exception {
        Properties props = System.getProperties();
        props.put(Context.INITIAL_CONTEXT_FACTORY,
                  "org.jnp.interfaces.NamingContextFactory");
        props.put(Context.URL_PKG_PREFIXES,
                  "org.jboss.naming:org.jnp.interfaces");
        props.put(Context.PROVIDER_URL, "jnp://localhost:1099");

        Context ctx = new InitialContext(props);
        Calc calc = (Calc) ctx.lookup("CalcBean/remote");

        double somme = 0;
        somme = calc.add(5.643, 8.2921);
        System.out.println("Addition de 5.643 + 8.2921 = "+somme);
    }
}
```

Beans sessions avec état

- La classe doit être annotée avec `@Statefull (import javax.ejb.Statefull;)`
- Ces beans supportent les callback d'évènements : `@PostConstruct`, `@PreDestroy`, `@PostActivate` et `@PrePassivate`.
- Il existe l'annotation `@Remove` applicable à une méthode métier et qui implique la destruction du bean après son appel.

Beans sessions avec état (Exemple) (1/3)

L'interface distante du Bean :

```
package panier;

import javax.ejb.Remote;
import java.util.*;

@Remote public interface Panier{
    public void ajouterArticle(int idArticle);
    public void supprimerArticle(int idArticle);
    public Vector listerArticles();
    public void setNom(String nomClient);
    public String getNom();
    public void remove();
}
```


Beans sessions avec état (Exemple) (2/3)

La classe du Bean :

```
package panier;

import javax.ejb.*;
import javax.annotation.PostConstruct;
import java.util.*;

@Stateful public class PanierBean implements Panier{
    private Vector articles;
    private String nomClient;

    @PostConstruct public void initialise() {
        articles = new Vector();
        nomClient = "";
    }

    public void ajouterArticle(int idArticle) {
        System.out.println ("Ajout d'un nouvel article");
        articles.add(new Integer(idArticle));
    }

    public void supprimerArticle(int idArticle){
        System.out.println ("Suppression d'un article");
        articles.remove(new Integer(idArticle));
    }

    public Vector listerArticles(){
        return articles;
    }

    public void setNom(String nomClient) {
        this.nomClient = nomClient;
    }

    public String getNom() {
        return nomClient;
    }

    @Remove public void remove() {
        articles = null;
    }
}
```


Beans sessions avec état (Exemple) (3/3)

Le client du bean :

```
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import java.util.*;
import panier.*;

public class PanierClient {
    public static void main(String[] args) throws Exception {
        Properties props = System.getProperties();
        props.put(Context.INITIAL_CONTEXT_FACTORY, "org.jnp.interfaces.NamingContextFactory");
        props.put(Context.URL_PKG_PREFIXES, "org.jboss.naming:org.jnp.interfaces");
        props.put(Context.PROVIDER_URL, "jnp://localhost:1099");

        Context ctx = new InitialContext(props);
        Panier monPanier = (Panier) ctx.lookup("PanierBean/remote");
        monPanier.ajouterArticle(65);
        monPanier.ajouterArticle(53);

        Vector mesArticles = monPanier.listerArticles();
        System.out.println ("Il y a "+mesArticles.size()+" article(s) dans le panier !");
        Enumeration e = mesArticles.elements();
        while (e.hasMoreElements()) {
            System.out.println((Integer)e.nextElement());
        }

        monPanier.ajouterArticle(23);
        monPanier.ajouterArticle(18);
        monPanier.supprimerArticle(65);

        mesArticles = monPanier.listerArticles();
        System.out.println ("Il y a "+mesArticles.size()+" article(s) dans le panier !");
        e = mesArticles.elements();
        while (e.hasMoreElements()) {
            System.out.println((Integer)e.nextElement());
        }
        monPanier.remove ();
    }
}
```

Beans messages

- La classe du bean doit être annotée avec `@MessageDriven (import javax.ejb.MessageDriven;)`.
- L'interface du bean doit être du même type que celle que le bean va utiliser. Dans la plupart des cas, il s'agira de `javax.jms.MessageListener`.
- Ces beans supportent les callback d'évènements : `@PostConstruct` et `@PreDestroy`.
- Dans le cas d'une utilisation avec JMS, la méthode `public void onMessage (Message msg)` doit être redéfinit.
- Il faut également définir les différents paramètres à utiliser pour la connexion à la destination.
  - Pour cela, on utilise les propriétés : `destinationType` et `destination`.

Beans messages : exemple (1/2)

- La classe du Bean :

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

@MessageDriven(activationConfig =
{
  @ActivationConfigProperty(propertyName = "destinationType",
                            propertyValue = "javax.jms.Queue"),
  @ActivationConfigProperty(propertyName = "destination",
                            propertyValue = "queue/MessageBean")
})
public class MessageBean implements MessageListener {
  @Resource MessageDrivenContext mdc;
  //Méthode de réception des messages
  public void onMessage(Message msg) {
    TextMessage tm;
    try {
      if (msg instanceof TextMessage){
        tm = (TextMessage) msg;
        String text = tm.getText(); System.out.println("Message Bean Reçu : " + text);
      }
      else {
        System.out.println ("Message de mauvais type : "+msg.getClass().getName());
      }
    }catch(JMSException e) {
      e.printStackTrace();
      mdc.setRollbackOnly ();
    }catch(Throwable te) {
      te.printStackTrace();
    }
  }
}
```


Beans messages : exemple (2/2)

- Le client :

```
import javax.naming.*;
import javax.jms.*;
import java.util.*;

public class Client {

  public static void main (String[] args) throws Exception {
    // Initialisation JNDI
    Properties props = System.getProperties();
    props.put(Context.INITIAL_CONTEXT_FACTORY, "org.jnp.interfaces.NamingContextFactory");
    props.put(Context.URL_PKG_PREFIXES, "org.jboss.naming:org.jnp.interfaces");
    props.put(Context.PROVIDER_URL, "jnp://localhost:1099");

    Context ctx = new InitialContext(props);
    // 1: recherche d'une fabrique de connexion via le JNDI
    QueueConnectionFactory factory = (QueueConnectionFactory) ctx.lookup("ConnectionFactory");
    // 2: Utilisation de la fabrique de connexions pour créer une connexion JMS
    QueueConnection connection = factory.createQueueConnection();
    // 3: Utilisation de la connexion pour créer une session
    QueueSession session = connection.createQueueSession(false, QueueSession.AUTO_ACKNOWLEDGE);
    // 4: Recherche de la file via le JNDI
    javax.jms.Queue queue = (javax.jms.Queue) ctx.lookup("queue/MessageBean");
    // 5: Création d'un producteur de message
    QueueSender sender = session.createSender(queue);
    // 6: Creation d'un  message texte et publication
    TextMessage msg = session.createTextMessage();
    for (int i=0;i<10;i++){
      msg.setText("Envoi du message : " + i);
      sender.send(msg);
    }
  }
}
```


Beans entités (1/5)

- Ce sont eux aussi des POJOs mais leurs spécifications ont été définies à part car ils ont subi beaucoup de modifications dans la norme 3.0.
- L'annotation `@Entity (import javax.persistance.Entity;)` définit le bean comme étant de type entité.
- Le bean doit posséder au moins un constructeur par défaut et devra hériter de l'interface Serializable afin d'être utilisable à travers le réseau pour la gestion de la persistance.
- On peut spécifier deux méthodes différentes pour la gestion de la persistance au moyen de l'option *access* :
  - `@Entity(access=AccessType.FIELD)` permet d'accéder directement aux champs à rendre persistant.
  - `@Entity(access=AccessType.PROPERTY)` oblige le fournisseur à utiliser les accesseurs.

Beans entités (2/5)

- La clé primaire peut-être simple ou composée et doit être déclarée avec l'annotation `@Id`.
  - Par exemple, pour obtenir une clé qui s'incrémente automatiquement : `@Id(generate=GenerateType.AUTO)`.
- Pour les clés composées, il faut respecter certains principes :
  - La classe de la clé primaire doit être public et posséder un constructeur sans arguments.
  - Si l'accès est de type PROPERTY, la classe peut-être soit de type public soit de type protected.
  - La classe doit être sérialisable (implémenter `Serializable`).
  - Implémentation des méthodes `equals()` et `hashCode()`.
  - Si la clé primaire est mappée à de multiples champs ou propriétés, les noms des champs de cette clé doivent être les mêmes que ceux utilisés dans le bean entité.
- Les annotations permettent d'effectuer le mapping objet/relationnel et la gestion des relations entre les entités.

Beans entités (3/5)

- Lors de la création d'un bean entité, il faut effectuer le mapping de tous ses champs. Un mapping par défaut intervient lorsqu'aucune annotation précède le champ : `@Basic` spécifie ce comportement.
- `@Table` définit la table correspondant à la classe, elle prend en argumant le nom de la table.

```
@Entity(access=accessType.FIELD)
@Table(name="PAYS")
public class Pays implements Serializable {
  @Id(generate=GeneratorType.AUTO) private int id;
  @Basic private String nom;

  public Pays() {
  }

  public Pays(String nom) {
    this.nom = nom;
  }

  public int getId() {
    return id;
  }
}
```


Beans entités (4/5)

- On peut faire correspondre une valeur à un champ spécifique de la base de données en utilisant l'annotation `@Column` et des options comme `name` qui spécifie le nom de la colonne, ou des options pour définir si champ peut être nul, ...

```
@Column(name="DESC", nullable=false)
public String getDescription() {
  return description;
}
```

- Il existe les relations `OneToOne`, `OneToMany`, `ManyToOne`, `ManyToMany` (définies par les annotations correspondantes). Dans ces cas, il ne faut pas oublier de spécifier les colonnes faisant les jointures.

```
@ManyToOne(optional=false)
@JointColumn(name = "CLIENT_ID", nullable = false, updatable = false)
public Client getClient (){
  return client;
}
```


Beans entités (5/5)

- Les beans entités se manipulent par l'intermédiaire d'un `EntityManager`.
- Cet `EntityManager` peut être obtenu à partir d'un Bean Session par injection de dépendance.

```
@Stateless public class EmployeeManager{
  @Resource EntityManager em;

  public void updateEmployeeAddress (int employeeId, Address address) {
    //Recherche d'un bean
    Employee emp = (Employee)em.find ("Employee", employeId);
    emp.setAddress (address);
  }
}
```

Beans entités : exemple (1/12)

- Principe de base :
  - On crée un bean entité qui permettra de manipuler des objets persistants.
  - Dans le bean entité, on met en place un mapping entre les attributs du bean et une table de la base de données.
  - Le client n'accède pas directement aux beans entités mais passe par des beans sessions qui effectueront les manipulations nécessaires sur les beans entités.
- Les beans de l'exemple :
  - `ArticleBean` : il s'agit du bean entité.
  - `ArticleAccessBean` : il s'agit d'un bean session sans état. Il est composé de la classe `ArticleAccessBean.java` et de l'interface `ArticleAccess.java`.

Beans entités : exemple (2/12)

- L'interface `ArticleAccess.java` : définition des méthodes métiers.

```
package article;

import javax.ejb.Remote;
import java.util.*;

@Remote public interface ArticleAccess {
  public int addArticle (String libelle, double prixUnitaire, String categorie);
  public void delArticle (int idArticle);
  public InfosArticle rechercherArticle (int idArticle);
  public List rechercherLesArticlesParCategorie (String categorie);
  public List rechercherTousLesArticles ();
}
```


Beans entités : exemple (3/12)

- La classe `ArticleAccessBean.java` : la méthode `addArticle`

```
package article;

import javax.ejb.*;
import javax.persistence.*;
import java.util.*;

@Stateless public class ArticleAccessBean implements ArticleAccess {

  @PersistenceContext(unitName="Articles")
  EntityManager em;

  public int addArticle (String libelle, double prixUnitaire, String categorie){
    ArticleBean ab = new ArticleBean ();
    ab.setCategorie (categorie);
    ab.setLibelle (libelle);
    ab.setPrixUnitaire (prixUnitaire);
    em.persist(ab);
    em.flush();
    System.out.println ("AccessBean : Ajout de l'article "+ab.getIdArticle ());
    return ab.getIdArticle ();
  }
```


Beans entités : exemple (4/12)

- La classe `ArticleAccessBean.java` : la méthode `delArticle`

```
  public void delArticle (int idArticle){
        Query query = em.createQuery("DELETE FROM ArticleBean AS a WHERE a.idArticle="+idArticle);
  }

  public InfosArticle rechercherArticle (int idArticle){
    Query query = em.createQuery("SELECT a FROM ArticleBean AS a WHERE a.idArticle="+idArticle);

    List<ArticleBean> allArticles = query.getResultList();

    ArticleBean article = allArticles.get(0);
    return new InfosArticle (article.getIdArticle(), article.getLibelle(),
                             article.getPrixUnitaire(), article.getCategorie());
  }
```

Beans entités : exemple (5/12)

- La classe `ArticleAccessBean.java` : la méthode `rechercherTousLesArticles`

```
public List rechercherTousLesArticles (){
      Query query = em.createQuery("SELECT a FROM ArticleBean AS a");
      List<ArticleBean> articlesbean = query.getResultList();
      Vector<InfosArticle> articles = new Vector();
      Iterator i = articlesbean.iterator();
      ArticleBean article;
      while (i.hasNext()) {
          article = (ArticleBean) i.next();
          InfosArticle infos = new InfosArticle (article.getIdArticle(), article.getLibelle(),
                                                 article.getPrixUnitaire(), article.getCategorie());
          articles.add(infos);
      }
      return articles;
 }
```


Beans entités : exemple (6/12)

- La classe `ArticleAccessBean.java` : la méthode `rechercherLesArticlesParCategorie`

```
public List rechercherLesArticlesParCategorie (String categorie){
      Query query = em.createQuery("SELECT a FROM ArticleBean AS a WHERE categorie='"+categorie+"'");
      List<ArticleBean> articlesbean = query.getResultList();
      Vector<InfosArticle> articles = new Vector();
      Iterator i = articlesbean.iterator();
      ArticleBean article;
      while (i.hasNext()) {
          article = (ArticleBean) i.next();
          articles.add(new InfosArticle (article.getIdArticle(), article.getLibelle(),
                                         article.getPrixUnitaire(), article.getCategorie()));
      }
      return articles;
  }
}
```


Beans entités : exemple (7/12)

- Le fichier de gestion de la persistance : `persistence.xml`

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                       http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
   version="1.0">
   <persistence-unit name="Articles">
      <jta-data-source>java:/DefaultDS</jta-data-source>
      <properties>
         <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
         <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
      </properties>
   </persistence-unit>
</persistence>
```


Beans entités : exemple (8/12)

- Le client `ArticleClient.java` : la méthode `creerArticle`

```
import javax.naming.*;
import javax.rmi.*;
import javax.ejb.*;
import java.util.*;
import article.*;

public class ArticleClient {

  public void creerArticle(String libelle, double montantUnitaire, String categorie) {
       ArticleAccess ah;
       Properties props = System.getProperties();
       props.put(Context.INITIAL_CONTEXT_FACTORY, "org.jnp.interfaces.NamingContextFactory");
       props.put(Context.URL_PKG_PREFIXES, "org.jboss.naming:org.jnp.interfaces");
       props.put(Context.PROVIDER_URL, "jnp://localhost:1099");

       try {
           Context ctx = new InitialContext(props);
           ah = (ArticleAccess) ctx.lookup("ArticleAccessBean/remote");
           System.out.println ("Ajout d'un article : "+ah);
           int id = ah.addArticle(libelle, montantUnitaire, categorie);
           System.out.println ("Affichage de l'article "+id);
           afficherArticle(id);
       } catch (Throwable th) {
           System.out.println("Erreur dans creerArticle : " + th);
       }
   }
```

Beans entités : exemple (9/12)

- Le client `ArticleClient.java` : la méthode `afficherArticle`

```
public void afficherArticle(int numeroArticle) {
    ArticleAccess ah;
    Properties props = new Properties();
    props.put("java.naming.factory.initial", "org.jnp.interfaces.NamingContextFactory");
    props.put("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.interfaces");
    props.put("java.naming.provider.url", "localhost:1099");
    try {
        Context ic = new InitialContext(props);
        ah = (ArticleAccess) ic.lookup("ArticleAccessBean/remote");
        InfosArticle infos = ah.rechercherArticle(numeroArticle);
        System.out.println("voici les infos sur l'article : " + infos.idArticle);
        System.out.println("   id : " + infos.idArticle);
        System.out.println("   libelle : " + infos.libelle);
        System.out.println("   prix unitaire : " + infos.prixUnitaire);
        System.out.println("   categorie : " + infos.categorie);
    } catch (Throwable th) {
        System.out.println("GereCommande.creerArticle : " + th);
    }
}
```


Beans entités : exemple (10/12)

- Le client `ArticleClient.java` : la méthode `afficherArticlesParCategorie`

```
public void afficherArticlesParCategorie(String categorie) {

    ArticleAccess ah;
    Properties props = new Properties();
    props.put("java.naming.factory.initial", "org.jnp.interfaces.NamingContextFactory");
    props.put("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.interfaces");
    props.put("java.naming.provider.url", "localhost:1099");
    try {
        Context ic = new InitialContext(props);
        ah  = (ArticleAccess) ic.lookup("ArticleAccessBean/remote");
        List<InfosArticle> articles = ah.rechercherLesArticlesParCategorie(categorie);
        Iterator i = articles.iterator();
        InfosArticle article;
        while (i.hasNext()) {
            article = (InfosArticle) i.next();
            afficherArticle(article.idArticle);
        }
    } catch (Throwable th) {
        System.out.println("Erreur dans rechercherArticlesParCategorie : " + th);
    }
}
```


Beans entités : exemple (11/12)

- Le client `ArticleClient.java` : la méthode `afficherTousLesArticles`

```
public void afficherTousLesArticles() {

    ArticleAccess ah;
    Properties props = new Properties();
    props.put("java.naming.factory.initial", "org.jnp.interfaces.NamingContextFactory");
    props.put("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.interfaces");
    props.put("java.naming.provider.url", "localhost:1099");
    try {
        Context ic = new InitialContext(props);
        ah  = (ArticleAccess) ic.lookup("ArticleAccessBean/remote");
        List<InfosArticle> articles = ah.rechercherTousLesArticles();
        Iterator i = articles.iterator();
        InfosArticle article;
        while (i.hasNext()) {
            article = (InfosArticle) i.next();
            afficherArticle(article.idArticle);
        }
    } catch (Throwable th) {
        System.out.println("Erreur dans rechercherArticlesParCategorie : " + th);
    }
}
```


Beans entités : exemple (12/12)

- Le client `ArticleClient.java` : la méthode `main`

```
public static void main(java.lang.String[] args) {

    ArticleClient a = new ArticleClient ();

    a.creerArticle("Les miserables", 21, "LIVRE");
    a.creerArticle("Celine Dion au stade de France", 120, "CD");
    a.creerArticle("Je l'aime a mourir", 28, "LIVRE");
    a.creerArticle("La mer", 38, "LIVRE");

    // Recherche de l'article 3
    System.out.println ("================================");
    a.afficherArticle(3);
    System.out.println ("================================");
    a.afficherTousLesArticles();
    System.out.println ("================================");

    // Recherche de la categorie
    a.afficherArticlesParCategorie("CD");
    System.out.println ("================================");

  }
}
```

Connexion avec une base de données MySQL (1/2)

- Créer une base de données (Exemple : JBossDB)
- Créer un fichier `mysql-ds.xml` (Le nom importe peu !)
- Copier ce fichier dans le répertoire de déploiement `$JBOSS_HOME$/server/default/deploy`.
- Télécharger le connecteur JDBC pour MySQL (fichier JAR) et copier dans le répertoire `$JBOSS_HOME$/server/default/lib`.

Connexion avec une base de données MySQL (2/2)

- Le fichier `mysql-ds.xml`

```
<?xml version="1.0" encoding="UTF-8"?>

<datasources>
    <local-tx-datasource>
        <jndi-name>MySQLDS</jndi-name>
        <connection-url>jdbc:mysql://localhost:3306/JBossDB</connection-url>
        <driver-class>com.mysql.jdbc.Driver</driver-class>
        <user-name>duvallet</user-name>
        <password>duvallet</password>
        <exception-sorter-class-name>
          org.jboss.resource.adapter.jdbc.vendor.MySQLExceptionSorter
        </exception-sorter-class-name>
        <!-- corresponding type-mapping in the standardjbosscmp-jdbc.xml (optional) -->
        <metadata>
            <type-mapping>mySQL</type-mapping>
        </metadata>
    </local-tx-datasource>
</datasources>
```


Les relations Un à Un (One To One)

- Une relation « One To One » est utilisée pour lier deux entités uniques indissociables.
- Ce type de relation peux-être *mappé* de trois manières dans la base de données :
  - en utilisant les mêmes valeurs pour les clés primaires des deux entités via l'annotation `@PrimaryKeyJoinColumn`.
  - en utilisant une clé étrangère d'un coté de la relation via l'annotation `@JoinColumn (name="identifiant", referencedColumnName="clé primaire dans l'autre table").`
  - en utilisant une table d'association de liens entre les deux entités via l'annotation `@JoinTable(name = "NomDeLaTableDeJointure", joinColumns = @JoinColumn (name="1ère clé étrangère", unique=true), inverseJoinColumns = @JoinColumns(name="2ième clé étrangère", unique=true)`

Les relations Un à Plusieurs (One To Many) et Plusieurs à Un (Many To One)

- Ces deux relations sont utilisées pour lier à une unique instance d'une entité A, un groupe d'instances d'une entité B.
- L'association « Many To One » est définie par l'annotation `@ManyToOne`
- Dans le cas d'une relation bidirectionnelle, l'autre coté doit utiliser l'annotation `@OneToMany`.
- Les attributs de ces deux annotations correspondent à ceux de l'annotation `@OneToOne`.

Les relations Plusieurs à Plusieurs (Many To Many)

- Ces relations sont utilisées pour lier des instances de deux entités entre elles.
- Exemple : un article appartient à éventuellement plusieurs catégories et une catégorie contient plusieurs articles.
- Il est nécessaire d'utiliser une table d'association pour mettre en œuvre cette association.