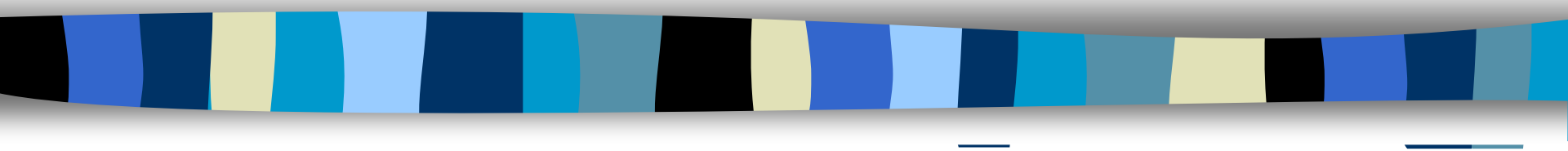
Java EE
Java Enterprise Edition



Mohamed SALLAMI

Au sommaire …

- JAVA EE (Rappel)
- Introduction aux EJB.
- JMS
- Persistance dans Java EE 6 et 7 (JPA, Hebernate)
- Introduction aux CDI

JAVA EE (RAPPEL)

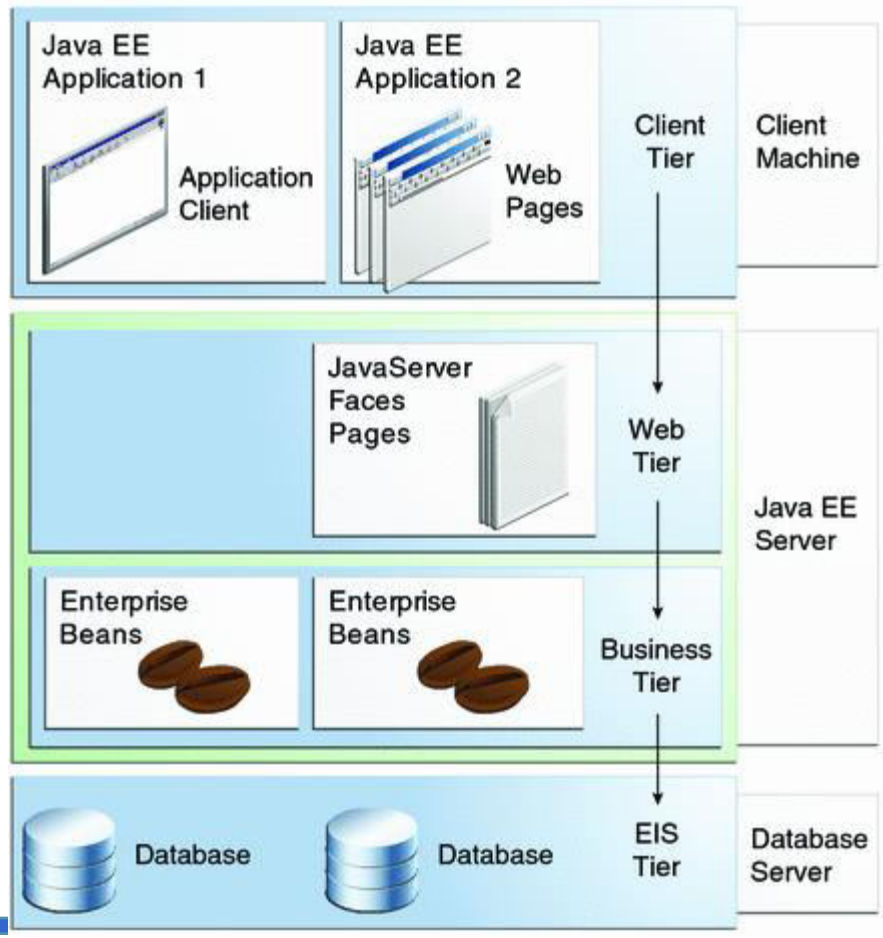
L'architecture Java EE

- Sorte d'architecture à 3 niveaux
- où la couche présentation est divisée en 2 :
  - Le client léger :
    - un navigateur Web
  - Un serveur Web
    - avec JSF, servlets.

Couche de présentation
Application standalone
Application web
Applets

Couche métier ; middleware
-> Traitements
(par des EJB, ou des JavaBeans)

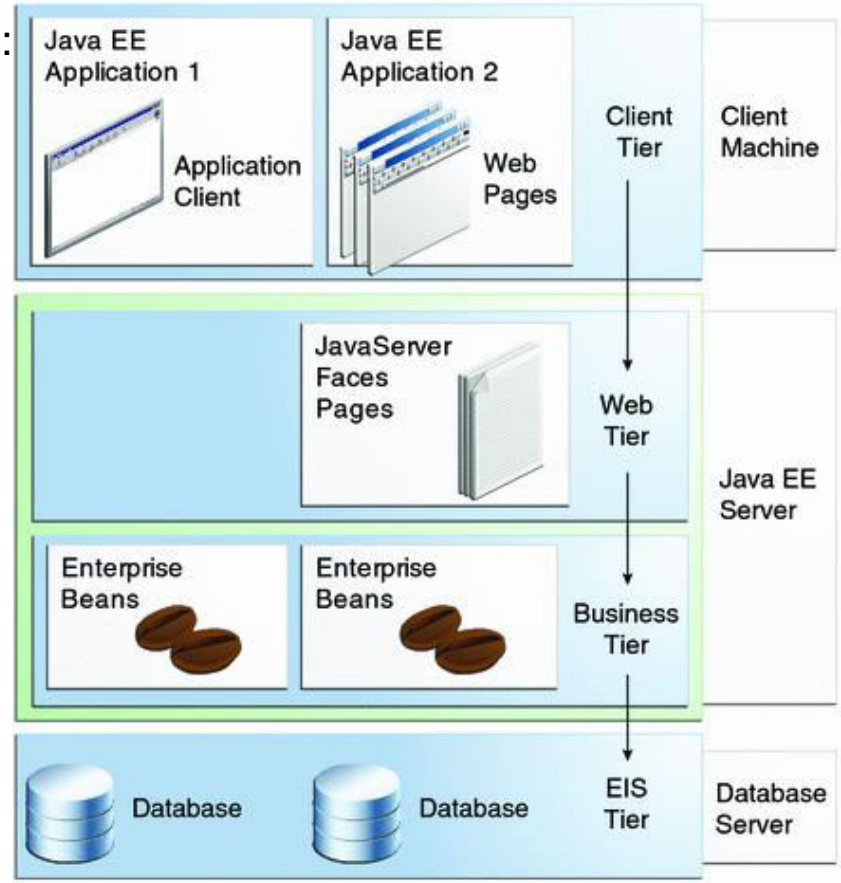
Couche de données
SGBD ; Stocke les données



Source: http://download.oracle.com/javaee/6/tutorial/doc/bnaay.html

Les conteneurs d'EJB et Web

- Les différents types de conteneurs :
  - Conteneur d'application cliente et d'applet :
    - la machine virtuelle Java
  - Conteneur Web :
    - pour l'exécution des servlet, JSF, etc.
  - Conteneur d'EJB :
    - composants métier



Source: http://download.oracle.com/javaee/6/tutorial/doc/bnaay.html

Java EE : les APIs

- Java EE comprend de très nombreuses API
  - Servlets, JSP, JSTL, JSF
  - JNDI,
  - JDBC,
  - EJB,
  - JMS
  - JavaMail
  - JPA
  - JTA
  - JAAS (Java Authentification and Authorization Service)
  - Java API for XML Parsing – JAXP
  - JAXB
  - Java RMI
  - …

Java EE 6 Web profile

- Java EE 6 propose ce qu'on appelle « un web profile »
  - Un sous-ensemble de java EE pour le développement d'application web simples
  - Dans ce web profile on propose :
  - Vue : JSP ou facelets/JSF2
  - Contrôleur web : Servlets ou web service
  - Composant métier : EJB type session + managed beans + classes java standard,
  - Accès aux données dans une BD via JPA2/EJB type entity

Introduction aux Enterprise Java Beans

Sommaire

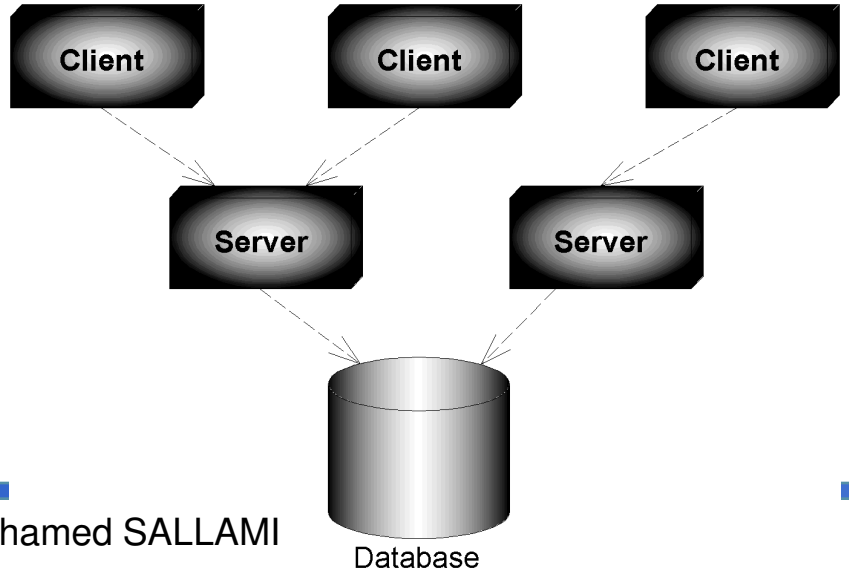
- Rappel générale sur Java EE
- Principes et concepts des EJB
- Complexités de EJB2
- Les apports de EJB3
- Les Annotations appliquées aux EJB3
- Conclusion

Les promesses des EJBs

- Enterprise JavaBeans
  - Standard industriel pour un modèle de composant logiciel distribué,
  - Permet d'implémenter des "objets métier" d'une manière propre et réutilisable,
- Questions :
  - De quoi a-t-on besoin lorsqu'on développe une application distribuée orientée objet ?
  - Qu'est-ce que les EJBs et qu'apportent-elles ?
  - Quels sont les acteurs dans l'écosystème EJB ?

Motivation des EJBs

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



Application distribuée

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

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

Le middleware s'occupe de tout ceci

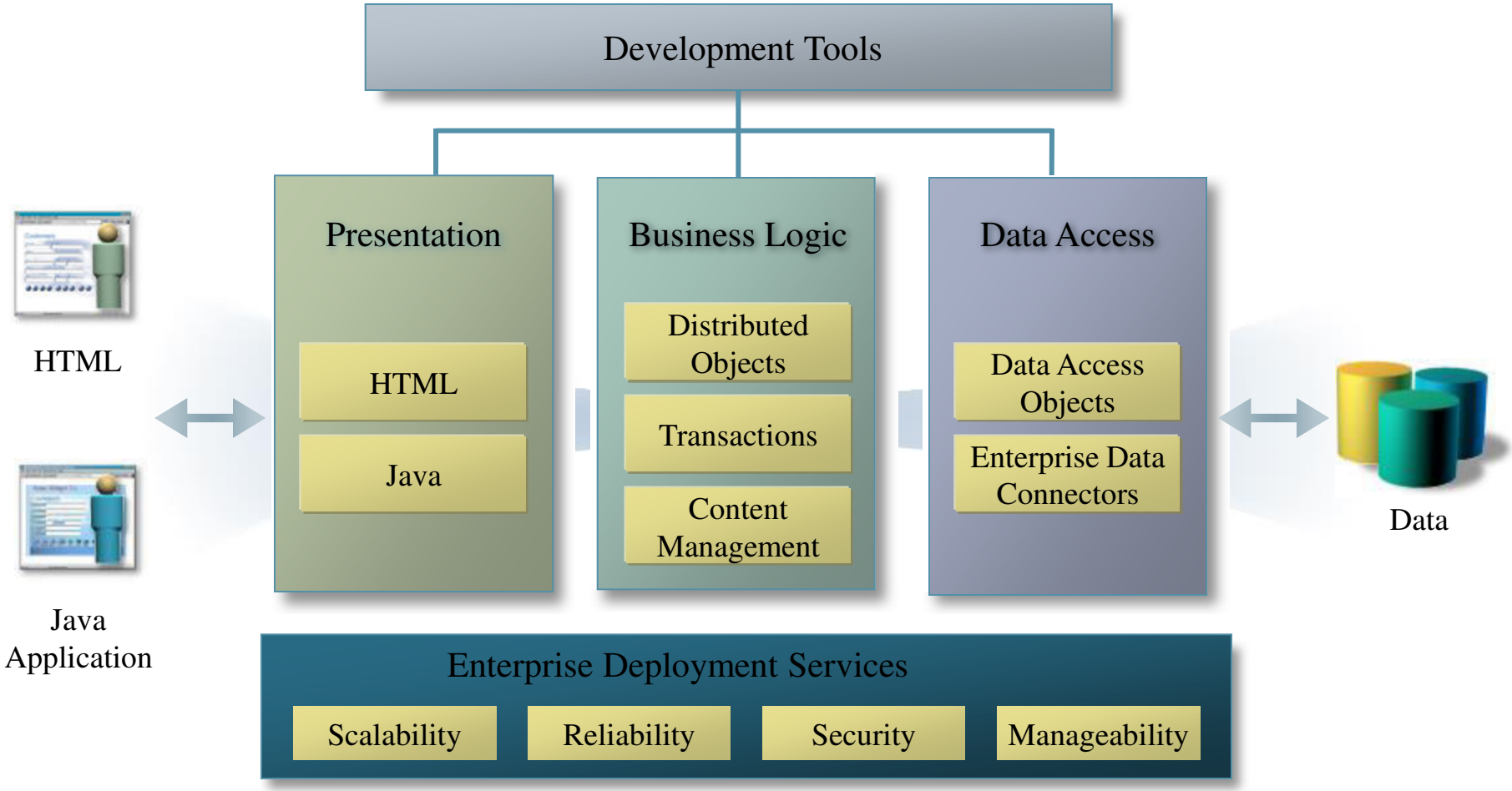
Rappel: un middleware est un logiciel qui crée un réseau d'échange d'informations entre différentes applications.

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

Serveur d'application

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

Serveur d'application



Encore mieux!

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

Composant logiciel réutilisable

- Une entreprise peut acheter un composant et l'intégrer avec des composants qu'elle a développé.
  - Par exemple, un composant qui sait gérer des prix.
  - On lui passe une liste de produits et il calcule le prix total.
  - Simple en apparence, car la gestion des prix peut devenir très complexe : remises, promotions, lots, clients privilégiés, règles complexes en fonction du pays, des taxes, etc…
- Ce composant répond à un besoin récurrent
  - Vente en ligne de matériel informatique,
  - Gestion des coûts sur une chaîne de production automobile,
  - Calcul des prix des expéditions par la poste,
  - Etc…

Quel intérêt?

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

Architectures de composants

- Plus de 50 serveurs d'applications ont vu le jour depuis une dizaine d'années,
- Au début, composants propriétaires uniquement.
  - Pas de cohabitation entre composants développés pour différents serveurs d'application
  - Dépendant d'un fabriquant une fois le choix effectué.
- Dur à avaler pour les développeurs java qui prônent la portabilité et l'ouverture !
- ➔ Nécessité de standardiser la notion de composants
  - Ensemble de définitions d'interfaces entre le serveur d'application et les composants
  - Ainsi n'importe quel composant peut tourner ou être recompilé sur n'importe quel serveur

Un tel standard s'appelle une architecture de composants

EJB?

- **Enterprise JavaBeans** (EJB) est une architecture de composants logiciels côté serveur pour la plateforme de développement JEE.
- Cette architecture propose un cadre pour créer des composants distribués écrit en Java hébergés au sein d'un serveur applicatif permettant
  - de représenter des données (EJB dit *entité*)
  - de proposer des services avec ou sans conservation d'état entre les appels (EJB dit *session*),
  - d'accomplir des tâches de manière asynchrone (EJB dit *message*).
- EJB signifie deux choses :
  1. Une spécification
  2. Un ensemble d'interfaces

Quels sont les principaux avantages des EJB ?

Les EJB définissent un standard JavaBean pour faciliter la réutilisation et l'interopérabilité des composants middleware. Et par conséquent : leur développement et leur intégration. Sans compter une grande facilité de configuration côté programmation.

Quels sont leurs grands domaines d'application ?

- Les EJB sont notamment utilisés dans:
  - Les domaines des transactions sécurisées
  - Les domaines du stockage d'objets java.
    - Côté gestion des transactions, les EJB simplifient les procédures. Grâce à eux, il n 'est plus nécessaire de programmer ni de maintenir le code de gestion des transactions.
    - Le développeur n'a plus qu'à définir les objets et les méthodes appropriés à une transaction particulière.
    - Et comme les EJB s'appuient sur Java, le composant peut être implémenté sur n'importe quelle plate-forme ou système d'exploitation intégrant ce langage

EJB2→ EJB3

- Critique des Enterprise JavaBeans (EJB) par plusieurs développeurs à cause de ses complexités.
- Gavin King, leader du Framework Hibernate et membre du groupe d'experts EJB 3 a dévoilé le premier brouillon de cette spécification EJB3
- Cette nouvelle version des EJB apporte des modifications notables dans le domaine du modèle de développement et intègre de nombreuses nouveautés notamment en ce qui concerne les EJBs entity tellement décrites dans les versions 2.X.

Historique

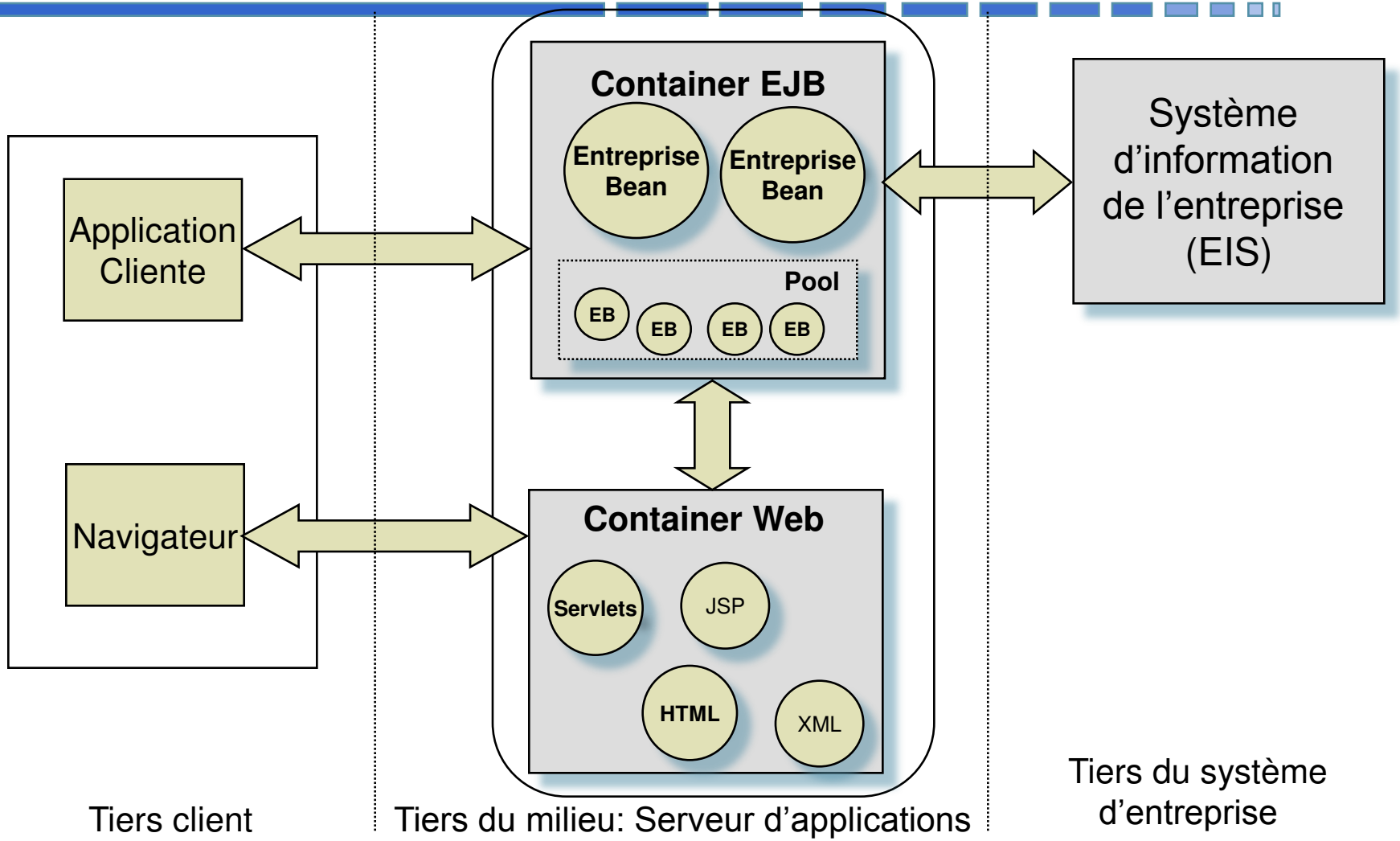
- EJB 1.1 publié en décembre 1999, intégré dans J2EE 1.2 :
  - Session beans (stateless/stateful)
  - Entity Beans
  - Interface Remote uniquement
- EJB 2.0 publié en septembre 2001, intégré à J2EE 1.3 :
  - Message-Driven Beans
  - Entity 2.x reposant sur EJB QL
  - Interface Local pour améliorer les performances des appels dans la même JVM
- EJB 2.1 publié en novembre 2003, intégré à J2EE 1.4 :
  - EJB Timer Service
  - EJB Web Service Endpoints via JAX-RPC
  - Amélioration du langage EJB QL
- EJB 3.0, publié en mai 2006,intégré à Java EE 5 :
  - utilisation de POJO, plus d'interface Home
  - utilisation des annotations, le descripteur de déploiement est optionnel
  - utilisation de JPA pour les beans de type entity
- EJB 3.1, publié en décembre 2009, intégré à Java EE 6
  - EJB lite dans Web profile
- EJB 3.2, publié en juin 2013, intégré à Java EE 7

EJB (PRINCIPES ET CONCEPTS)

Objectifs des EJB

- Offrir une solution Web Middleware
- Exécution dans un *Container*
- Architecture n tiers
- Faciliter la tache des développeurs en leur permettant de se concentrer sur la logique métier.

Architecture Java EE

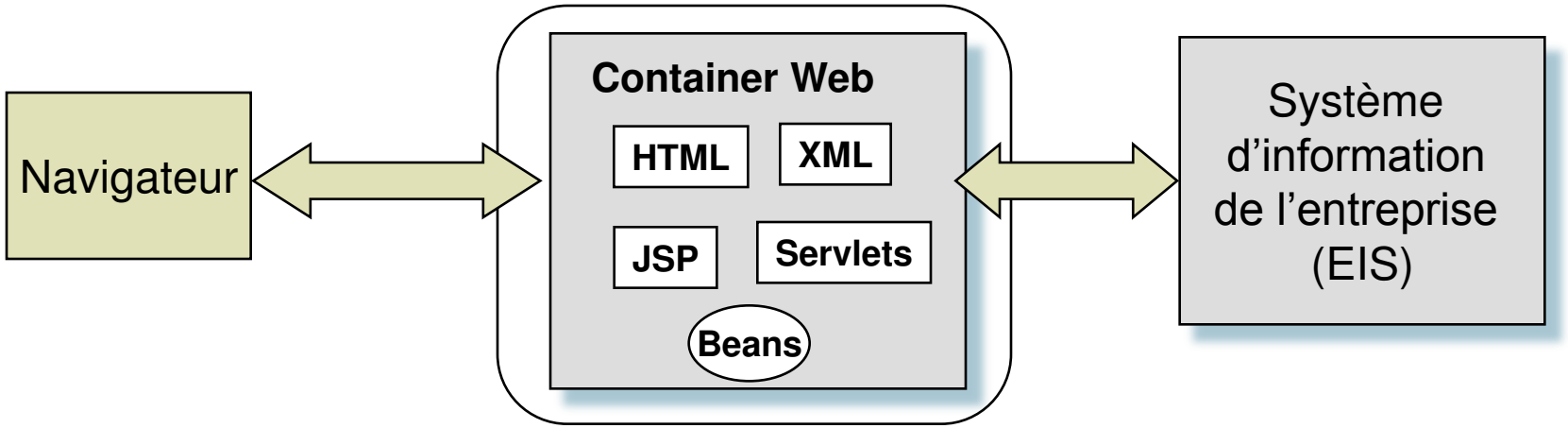


Serveurs d'applications

- Infrastructures d'exécution de composants et d'applications
- Dépôt de composants qui peuvent être utilisés sur demande.
- Outils :
  - Basés Java EE:
    - WebSphere (IBM), Web Logic (BEA), Sun Java System Application Server 7.x, JBoss, JOnAS, OpenEJB, EasyBeans …
  - Non basés J2EE :
    - PHP, Cold Fusion,…
  - Microsoft .NET

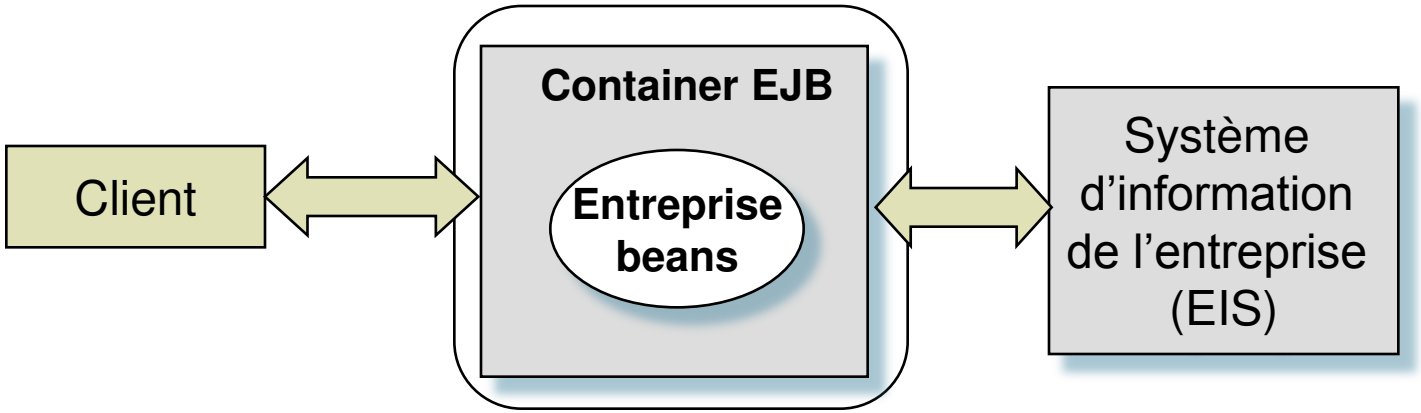
Architecture Java EE Web

- Architecture Java EE web :
  - Client : Navigateur
  - Serveur d'applications web (Container web):
    - Servlets , JSP, HTML, XML et Java Beans



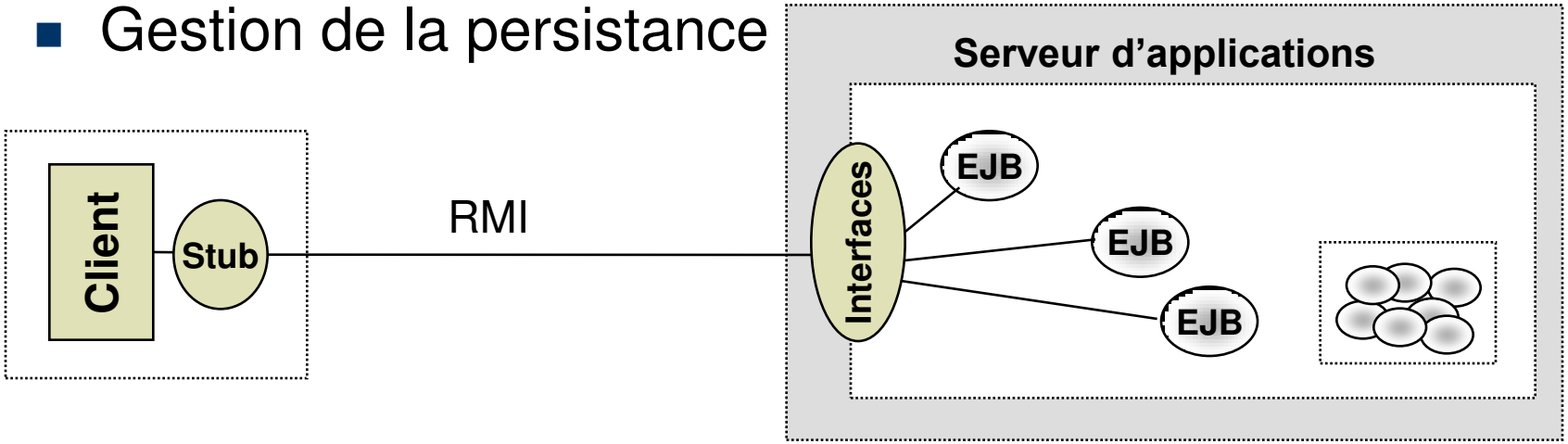
Architecture Java EE entreprise

- Utilisation de Beans d'entreprise à travers des interface standardisées
- Invocations par RMI
- Interaction avec systèmes existants avec :
  - JDBC, JNDI , JTS , JMS, RMI, …

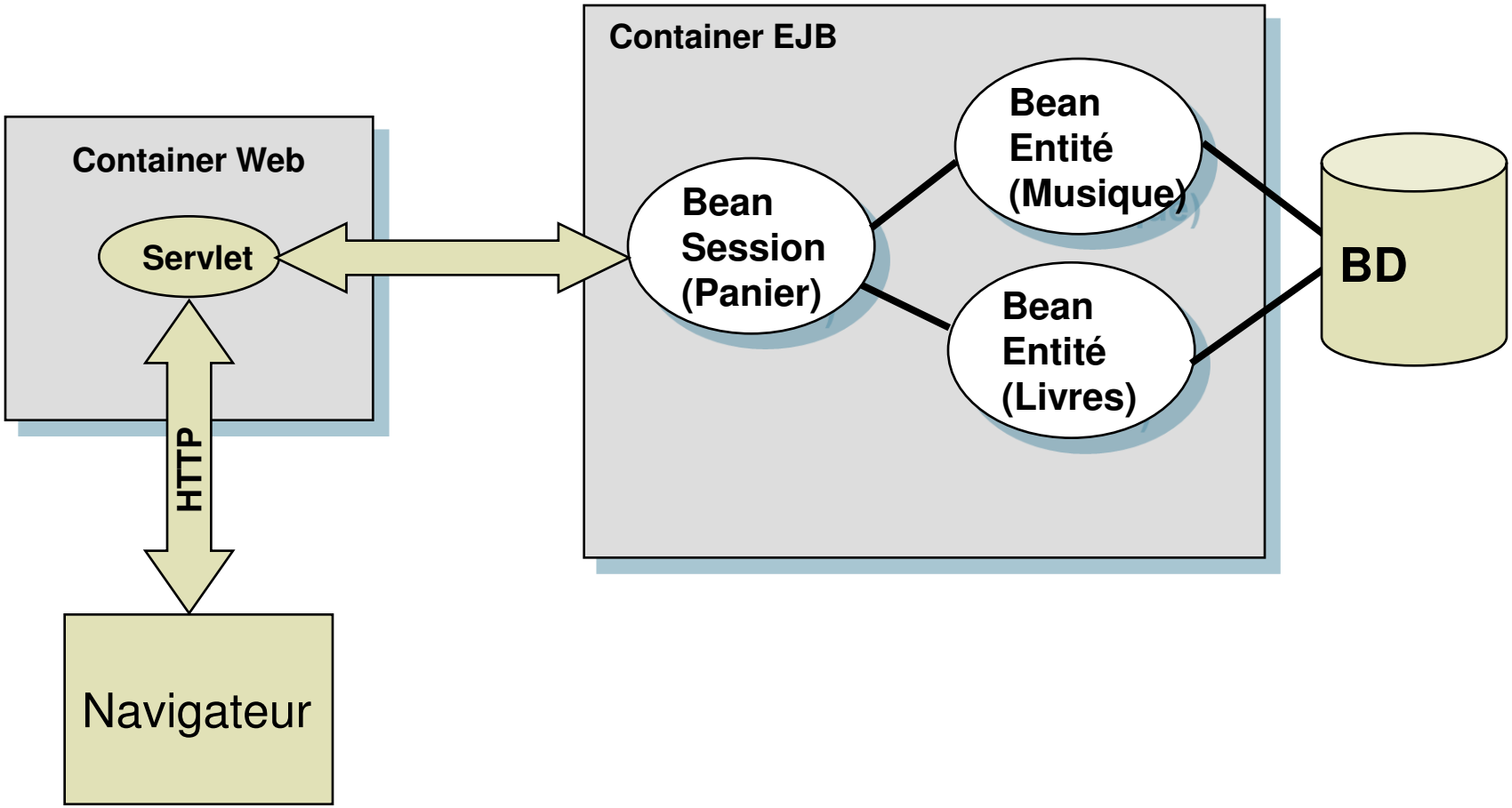


Rôle du container

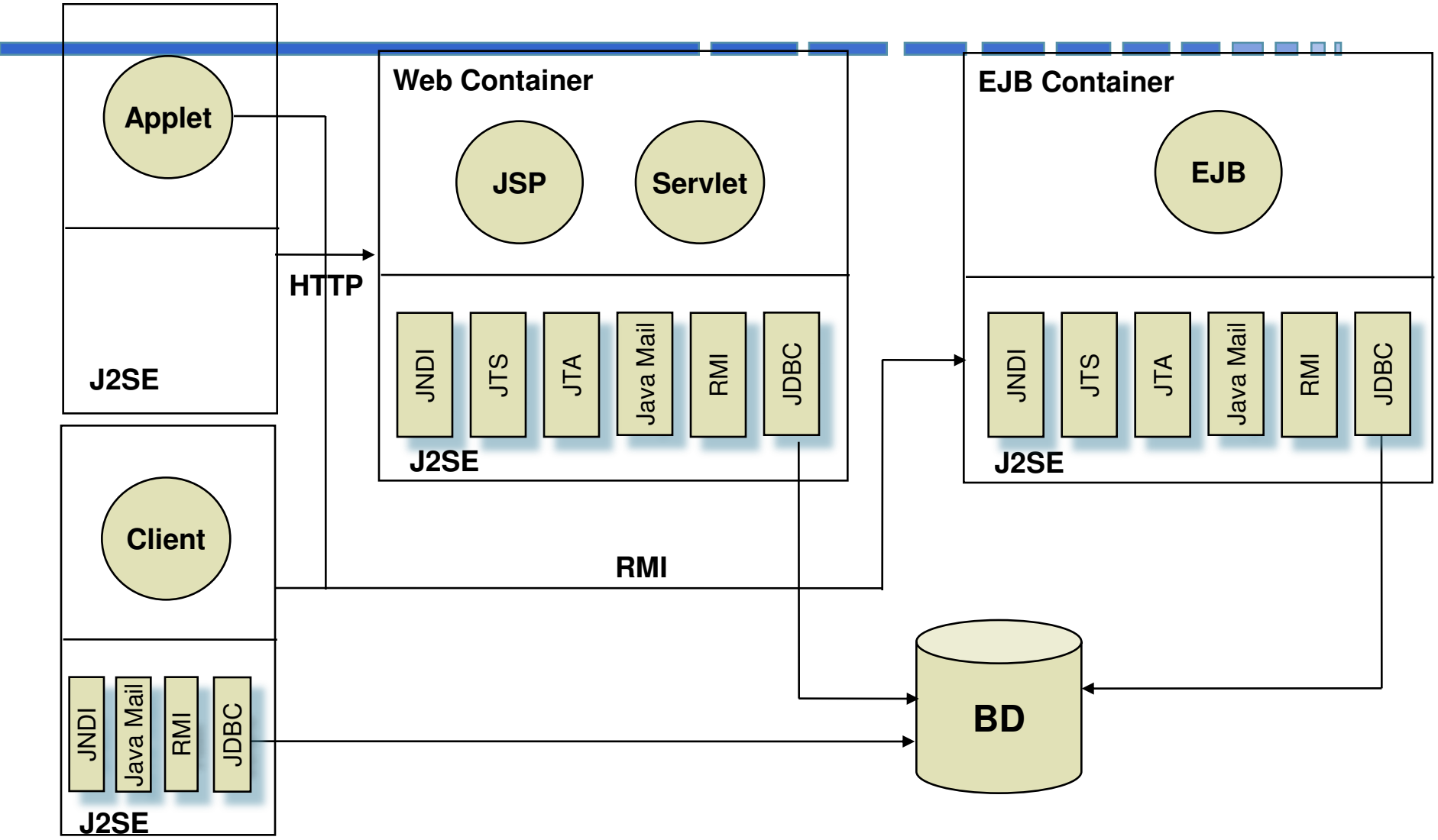
- Environnement d'exécution des EJB
- Connexion entre clients et EJB
- Injection de dépendance
- Gestion de la concurrence et des transactions
- Gestion du cycle de vies des Beans
- Gestion de la persistance



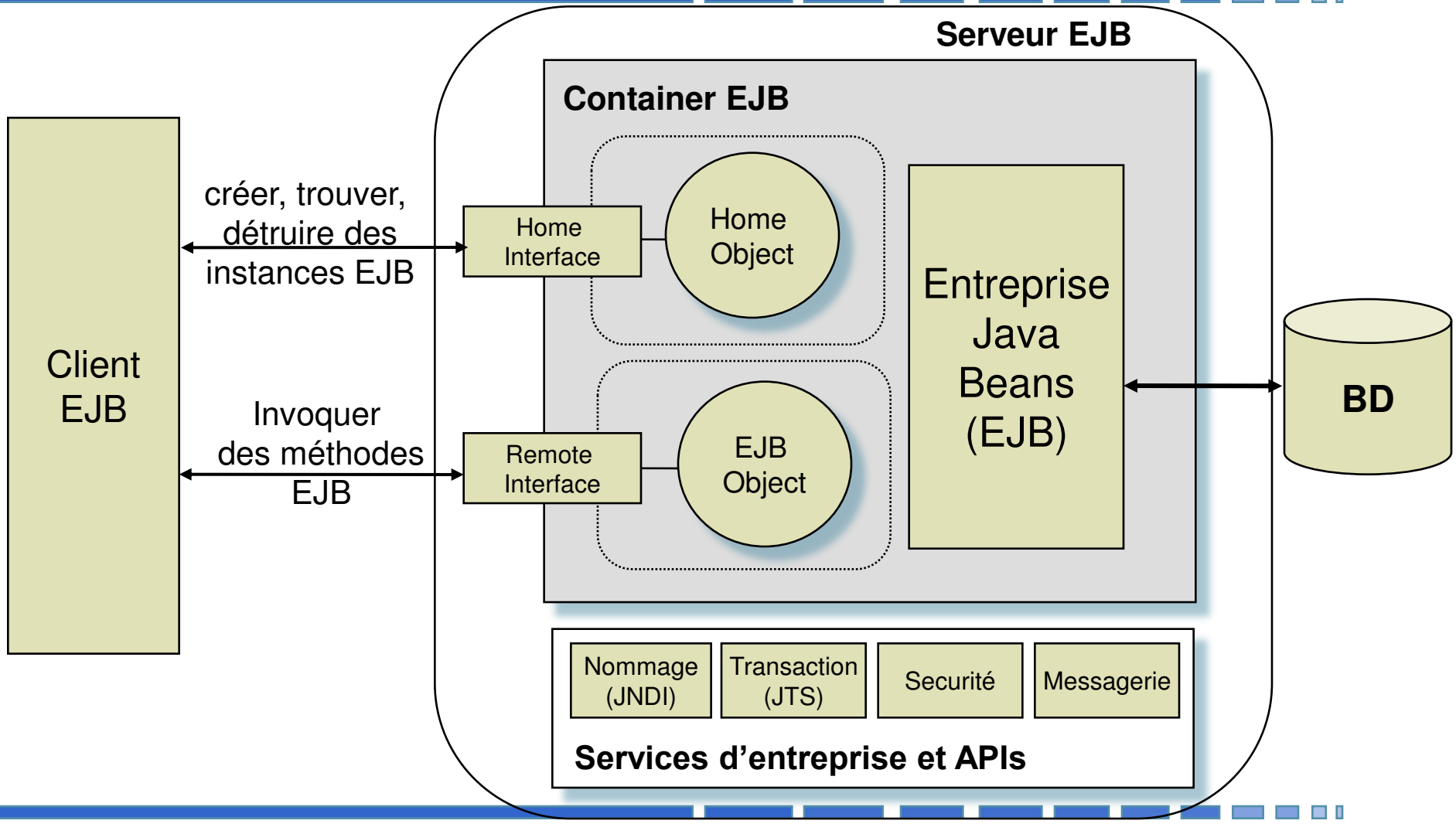
Application web



Architecture : conteneurs Java EE



Architecture : conteneur EJB Java EE



Enterprise Bean

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

Enterprise Bean

- Le client d'un Bean peut être
  - Une servlet
  - Une applet
  - Une application classique
  - Un autre Bean
- On peut décomposer une application en un graphe de tâches/sous-tâches
- Modèle flexible, extensible…

Les Beans : 3 types

- Bean session :
  - Représente une session d'un client unique
  - Non persistante
  - Modèlise un traitement (Logique métier)
  - Corresponds à des *verbes*, à des *actions*
  - *Représenté par une classe Java et une interface qui expose certaines méthodes.*
  - Souvent clients d'autres Beans
  - Ex : gestion de compte bancaire, affichage de catalogue de produit, vérifieur de données bancaires, gestionnaire de prix…
- Bean entité :
  - Représente des données persistantes
  - Plusieurs clients peuvent accéder au même bean: Un client par instance.
  - Ex: un Bean Personne, un Bean compte bancaire, un Bean produit, un Bean commande.
- Bean message ou Message Driven Bean (MDB):
  - Communique par échange de messages (JMS)
  - Basé sur des middlewares orientés messages
  - Ex: message pour déclencher des transactions boursières, des autorisations d'achat par CB

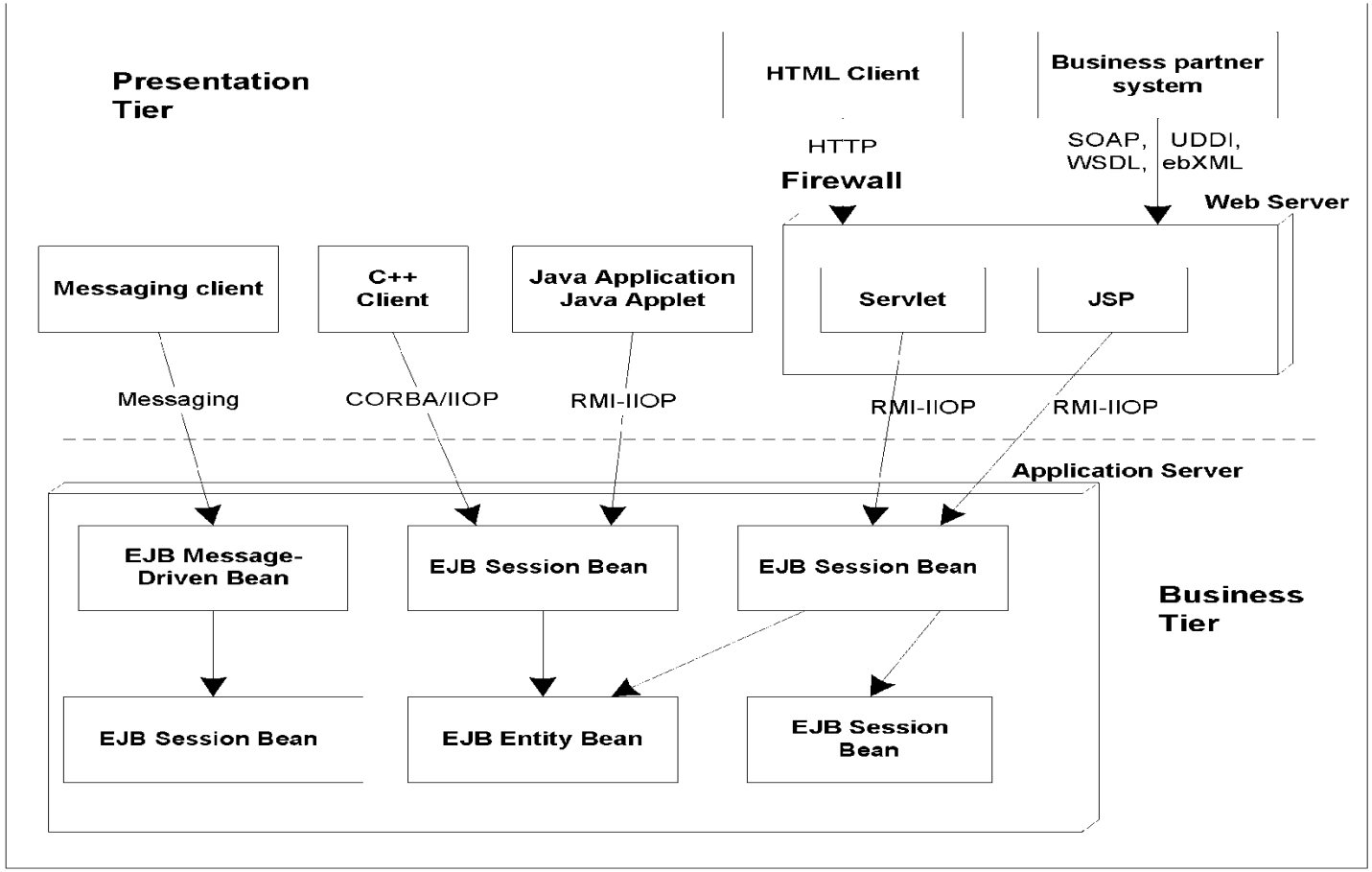
3 types de Session Bean

- Bean sans état (*stateless*)
  - Pour traiter les requêtes de plusieurs clients,
  - sans garder un état entre les différentes requêtes
  - Exemple : obtenir la liste de tous les produits
- Bean avec état (*stateful*)
  - Pour tenir une conversation avec un seul client,
  - en gardant un état entre les requêtes
  - Exemple : remplir le caddy d'un client avant de lancer la commande (le caddy est rempli en cliquant sur les différentes pages des produits)
- Bean singleton
  - Garantie de n'avoir qu'une seule instance du bean dans tout le serveur d'application
  - Supporte les accès concurrents (configurable)
  - Exemple : bean qui « cache » une liste de pays, utilisé par les classes de l'application pour éviter d'interroger la BD

Exemple de Session/Entity bean

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

Clients interagissant avec un serveur à base d'EJBs



Le développement d'un EJB

- Le cycle de développement d'un EJB comprend :
  - la création des interfaces et des classes du bean
  - le packaging du bean sous forme de fichier archive jar
  - le déploiement du bean dans un serveur d'EJB
  - le test du bean

UN PEU D'IMPLÉMENTATION AVEC EJB 2.X

EJB 2.x

- EJB 3.0 simplifie la tâche du développeur en cachant des détails d'implémentation
- L'étude de EJB 2.x permet de comprendre comment fonctionnent les EJB
- Pour chaque EJB écrit par le développeur, le serveur d'application crée un objet (*EJB Object*) qui contient le code qui va permettre au serveur d'intercepter les appels de méthode de l'EJB

Rôle de l'EJB Object

- Les clients n'invoquent jamais directement les méthodes de la classe du Bean
- Les appels de méthodes sont en fait envoyés à l'EJB Object
- Une fois les traitements effectués pour les transactions, sécurité,.. le container appelle les méthodes de la classe du bean

Interfaces

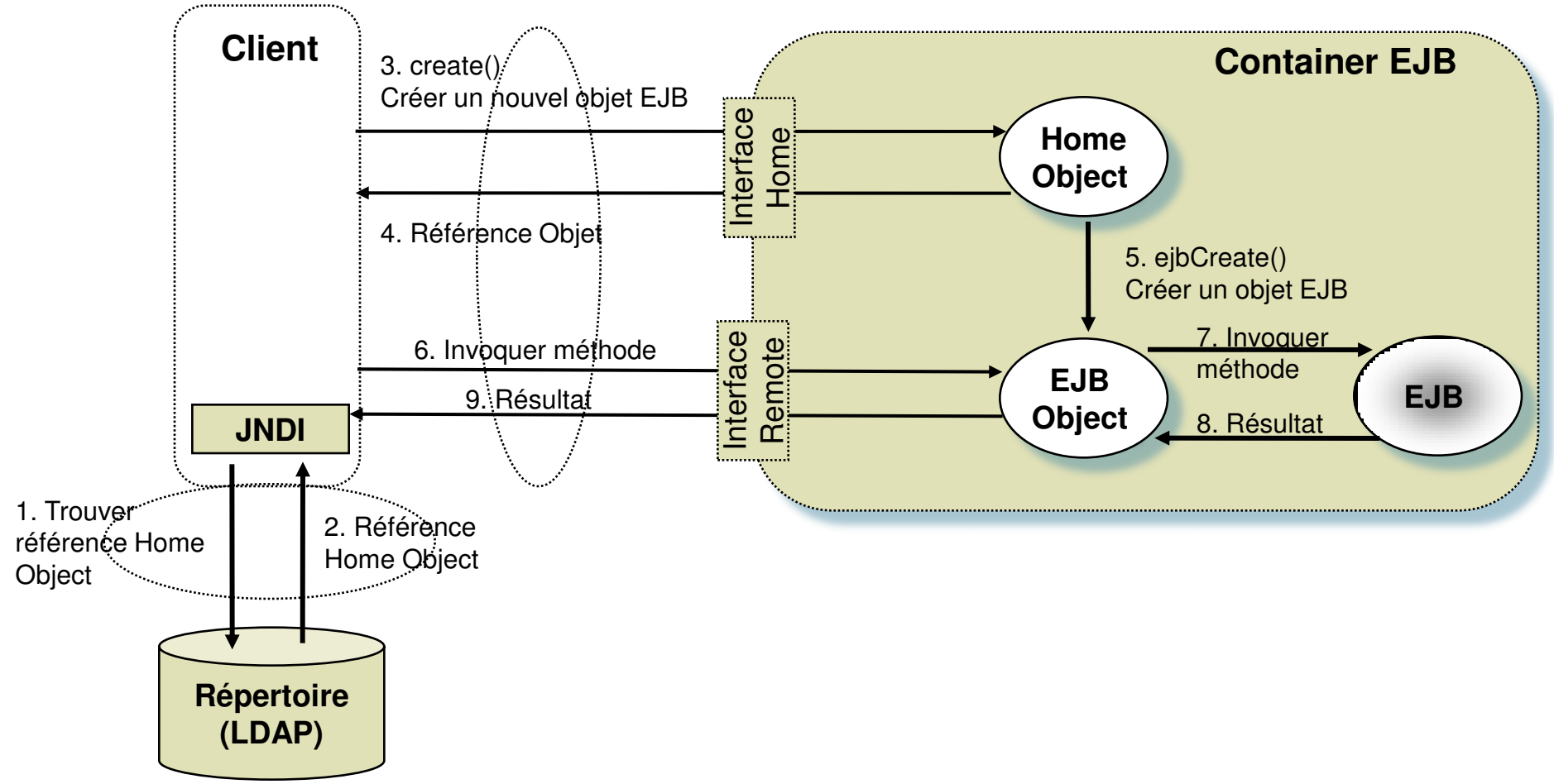
- Pour chaque EJB session, le développeur doit fournir une (ou 2) interface qui indique les méthodes de l'EJB que les clients de l'EJB pourront appeler
- Les autres méthodes de l'EJB servent au bon fonctionnement de l'EJB
- Un EJB session peut avoir une interface locale et une interface distante

Création d'un bean

- La création d'un bean nécessite la création d'au minimum deux interfaces et une classe pour respecter les spécifications de Sun.
  - **L'interface remote:** permet de définir l'ensemble des services fournis par le bean. Cette interface étend l'interface EJBObject
  - **L'interface home:** permet de définir l'ensemble des services qui vont permettre la gestion du cycle de vie du bean. Cette interface étend l'interface EJBHome.
  - **La classe du bean:** contient l'implémentation des traitements du bean. Cette classe implémente les méthodes déclarées dans les interfaces home et remote.

L'accès aux fonctionnalités du bean se fait obligatoirement par les méthodes définies dans les interfaces home et remote.

Diagramme d'exécution des Beans session



Interface locale

- Si l'EJB n'a qu'une seule interface locale, il ne peut être utilisé que par les classes qui sont dans le même container
- Le développeur **peut ne fournir aucune interface** ; en ce cas, une interface **locale** est automatiquement créée, qui contient toutes les méthodes publiques de l'EJB

Interface distante

- Indispensable si l'EJB peut être utilisé par des classes qui ne sont pas dans le même container (application distribuée)
- Pour manipuler un EJB à travers une interface distante, le serveur d'application utilisera RMI-IIOP, ce qui implique
  - des performances moins bonnes
  - les paramètres et les valeurs de retour sont transmis par recopie des valeurs (références pour un appel local)

Conclusion

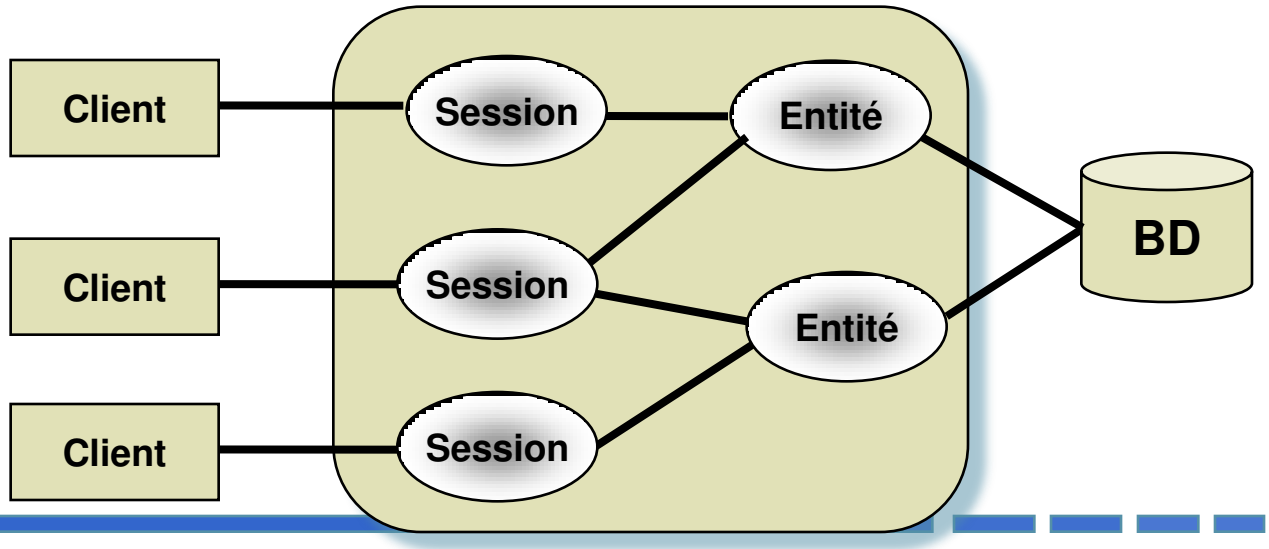
- Favoriser les interfaces locales
- Ne jamais utiliser d'interfaces distantes si les EJBs et leurs clients sont dans le même container

Les EJB session

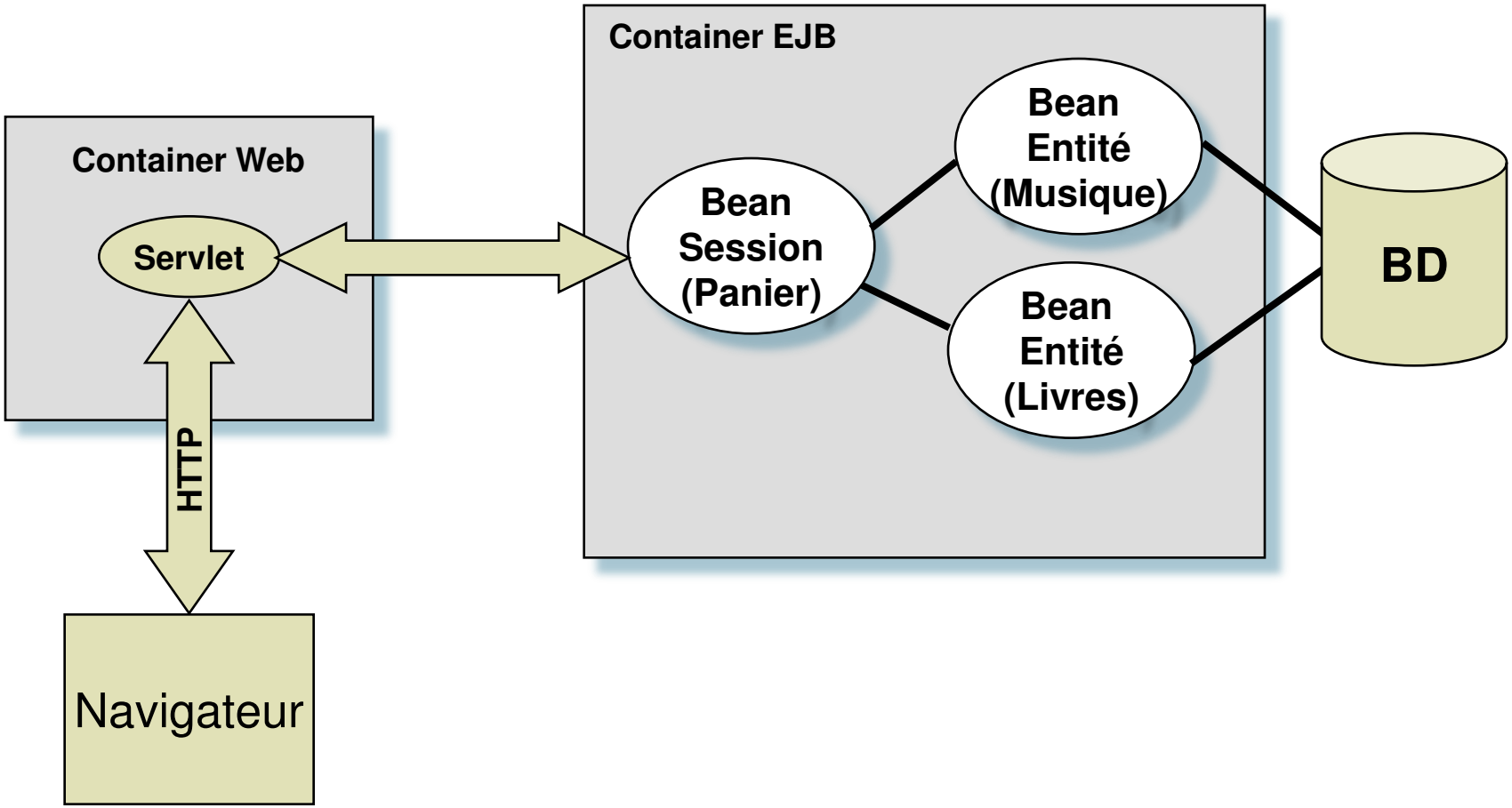
- Un EJB session est un EJB de service dont la durée de vie correspond à un échange avec un client. Ils contiennent les règles métiers de l'application.
  - Une session interactive
  - Un seul client à la fois
- Deux types :
  - Stateless (sans état): peut traiter plusieurs clients successivement pendant la durée de la session.
    - Ex: Virement d'un fond entre deux comptes.
  - Statefull (avec état): un seul client qui peut l'invoquer successivement
    - Ex: Panier des achats.

Les EJB Entité

- Ces EJB permettent de représenter et de gérer des données enregistrées dans une base de données. Ils implémentent l'interface EntityBean.
  - Ex: Une ligne dans une table, un client, un article.
- Les attributs du Bean sont stockés dans un support de mémoire persistante (BD, fichier, …)



Application web



Bean Entité

- Possède un identifiant unique
- Partageable
- Peut participer à une transaction
- Peut avoir des relations avec d'autres Beans
- Il faut assurer la correspondance Bean-table:
  - Ex: Attribut du Bean - Colonne d'une table de BD
- Implante des opérations de recherche, insertion, destruction et modification de données.

La Persistance

- Les beans entités assurent la persistance des données en représentant tout au partie d'une table ou d'une vue.
- Il existe deux types de bean entité :
  - Bean entité CMP (container-managed persistence)
    - c'est le conteneur d'EJB qui assure la persistance des données grâce aux paramètres fournis dans le descripteur de déploiement du bean. Il se charge de toute la logique des traitements de synchronisation entre les données du bean et les données dans la base de données.
  - Bean entité BMP (bean-managed persistence)
    - Il assure lui même la persistance des données grâce à du code inclus dans les méthodes du bean.
- Plusieurs clients peuvent accéder simultanément à un même EJB entity. La gestion des transactions et des accès concurrents est assurée par le conteneur.
- Le choix entre les deux est décrit à l'aide de l'outil de déploiement

Le déploiement des EJB 1/2

- Un EJB doit être déployé sous forme d'une archive jar qui doit contenir un fichier qui est le descripteur de déploiement et toutes les classes qui composent chaque EJB (interfaces home et remote, les classes qui implémentent ces interfaces et toutes les autres classes nécessaires aux EJB).
- Une archive ne doit contenir qu'un seul descripteur de déploiement pour tous les EJB de l'archive. Ce fichier au format XML doit obligatoirement être nommé ejb-jar.xml.
- L'archive doit contenir un répertoire META-INF (attention au respect de la casse) qui contiendra lui même le descripteur de déploiement.
- Le reste de l'archive doit contenir les fichiers .class avec toute l'arborescence des répertoires des packages.
- Le jar des EJB peut être inclus dans un fichier de type EAR.

Le déploiement des EJB 2/2

- Le descripteur de déploiement
  - un fichier au format XML qui permet de fournir au conteneur des informations sur les beans à déployer.
  - Le contenu de ce fichier dépend du type de beans à déployer.
- La mise en package des beans
  - Une fois toutes les classes et le fichier de déploiement écrit, il faut les rassembler dans une archive .jar afin de pouvoir les déployer dans le conteneur.

Clients EJB

- Peut être une Servlet, une JSP, une application Java ou RMI.
- Localise l'interface Home (via JNDI) qui contient le Bean désiré.
- Reçoit une référence à l'objet Home Object qui lui permet de communiquer avec le Bean.
- Utilise Home Object pour créer (et détruire) une instance du bean
- Utilise Remote Object pour invoquer les méthodes métier.

Clients EJB

- Trouver la référence de l'interface *Home*
  - Utiliser JNDI pour la trouver
  - Utiliser le contexte pour la recherche par JNDI
  - Utiliser l' interface Home pour créer un objet bean

Des EJB2 aux EJB3

Quelles sont les complexités des EJB2 ?

Les complexités de EJB 2.x (1)

- EJBs sont contre-productifs

  - 1 bean a des dépendences avec *EJBLocalObject, EJBLocalHome, EJBHome, EJBObject, ejb-jar.xml* …

  - Les outils pour masquer la complexité ne sont pas forcément à la hauteur

  - Difficile à tester, mise au point,…

- Nécessité d'avoir une connaissance détaillée à propos de JNDI pour réussir une application EJB.

Les complexités de EJB 2.x (2)

- Les EJB components ne sont pas totalement orientés objet

  → les classes persistantes doivent hériter de certaines classes fournies par le framework

Les complexités de EJB 2.x (3)

- Verbosité : beaucoup de fichiers à écrire
- Compléxité des descripteurs de déploiement (EJB).
- Difficulté d'utiliser les classes persistantes en dehors du container en particulier pour les tests
- Le framework ne peut pas prendre en charge la persistance d'objets des classes Java ordinaire (POJO)

Les complexités de EJB 2.x (4)

- Certains containers nécessitent une génération de stubs (JOnAS,..)
- D'autres génèrent les proxys automatiquement (JBoss, Geronimo…)
- Des containers génèrent des finders automatiquement (JBoss)
- Le développement multi-container est « complexe »
- Mais ne pas en tenir compte est aussi un risque

Les complexités de EJB 2.x (5)

Donc:

EJB 2.1 aujourd'hui est puissant, mais relativement complexe à utiliser!

Des EJB2 aux EJB3 (2)

A-t-on changé les choses avec EJB3 ?

Apports des EJB3 (1)

- Simplifier le développement
- Simplifier l'architecture des EJBs en réduisant sa complexité de point de vue développeur
- Génération automatique d'interface possible
- Pas de JNDI pour le développeur ou le client

Apports des EJB3 (2)

- Simplification des composants par l'utilisation des POJOs / JavaBeans
- Elimination des interfaces *EJBObject, EJBLocalObject, Remote, Home, …*
- Simplification des CMPs

Apports des EJB3 (3)

- Améliorer la *testabilité* hors du container
- Utilisation des *metadata* pour simplifier la maintenance des composants
- Simples annotations :
  - Utilisation systématique des annotations pour simplifier la programmation.
  - Elimination de descripteur de déploiement
  - Génération des interfaces dont on a besoin

Compatibilité et migration

- Une application EJB 2.1 doit fonctionner sans regression dans un conteneur EJB 3.0

- Migration vers EJB 3.0
  - Client EJB 3.0 & Composant EJB 2.1
  - Client EJB 2.1 & Composant EJB 3.0

Apports des EJB3 (5)

EJB 3 définit un standard de persistance qui ne cible que le relationnel à l'exclusion des autres formes de données (XML, mainframe) et qui ne peut s'exécuter que dans le contexte d'un container.

EJB3 : POJO (1)

- *POJO : Plain Old Java Objects* (bons vieux objets Java tous simples)
  - Cet acronyme est principalement utilisé pour faire référence à la simplicité d'utilisation d'un Objet Java en comparaison avec la lourdeur d'utilisation d'un composant EJB. Ainsi, un objet POJO n'implémente pas d'interface spécifique à un framework comme c'est le cas par exemple pour un composant EJB.
  - Le terme a été inventé par Martin Fowler, Rebecca Parsons et Josh MacKenzie en septembre 2000. « *Nous nous sommes demandés pourquoi tout le monde était autant contre l'utilisation d'objets simples dans leurs systèmes et nous avons conclu que c'était parce que ces objets simples n'avaient pas un nom sympa. Alors, on leur en a donné un et ça a pris tout de suite.* ».

EJB3 : POJO (2)

- Utiliser un modèle objet simple (facile à développer, centré sur l'applicatif plus que sur le technique) et sans dépendances avec telle API ou tel framework.

- Dans le monde Java, le modèle EJB3 ou les deux frameworks de persistance :
  - Hibernate et TOPlink sont les trois solutions les plus utilisées pour assurer la persistance des concepts métier implémentés sous forme de POJO

EJB3 : POJO (3)

EntityManager:

- EntityManager est un point de contrôle (“home”) pour les opérations sur les EJB entité

  1- Méthodes de cycle de vie :
  - Persist, remove, merge, flush, refresh, etc

  2- Méthodes de requêtage pré-définies (find)

EJB3 : POJO(4)

3- Factory pour des objets Query

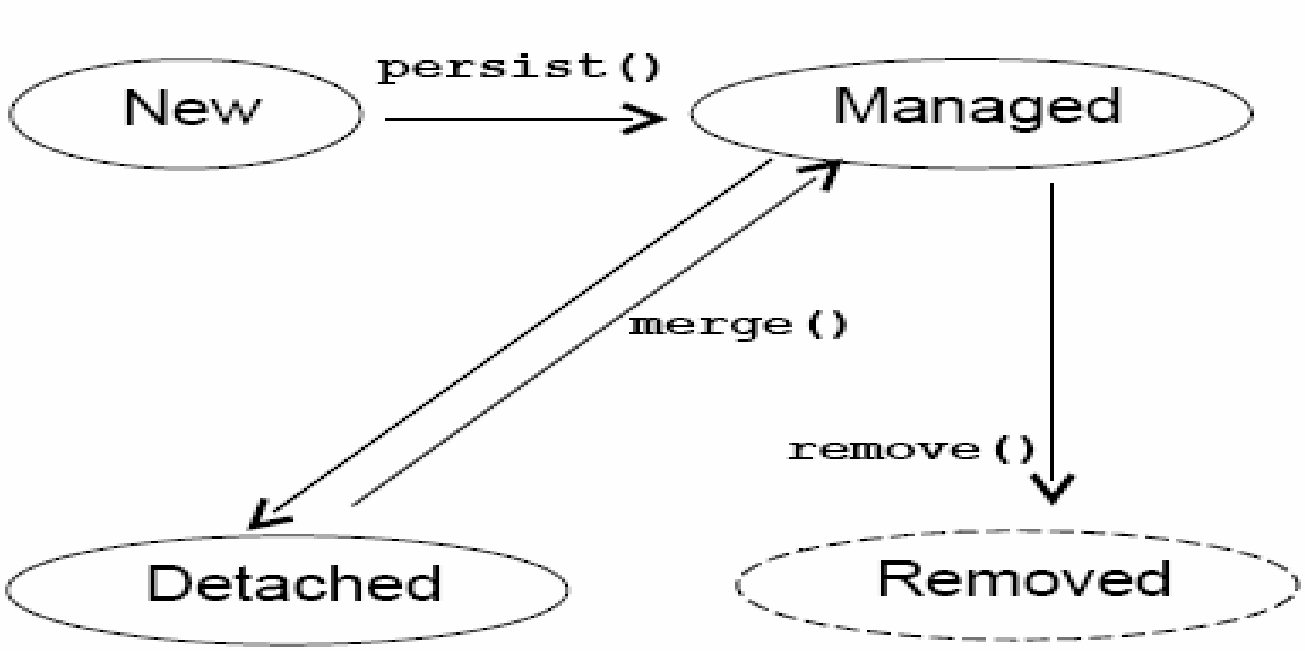
- Requêtes (nommées) statiques et dynamiques
- Requêtes EJB-QL et SQL

4- Gestion du contexte de persistance

- Réunion des identités persistantes
- Contexte étendu de persistance (transactions multiples)

EJB3 : POJO (3)

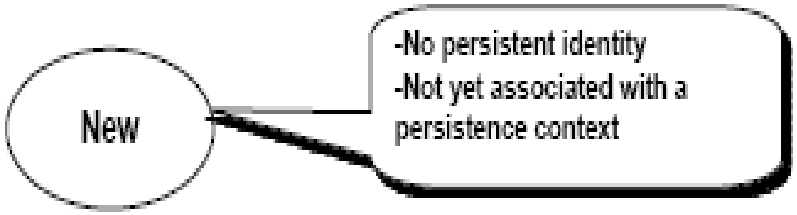
Cycle de vie



POJO: cycle de vie

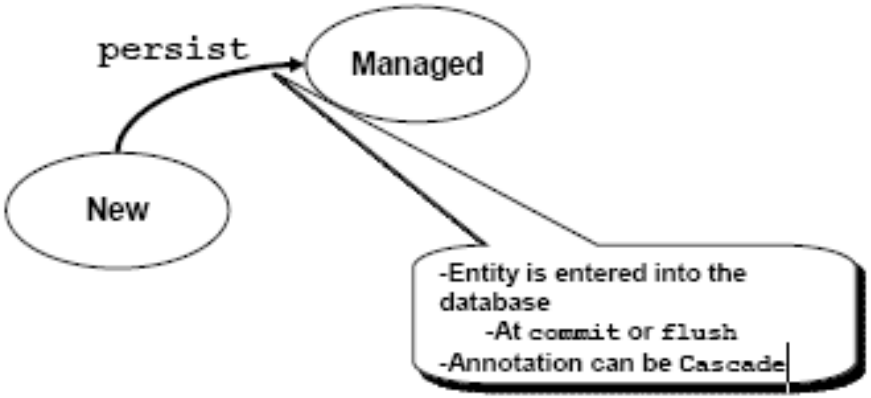
- **New**

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



- **Managed**

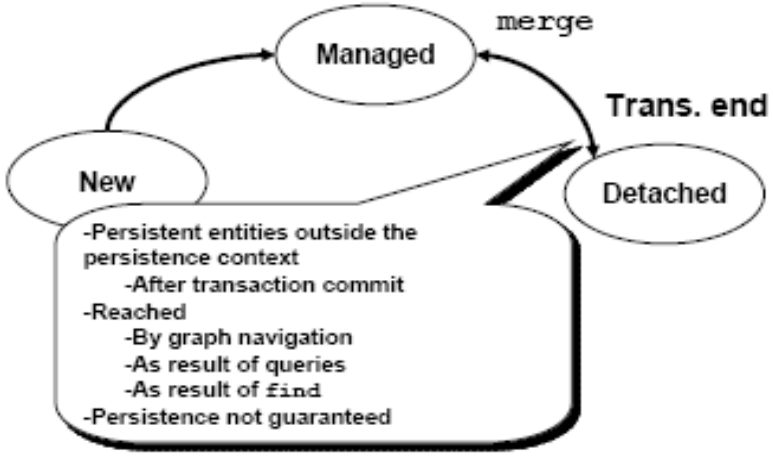
- Identité de persistance
- Géré par le conteneur
- Synchronisation SGBDR
- Gestion des relations



POJO: cycle de vie

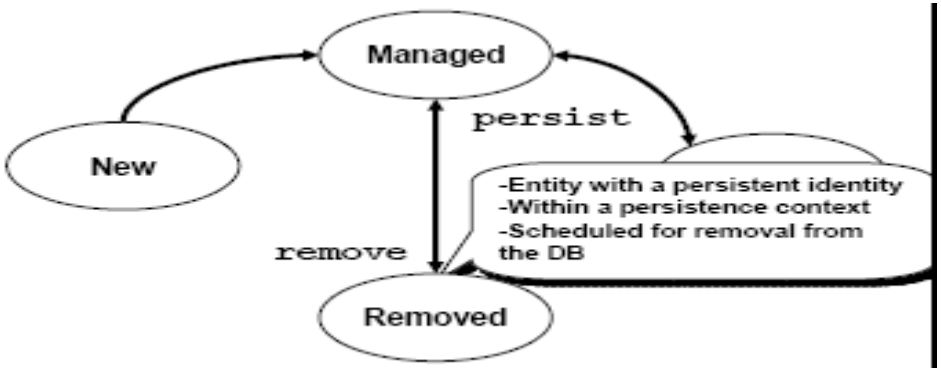
- **Detached**

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



- **Removed**

- Identité de persistance
- Suppression du bean



Annotations (1)

- Introduites dans JDK1.5 (JSR 175)
- Rendent les descripteurs de déploiement facultatifs
- Moins de verbosité pour les cas simples
- Méta-données proche du code
- gestion de source simplifiée
- Alléger le code

Annotations(2)

Annotations EJB 3.0

1- Nature de l'EJB

- @Stateless, @Stateful, @MessageDriven ou @Entity

2- Accès distant ou local

- @Remote, @Local
- Plus de RemoteException

3- Cycle de vie

- @PostConstruct, @PostActivate, @Remove, @PreDestroy, @PrePassivate

Annotations(3)

- **@Entity** indique que le User class est un bean entité.
- **@EJB** configure la valeur EJB pour chaque méthode
- **@Id** annotation indique l'attribut de la clé primaire du bean entité
- **@Table** spécifie la table de base de données pour les bean entité

Annotations (4)

- **@SecondaryTable** spécifie la table secondaire
- **@Serialized** specifie que la propriété persistante doit être serializable
- **@JoinColumn** spécifie la colonne de jointure avec la colonne secondaire
- **@OneToMany** Specifie le champ comme une relation1-n

Injection

- Le composant déclare ce dont il a besoin au travers de meta-données

- Le conteneur fournit au composant ce qu'il réclame
  - Les ressources et services nécessaires à un composant sont « injectées » à la création de chaque instance de composant
    - Transactions
    - Notifications de cycle de vie
    - Mécanisme de persistance

Les annotations d'injection

- **@EJB**
  - Annote les interfaces métier des EJB
  - Annote les interfaces Home

- **@Inject ou @PersistenceContext**
  - Permet l 'injection de tout ce qui se trouve dans JNDI
  - Permet d 'injecter SessionContext, UserTransaction, EntityManager, ...

- **@Resource**
  - Annote toute le reste (ou presque)
  - Injection de variable d 'instance ou de *setter*

Annotations vs. Descripteur Déploiement

- Le descripteur de déploiement reste pertinent

  - Pour un développeur ou l 'entreprise qui a ses habitudes
  - Pour externaliser des meta-informations
  - Pour redéfinir des annotations

EJB3 : Bean Session(1)

- Spécification par @Stateful ou @Stateless
- stateful session beans sont transitoires
- @Remove : spécifier au conteneur que le stateful session bean instance peut être supprimé

EJB3 : Bean Session (exemple)

```java
package com.ejb.test;

import java.util.List;

import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import com.test.jpa.Employee;

/**
 * Session Bean implementation class emplManager
 */
@Stateless
public class EmplManager implements EmplManagerRemote {

    /**
     * Default constructor.
     */
    @PersistenceContext
    private EntityManager entityManager = null;
    public EmplManager() {
        // TODO Auto-generated constructor stub
    }

    @Override
    public String getEmploye() {
        List<Employee> list= entityManager.createNamedQuery("all-employee").getResultList();

        String s = list.get(0).getFirstName();
        return s;
    }

}
```

```java
package com.ejb.test;

import javax.ejb.Remote;

@Remote
public interface EmplManagerRemote {
    public String getEmploye();
}
```

EJB3 : Bean Entity (exemple)

```java
package com.test.jpa;

import java.io.Serializable;
import javax.persistence.*;
import java.util.Date;

@Entity
@NamedQueries({@NamedQuery(name="all-employee", query="select o FROM Employee o")})
public class Employee implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    private String userName;

    private String dept;

    private String emailAddr;

    private Date empDate;

    private String firstName;

    private String lastName;

    private Date modDate;

    private String password;

    public Employee() {
    }

    public String getUserName() {
        return this.userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }
}
```

EJB3 : Client EJB (exemple: JSP)

```jsp
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
<%@ page import="javax.naming.*, javax.rmi.PortableRemoteObject, java.util.List, java.util.Hashtable" %>
<%@ page import="com.ejb.test.EmplManagerRemote" %>
</head>
<body>
<h3>Un exemple de page JSP qui utilise un Bean de Session et un Bean Entity (A ENRICHIR)</h3>
<%
    // exactement le code de la servlet
    out.println("<h4> Nous allons acc&eacute;der au Bean Session (EJB)</h4><hr/>");
    // on utilise le contexte par defaut
    Context initialContext = new InitialContext();

    EmplManagerRemote v_EmployeeManager =
    (EmplManagerRemote) initialContext.lookup("com.ejb.test.EmplManager"
            + "_" + EmplManagerRemote.class.getName() + "@Remote");

    //Récupération de l'employé
    String emp = v_EmployeeManager.getEmploye();


    out.println("<h4> Voici les informations de : "+emp+"</h4><hr/>");

%>
</body>
</html>
```

EJB3 : Exemple Client (Servlet)

```java
import java.io.IOException;
import java.io.PrintWriter;
import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.Test.EJB.HelloWorldRemote;

public class TestServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public TestServlet() {
        super();

    }
    @EJB
    private HelloWorldRemote hellobean;

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
         response.setContentType("text/html");
            PrintWriter  out = response.getWriter();
            out.println("<html>");
            out.println("<head>");
            out.println("<title>" + "simple servlet"+ "</title>");
            out.println("</head>");
            out.println("<body>");

            out.println("<h3>"+hellobean.say("123")+"</h3>");
            out.println("<h3>hello</h3>");

            out.println("</body>");
            out.println("</html>");
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // TODO Auto-generated method stub
    }
}
```

EJB3 : Entity Bean(1)

- Un EJB entité devient une simple classe Java

  1- Classe concrète
  2- Héritage et polymorphisme
  3- Méthodes **getXX/setXX** pour les propriétés persistantes
  4- Pas d 'interface à implémenter
  5- Pas de méthode de callback requise

- Specifié par l'annotation @Entity

EJB3 : Entity Bean(2)

- Entity bean représente les données persistantes de la base de données
- Entity beans sont aussi partagés par plusieurs clients
- Utilisable comme objet « détaché » dans d'autres parties de l'application:
    1- Plus besoin de Data Transfer Object (DTO)
    2- Doit implémenter **Serializable**

EJB3 : Entity Bean(3)

```
@Entity
@Table(name="ORDER_TBL")      // nom table explicite
public class Order {

   ...

   @Id(generate=AUTO)         // clé primaire
   public Long getID() {
     return id;
   }

   @ManyToOne(cascade=ALL)          // create, merge, delete
   @JoinColumn(name="cust_id")      // clé étrangère
   public Customer getCustomer() {
     return cust;
   }

   public setCustomer ( Customer c ) {
     this.customer = c;
   }

   ...

}
```

EJB3 : Entity Bean(3)

```
@Entity
@Table(name="ORDER_TBL")      // nom table explicite
public class Order {

   ...

   @Id(generate=AUTO)         // cl
   public Long getID() {
      return id;
   }

   @ManyToOne(cascade=ALL)          // create, merge, delete
   @JoinColumn(name="cust_id")      // clé étrangère
   public Customer getCustomer() {
      return cust;
   }

   public setCustomer ( Customer c ) {
      this.customer = c;
   }

   ...

}
```

Spécifie la table de la base pour l'entité bean

Spécifie le champ comme une relation n-1

Définit la clé étrangère

EJB3 : Bean Message

Message-Driven Beans :

- Les beans messages (MDB) offrent une méthode simple pour implémenter une communication asynchrone.
- MDBs sont crées pour recevoir des messages JMS asynchrones.
- javax.jms.MessageListener interface
- Annotation: @MessageDriven.

EJB3 : persistance (1)

- **La persistance** : ce terme est utilisé pour indiquer le fait qu'un objet peut être sauvegardé de façon plus ou moins automatisée

- *Le groupe EJB 3 a décidé de proposer une nouvelle technologie de persistance, totalement en opposition avec les choix précédents.*

EJB3 : persistance (2)

- **La notion de persistance**

1. Simple et incomplète avant EJB 3.0
2. Nécessite de trouver le lien entre approche object et relationnel
3. EJB 3.0 propose au développeur d'établir simplement un mapping entre domaine métier (objet) et base de données relationnelle

EJB3 : persistance (3)

4. Le développeur a conscience de l'existence de ce mapping entre schéma SGBDR et modèle objet

   - Dans la manipulation des requêtes

   - Dans l 'utilisation des annotations dédiées au mapping O/R

   - Dans les règles de configuration par défaut

EJB3 : persistance (4)

Objets persistants :

- Objets à vie courte contenant l'état de persistance et la fonction métier.
- Ils sont en général les objets de type JavaBean (ou POJOs)
- La seule particularité est qu'ils sont associés avec une (et une seule) Session.

Unité de Persistance (Persistence Unit)

- **Une unité de persistance est caractérisée par :**
  - **Un ensemble d'Entity Beans**
  - **Un fournisseur de persistance (Provider)**
  - **Une source de données (Datasource)**
- **Le rôle d'une unité de persistance est de :**
  - **Savoir où stocker les informations**
  - **S'assurer de l'unicité des instances de chaque identité de persistance**
  - **Gérer les instances et leur cycle de vie (Entity Manager)**

PACKAGING

Intégration et Packaging

- L'utilisation des Entity Beans ouverte aux :
  - Applications Entreprise (EAR)
  - Module Java Bean Entreprise (EJB-JAR)
  - Application Web (WAR)
  - Client d'application d'entreprise (JAR)
- Le packaging est standardisé
  - La persistance est configurée dans « persistence.xml »
  - On choisit un « Entity Manager »
  - On configure le mapping spécifique
  - On caractérise les propriétés de l'Entity Manager
  - Hiérarchie d'assemblage avant déploiement

Fichier ear

- Les applications Web ne peuvent contenir qu'un seul fichier war
- Les applications plus complexes, par exemple qui utilisent des MDB, contiennent plusieurs fichiers jar qui sont réunis en un seul fichier ear (format jar avec une structure particulière qui permet de contenir plusieurs fichiers jar)

Exemple de fichier de persistance Hibernate

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
    <persistence-unit name="testJPA" transaction-type="JTA">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <jta-data-source>java:/MySQLDS</jta-data-source>  <!-- See <datasource jndi-name="..."> . -->
        <class>model.RegisteredUser</class>
        <properties>
            <property name="showSql" value="true" />
            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
            <property name="hibernate.hbm2ddl.auto" value="update" />
            <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />

            <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/test" />
            <property name="javax.persistence.jdbc.user" value="root" />
            <property name="javax.persistence.jdbc.password" value="root" />
        </properties>
    </persistence-unit>
</persistence>
```

Conclusion

- EJB 3.0 – Simplification drastique pour le développeur
  - 1- Suppression de classes et d 'interfaces
  - 2- Modèle d 'injection de services et de ressources
  - 3- Descripteurs de déploiement rendus facultatifs
  - 4- EJB entité POJO
  - 5- Capacités de requêtage améliorées
  - 6- Spécification du mapping O/R
- EJB 3.0 constitue les fondations de Java EE 5.0 pour le développeur

EJB 3 SESSION BEANS

Session Beans

- **Sans état** : après chaque appel, son état est réinitialisé
  - Exemples :
    - récupérer la liste des comptes bancaires d'un client dans une base de données
    - effectuer une transaction bancaire
- **Avec état** : son état est maintenu pendant toute la durée de vie du contexte auquel il est associé,
  - Exemple :
    - Un panier sur un site de vente en ligne
- **Singleton** : quand on veut être assuré qu'il n'y a qu'une seule instance du bean pour tous les utilisateurs de l'application
  - Exemple :
    - cache d'une liste de pays (pour améliorer les performances)
    - ...

Lifecycle callbacks

- Comme pour tout bean, on a les callbacks de cycle de vie classique
  - @PostConstruct
  - @PreDestroy

Stateless Session Beans

- Annotation **@Stateless**
- Le bean est **réinitialisé** après chaque appel de méthode
  - Le client passe toutes les données nécessaires au traitement lors de l'appel de méthode
  - Pas d'appel concurrent

- **Réinitialisé, pas forcément détruit**:
  - le container gère un **pool** de beans sans état:
    il choisit si il veut créer plus de beans, ou en destruire certains
  - Chaque serveur d'application peut permettre de configurer:
    - Le nombre de beans au départ
    - Le nombre de nouveau beans à créer en une fois si besoin

Exemple de bean sans état

```
@Stateless
public class GestionnaireDeCB {

  @Inject EntityManager em;

  public CompteBancaire find(int id) {
     return em.find(CompteBancaire.class, id);
  }
}
```

Interface locale et implémentation

```
@Local
public interface Convertisseur {
  public CompteBancaire find(int id);
}
```

```
@Stateless
public class ConvertisseurBean
        implements Convertisseur {
...
}
```

- Donner une interface locale explicite permet de choisir les méthodes exposées aux clients
- Si on ne spécifie pas d'interface, par défaut:
  - Toutes les méthodes publiques sont exposées
  - Le bean session ne peut être utilisé que par les composants qui sont dans le même serveur d'application

Interface distante et implémentation

```
@Remote
public interface Convertisseur {
  public CompteBancaire find(int id);
}
```

```
@Stateless
public class ConvertisseurBean
        implements Convertisseur {
...
}
```

- Le bean session peut maintenant être utilisé par un composant situé sur un autre serveur d'application
- Un bean peut implémenter 2 interfaces locale et distante.

Lifecycle callbacks

- Comme pour tout bean, on a les callbacks de cycle de vie
  - @PostConstruct (ouvrir les connexions etc...)
  - @PreDestroy (fermer les connexions etc...)

Stateful Session Beans

- Annotation **@Stateful**
- Certaines conversations se déroulent sous forment de requêtes successives.
- L'état du Stateful Session Bean est maintenu pendant toute la durée de vie du contexte auquel il est associé,
  - au cours d'appels de méthodes successifs.
  - au cours de transactions successives.
  - Si un appel de méthode change l'état du Bean, lors d'un autre appel de méthode l'état sera disponible.
- Comme pour les requêtes HTTP, sauf qu'ici on a accès aux services middleware du conteneur d'EJB

Exemples d'utilisation

```
@Stateful
@SessionScoped
public class CaddyEJB {...}
```

```
@Model
public class PanierBean {
  @EJB CaddyEJB caddy;
   ...
}
```

Annotation @Remove

Lorsqu'on a fini avec le session bean avec état, on peut donner au conteneur le feu vert pour supprimer le bean.

```
@Stateful
public class CaddyEJB {
  ...
  @Remove
  public void checkout() {
    caddy.clear();
  }
}
```

Annotation @StatefulTimeout

On peut indiquer une limite de temps d'existance du session bean avec état

```
@Stateful
@StatefulTimeout(300000) // 5 minutes maximum
public class CaddyEJB {
  ...
}
```

Problème de ressource

- Le client entretient une conversation avec le bean, dont l'état doit être disponible lorsque ce même client appelle une autre méthode.
- Problème si trop de clients utilisent ce type de Bean en même temps.
  - Ressources limitées (connexions, mémoire, sockets…)
  - Mauvaise scalabilité du système,
  - L'état peut occuper pas mal de mémoire…

Passivation / Activation

- **Passivation** : pour économiser la mémoire, le serveur d'application peut retirer temporairement de la mémoire centrale les beans sessions avec état pour les placer sur le disque
- **Activation** : le bean sera remis en mémoire dès que possible quand les clients en auront besoin
- Pendant la passivation il est bon de libérer les ressources utilisées par le bean (connexions avec la BD par exemple)
- Au moment de l'activation, il faut alors récupérer ces ressources

Activation/Passivation callbacks

- Lorsqu'un bean va être mis en passivation, le container appelle la méthode annotée @PrePassivate
  - Il peut libérer des ressources (connexions…)
- Idem lorsque le bean vient d'être activé (@PostActivate)

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

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

Bean Singleton

- On est sûr qu'il n'y a qu'une seule instance dans l'application, qu'elle va être créée « au début », et détruite lorsque l'application sera arrêtée.
- Annotation @Startup pour chargement aggressif du singleton
- A priori partagé par les clients
  → gestion des accès concurrents !

Accès concurrents d'un bean singleton

- Le code des beans singleton n'a pas besoin d'être thread-safe puisque le container ne permettra jamais l'accès par plusieurs requêtes
- On peut laisser le conteneur gérer les accès concurrents (par défaut), ou déclarer que c'est le singleton qui le fera
  - @ConcurrencyManagement(value = BEAN ou CONTAINER)
- Si le container gère la concurrence (par défaut)
  - @Lock(value = READ ou WRITE)

INJECTION D'UN EJB

Injection de dépendance pour un EJB

- Si on utilise CDI

```
@Inject MyEJBLocal myEJB; // permet de tirer tous les bénéfices de CDI !
```

- Si on est dans un objet géré par le conteneur d'EJB

```
@EJB MyEJBLocal myEJB; // injection sans la robustesse de CDI (qualifieurs, ...)
```

- Sinon: lookup JNDI

```
MyEJBLocal myEJB = lookupMyEJBLocal();
private MyEJBLocal lookupMyEJBLocal() {
        try {
            Context c = new InitialContext();
            return (MyEJBLocal) c.lookup("java:global/App/App1-war/MyEJB!fr.unice.ejb.MyEJBLocal");
        } catch (NamingException ne) {
            Logger.getLogger(getClass().getName()).log(Level.SEVERE, "exception caught", ne);
            throw new RuntimeException(ne);
        }
    }
```

Java Message Service (JMS)

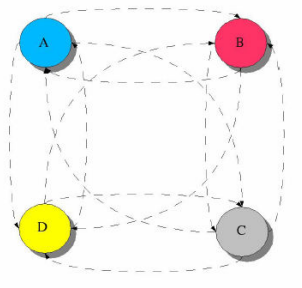
Message Oriented Middleware (MOM)

- Les Middleware Orientés Messages sont des systèmes fournissant à leurs clients un service de Peer to peer.
  - La définition standard du terme de Peer to Peer désigne un échange de données ou de fichiers d'une machine à une autre
- un MOM est un logiciel serveur dont le rôle est de fédérer l'envoi et la réception de ces messages entre les différents types d'applications.
- La communication de deux applications via un Message Oriented Middleware est complètement asynchrone, c'est à dire que l'émetteur et le destinataire n'ont pas besoin d'être connectés simultanément lorsqu'ils communiquent.
  - La communication n'est synchrone qu'entre l'emetteur et le MOM d'une part, et le MOM et le destinataire d'autre part

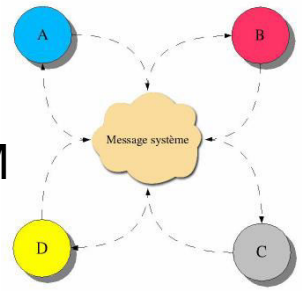
Message Oriented Middleware (MOM)

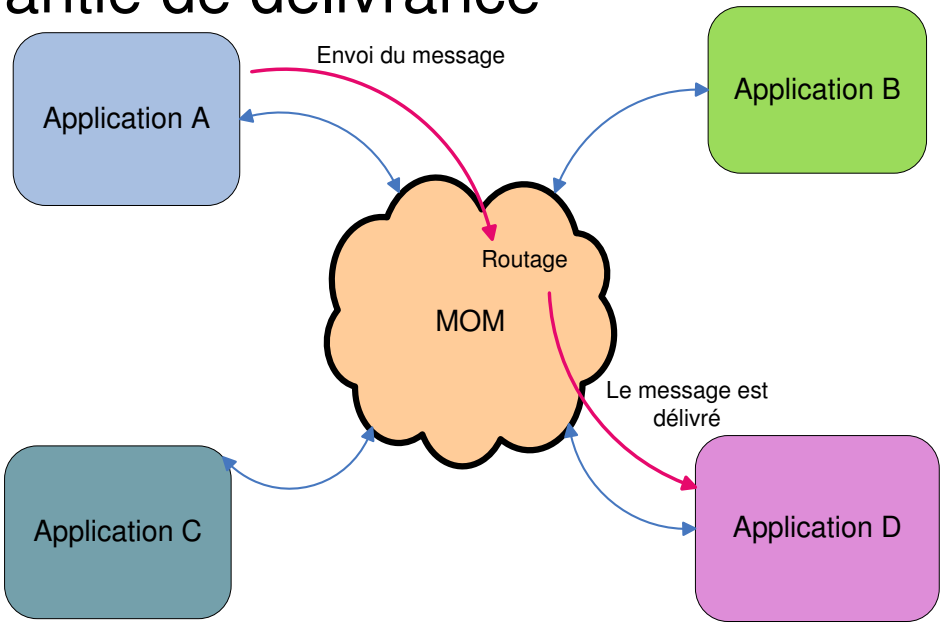
- Modèle de communication entre logiciels
  - Intégration de modules hétérogènes distribués
  - Indépendance (asynchronisme)
  - Fiabilité et garantie de délivrance

RPC



MOM





L'API Java Message Service

- JMS est une API, spécification de Sun depuis 1998, pour la création, l'envoi et la réception des messages de façon :
- Asynchrone :permet d'envoyer et de recevoir des messages de manière asynchrone entre applications ou composants Java. JMS permet d'implémenter une architecture de type MOM (Message Oriented Middleware). Un client peut également recevoir des messages de façon synchrone dans le mode de communication point à point.
- L'API JMS permet aux applications Java de s'interfacer avec des intergiciels (middleware) à messages ou MOM. Les MOMs permettent des interactions entre composants applicatifs dans un cadre faiblement couplé, asynchrone et fiable.
- un client reçoit les messages dès qu'ils arrivent ou lorsque le client est disponible et sans avoir à demander si un message est disponible.
- Fiable : le message est délivré une fois et une seule.

L'API Java Message Service

- Versions:
  - JMS 1.0.2b, juin 2001
  - JMS 1.1, mars 2002
  - JMS 2.0 (JSR 343), 20 mars 2013
- Fournisseurs de service JMS

  Pour utiliser l'API JMS il est nécessaire d'avoir un fournisseur de service qui gère les connexions, les sessions, les destinations et les messages. Il y a de multiples fournisseurs de service JMS :
  - Implémentation Open Source
    - Apache ActiveMQ, OpenJMS, JBoss Messaging et HornetQ de Jboss, JORAM, de ObjectWeb maintenant OW2, Open Message Queue, de Sun Microsystems
  - Implémentation commerciales
    - BEA Weblogic, Oracle AQ, SAP NetWeaver, SonicMQ, Tibco Software, webMethods Broker Server, WebSphere MQ, FioranoMQ de Fiorano

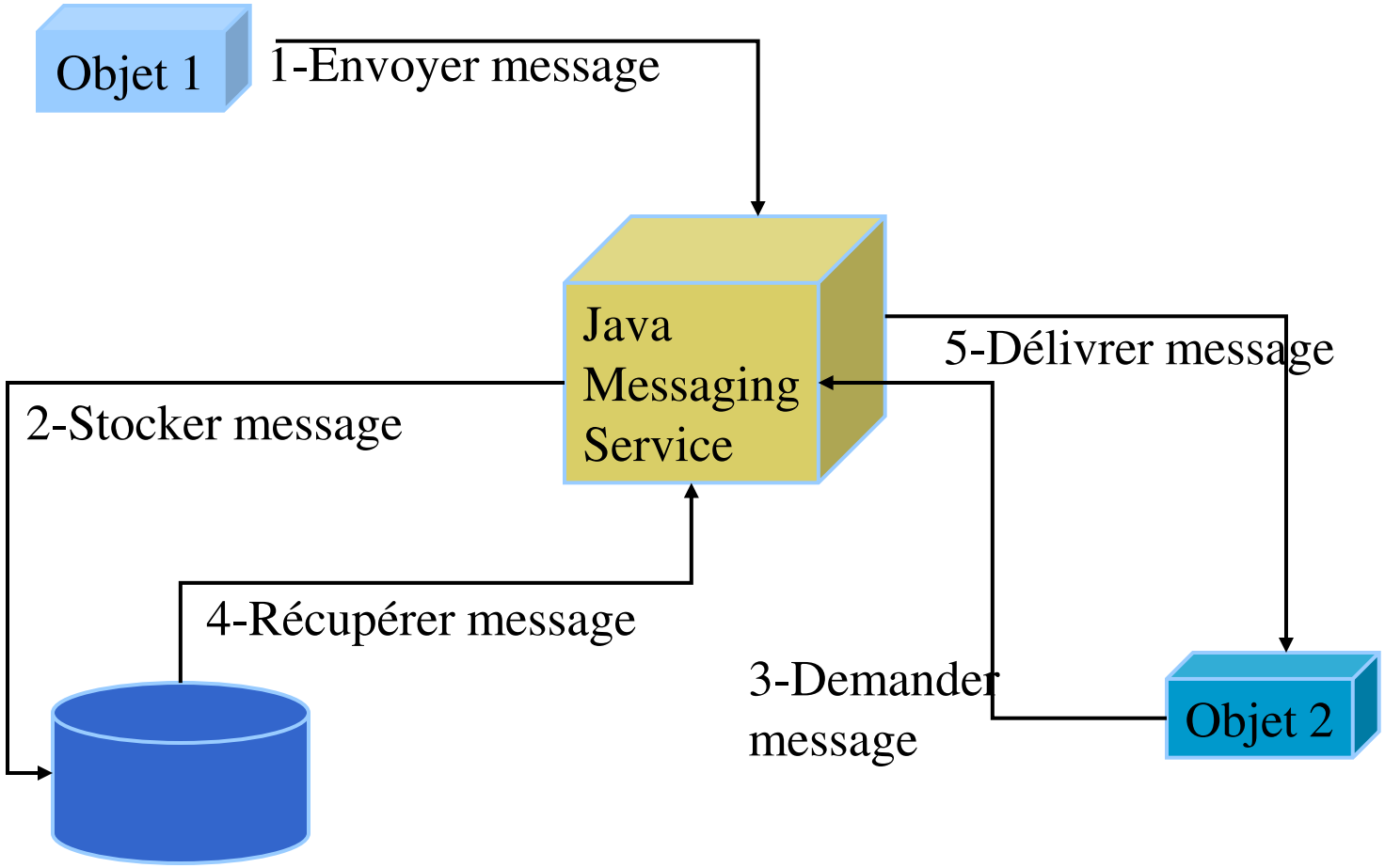
Quand utiliser JMS?

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

Comment fonctionne JMS avec Java EE?

- JMS peut accéder aux Messaging-oriented middleware (MOM), tels que JORAM de ObjectWeb ou MQSeries d'IBM ou JBoss Messaging et HornetQ de Jboss,
- Elle fait partie intégrante de Java EE à partir de la version 1.3.
- Réception et envoi de messages de type :
  - Synchrone : applications clientes , EJBs, composants Web (sauf applets, hors spécification)
  - Asynchrone : applications clients, les Beans orientés messages.
- Les composants messages participent aux transactions distribuées des autres composants.

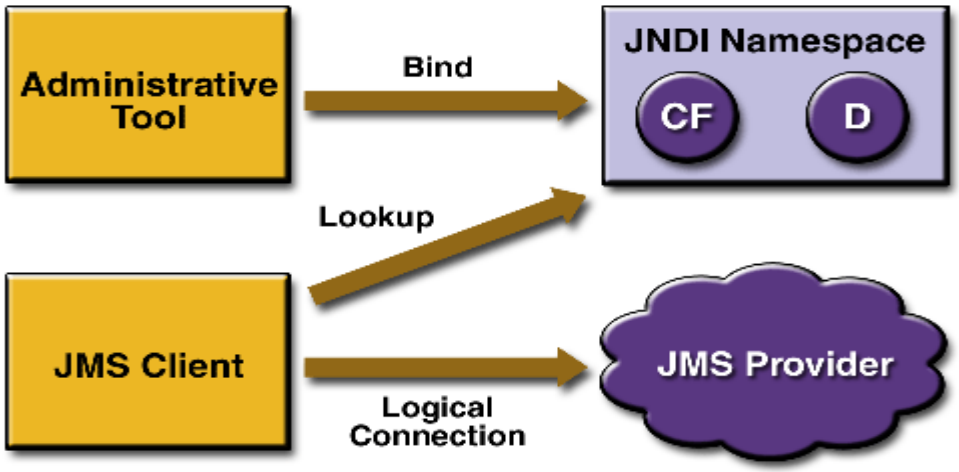
Middleware orienté message Java EE : JMS



L'architecture JMS

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

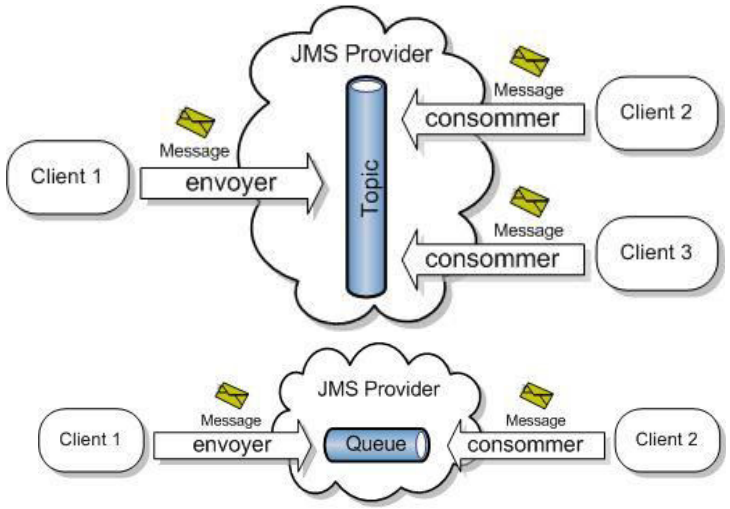
L'architecture JMS

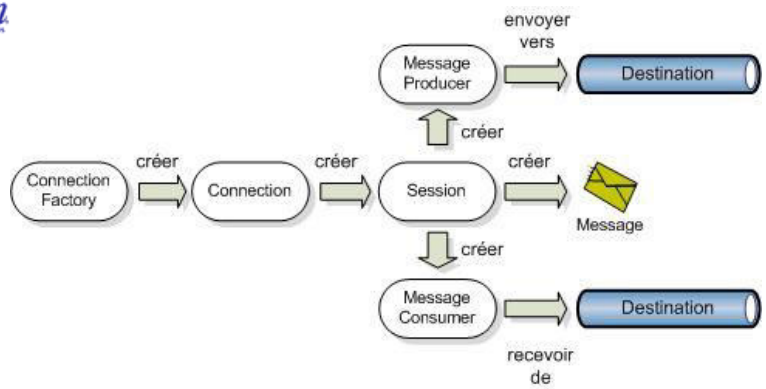


- Les *Administrative tools* permettent de lier les destinations et les connections factories avec JNDI.
- *Un client* JMS peut alors rechercher les objets administrés dans JNDI et établir une connexion logique à travers le JMS Provider.

Composants d'une application JMS

- Fournisseur JMS (MOM)
  - Modèle P2P, Pub/sub
  - Fonctions administration
- Clients JMS
  - Producteur, consommateur
- Objets administrés
  - Connexions et destinations
- Messages

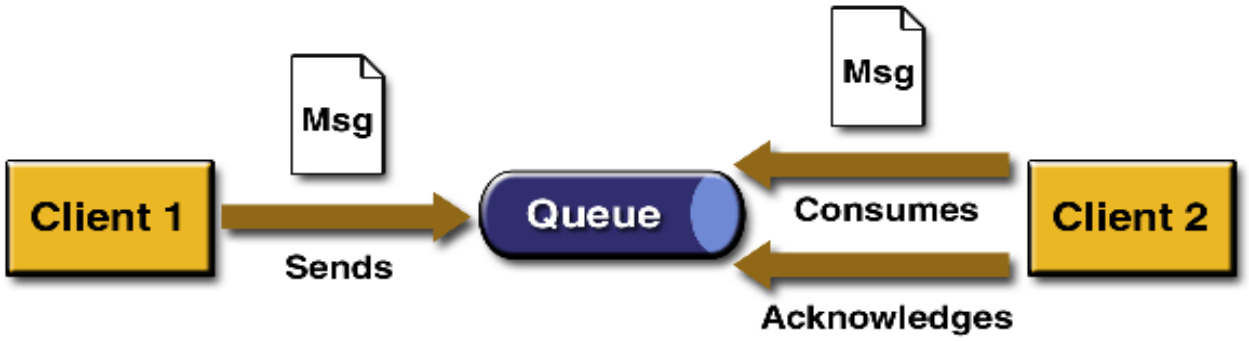




Les modes de communication

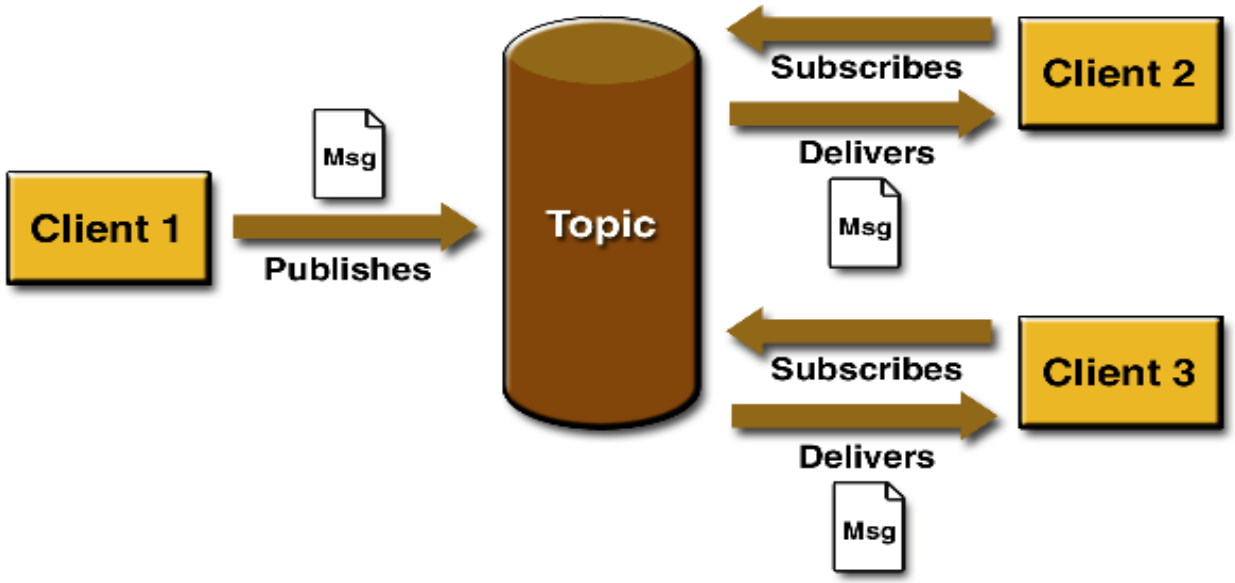
- JMS définit deux modes distincts pour envoyer des messages:
  - le mode Point à Point (ou Point to Point)
    - le producteur publie les messages dans une file (queue) et le consommateur lit les messages de la file. Dans ce cas le producteur connaît la destination des messages et poste les messages directement dans la file du consommateur. Pour utiliser ce modèle, le consommateur doit invoquer la méthode receive() qui est bloquante
  - Publish/Subscribe (Publication/Abonnement)
    - Dans le modèle publication/abonnement, des entités s'inscrivent sur un topic pour recevoir des messages. En effet, il ne s'agit plus d'envoyer des messages sur une file (queue) mais sur un topic. Celui qui publie les messages et ceux qui les reçoivent ne se connaissent pas.
  - Pour préciser rapidement, on peut dire que le mode Point à Point s'apparente à l'envoi d'un mail et le mode Publish/Subscribe à une souscription auprès d'un serveur de news.

Le mode point-à-point



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

Le mode publier/souscrire



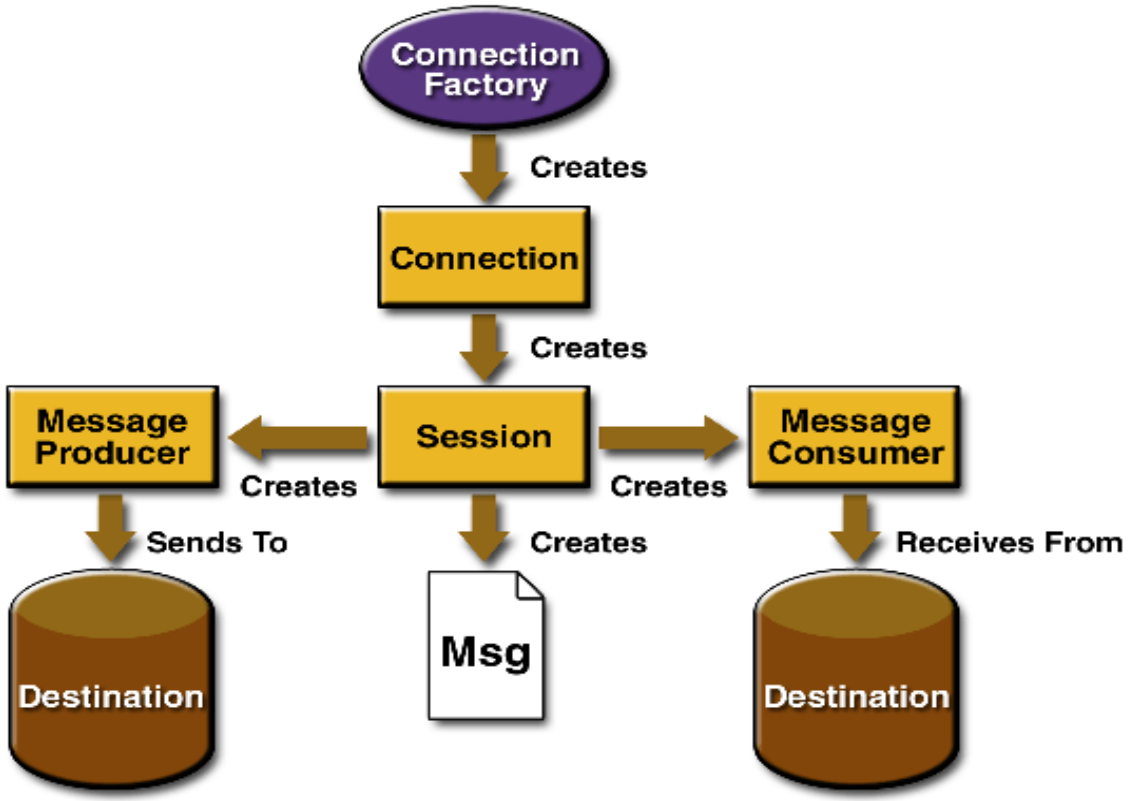
- Les clients sont anonymes et adressent les messages selon certaines rubriques.
- Le système distribue le message à l'ensemble des souscripteurs puis l'efface.
- Ce mode est efficace lorsqu'un message doit être envoyé à zéro, un ou plusieurs clients.

La consommation de message

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

Le modèle de programmation JMS

- **Une *connection factory*:** est l'objet qu'un client utilise pour créer une connexion avec un JMS provider.
- **Une destination** est un objet qu'un client utilise pour spécifier la cible des messages produits et la source des messages consommés.
- **Un objet connection** encapsule une connexion virtuelle avec le JMS Provider
- **L'objet session** est un contexte transactionnel pour produire et consommer des messages.
- **Un producteur de message** est un objet créé par une session et utilisé pour envoyer des messages à une destination



Les évolutions de JMS 2.0

Objectifs de JMS 2.0

- Parmi les objectifs de JMS 2.0
  - Simplification de l'API
  - Intégration avec CDI
  - Clarifier les ambiguités de l'API et les relations avec les autres spécifications Java EE
  - Standardisation de l'interface entre les fournisseurs JMS et les serveurs d'application
  - Standardisation de fonctionnalités proposées par les différentes implémentations JMS disponibles

Simplifications de l'API

- Création des sessions JMS
  - signature actuelle :
    - connection.createSession(transacted, deliveryMode)
  - nouvelle signature pour Java SE :
    - connection.createSession(sessionMode)
  - nouvelle signature pour Java EE :
    - connection.createSession()
- Méthode raccourcie pour envoyer un message
  - producer.send(String body)
  - au lieu de devoir créer un message JMS avant de l'envoyer via :
  - producer.send(javax.jms.Message message)

Simplifications de l'API

- **Création d'une nouvelle API JMS**
  - Un des problèmes de l'API JMS est qu'elle demande beaucoup de code qui n'est pas toujours utile.
    Ainsi rien que pour envoyer un message, elle nécessite l'obtention de nombreux objets : en particulier une ConnectionFactory,
    une Connection, une Session, un MessageProducer…
    - la création d'une nouvelle classe : JMSContext regroupant les responsabilités de : Connection + Session + MessageProducer

```
@Inject
JMSContext context;

@Resource(mappedName="java:global/jms/myQueue")
Queue queue;

public void sendMessage(String message) {
context.createProducer().send(queue, message);
}
```

Intégration avec CDI

```
@Inject
JMSContext context;

@Resource(mappedName="java:global/jms/myQueue")
Queue queue;

public void sendMessage(String message) {
context.createProducer().send(queue, message);
}
```

```
@Inject
 private JMSContext context;

 @Resource(mappedName="java:/jms/queue/test")
 Queue myQueue;

 public String receiveMessage() {
 String message = context.createConsumer(myQueue).receiveBody(String.class, 1000);
 return "Received " + message;
 }
```

Références

- http://www.oracle.com/technetwork/articles/java/jms20-1947669.html
- https://docs.oracle.com/javaee/7/tutorial/
- https://java.net/projects/jms-spec/pages/Home
- Java Message Service: http://www.oracle.com/technetwork/java/jms/index.html
- Apache ActiveMQ: http://activemq.apache.org/
- HornetQ Jboss: http://hornetq.jboss.org/