Client-serveur à client passif

- Terminal ou émulation de terminal
- Application hébergée sur un serveur
- Logique applicative et données concentrées en un seul point : un site central
- Le client ne fait que de la visualisation et de l'insertion de données
- Flux réseau : uniquement des informations de présentation

Client-serveur de données

- Vague PC dans les années 80
- Le mainframe contient la logique applicative et les données d'entreprise => pas possible de migrer totalement
- Le PC effectue les traitements et le serveur la gestion des données
  - avènement des SGBD et outils L4G
  - communication via SQL
- Avec le recul : coût de maintenance prohibitif
- Déploiement à grande échelle complexe => le mainframe reste indispensable

Client-serveur distribué

- Succès mitigé du cas précédent => on ne transfère vers le PC que quelques portions de programmes sous forme de procédures stockées.
- Toujours le problème du déploiement et de la maintenance des PC => amélioration coté serveur nécessaire
- Introduction d'un serveur tiers (dit d'application car il exécute des portions d'application) qui décharge le client d'une part des traitements. Moins de données qui transitent sur le réseau.
- Problème : technologies, solutions généralement propriétaires

Client-serveur à objets distribués

- Solution + complète : CORBA
- Avantage : standardisé
- RPC est remplacé par des requêtes IIOP
- Problème : complexe à mettre en œuvre

Problèmes spécifiques

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

Serveur d'application et EJB

- Tendance actuelle :
  - les applicatifs sont développés avec des langages objets et conditionnés sous forme de composants réutilisables
- Modèle de SUN : J2EE
  - but : fournir un espace d'exécution et un framework clé en main, sur un plate-forme JAVA
  - les développeurs n'ont plus qu'à développer leurs services métiers spécifiques en suivant des spécifications précises
    - programmation par assemblage et focalisation de l'expertise sur les problèmes du domaine à informatiser plutôt que sur des compétences techniques informatiques (gestion des transactions, sécurité, parallélisme, …)
  - de plus, le modèle propose une répartition précise des rôles lors du développement d'applications distribuées

Acronyme Entreprise Java Bean

- L'acronyme EJB désigne à la fois un composant et une architecture
- Un EJB, c'est :
  - une entité de traitement dans une application distribuée
  - qui s'exécute dans un environnement adapté : conteneur
  - qui nécessite des technologies additionnelles : transaction, appels à distance, …
- L'architecture :
  - il s'agit d'un modèle de composants coté serveur qui spécifie comment créer des applications serveur portables, transactionnelles, multi utilisateurs, et sécurisées
  - ces applications peuvent être déployées sur des systèmes de gestion de transaction existants, comme les moniteurs de transactions traditionnels, les serveurs web, les serveurs de BD, …

Composant EJB

- 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

Intérêts

- Les EJB fournissent aux développeurs une indépendance vis-à-vis de la plate-forme :
  - dans une architecture n-tiers, peu importe où se trouve la logique applicative. D'une part les développeurs sont isolés du middleware sous-jacent, et d'autre part cela permet aux vendeurs d'apporter des modifications sans affecter les applications existantes
- WORA : tant qu'on reste conforme à la spécification, on peut faire tourner un EJB sur n'importe quel serveur
- Des rôles sont définis pour les différents participants au projet => meilleure organisation des tâches
- Les EJB prennent en charge la gestion des transactions : le vendeur du conteneur fournit le service pour les contrôler, et le développeur ne s'occupe pas de les débuter ou terminer
- Les EJB fournissent un service de transaction distribuées transparent : ils peuvent être présents sur différents serveurs, tourner sur des machines plate-formes ou machines virtuelles JAVA différentes. Le développeur est assuré qu'elles seront exécutées dans le même contexte transactionnel

Technologies préalables

- JAVA : WORA
- BEANS : on respecte certaines conventions sur le nommage, la construction et le comportement des méthodes
- SERVLET : étendre dynamiquement le comportement d'un serveur
- JDBC : connecter facilement les applications à une BD
- JPA : API qui permet un mapping objet/relationnel
- Annotations : ajout de métadonnées à du code Java

Composant logiciel

- Module logiciel : objet + configurateur + installateur
  - qui exporte différents attributs, propriétés, méthodes,
  - qui est prévu pour être configuré
  - qui est prévu pour être installé
  - qui fournit un mécanisme lui permettant de s'auto-décrire
- Caractéristiques
  - fournit des services
  - utilise d'autres composants
  - possède des propriétés configurables
  - spécifie quelles doivent être les caractéristiques de l'environnement
    - en terme de système (OS, librairies)
    - en terme de service (transactions, sécurité, persistance, …)

Transactions

- Concept essentiel
  - Pour l'utilisateur : un simple changement qui se produit ou pas
  - Pour le développeur un style de programmation qui peut impliquer plusieurs modules qui participent à une transaction distribuée.
- Ex : transfert d'argent d'un compte à un autre :
  - difficulté à faire fonctionner ce traitement dans un système distribué, car problème de contrôle de la transaction
- Objectif : réunir dans un seule unité d'exécution tout un ensemble d'opérations
- Dans une transaction plusieurs entités totalement indépendantes doivent s'accorder avant que le changement soit conclu. Si un des participants n'accepte pas, il faut que chacune des entités revienne à son état initial.

Propriétés ACID

- Atomicité : la suite d'opérations est indivisible ; une transaction se termine soit par commit : tous les modifications sont conservées ou roll back : aucune modification n'est conservée
- Consistance : transformation correcte de l'état de la base en en préservant la cohérence. La cohérence est propre au domaine et c'est l'application qui délimite des transactions cohérentes
- Isolation : des transactions concurrentes sont isolées des modifications des autres transactions (aussi nommée serializability)
- Durabilité : après transaction les changements sont permanents

JTS : Java Transaction Service

- Joue le rôle d'un coordinateur de transaction entre les différents composants de l'architecture EJB. Un objet est chargé de gérer la transaction, le transaction manager directeur. Les autres participants : resource managers.
- Quand une application débute une transaction, elle crée un objet transaction, puis invoque les différents participants. C'est le transaction manager qui conserve la trace de tous les objets participants.
- En cas d'échec dans l'application, le JTS interrompt la transaction. En cas de succès le JTS met en œuvre une protocole two-phase commit pour valider tous les participants

Commit à deux phases

- Protocole qui assure que tous les resource managers valident la transaction ou l'abandonnent
- Le JTS interroge chaque RM pour savoir si il est prêt à valider. Quand tous les RM acceptent, le JTS renvoie un message de validation
- Si un des RM n'accepte pas, le JTS envoie à tous les RM l'ordre d'abandon

Le service de nommage

- Rôle : identifier et associer des noms à des données
  - Ex : système de fichiers, ou BDR
- Logiciel dédié chargé de gérer un espace de noms
  - indépendant de la plate-forme, s'adapte à tout système qui connaît le protocole et peut s'y connecter
  - généralement 2 niveaux : client et serveur
    - le serveur est responsable de maintenir et retrouver les noms effectifs (établir les liens, contrôler les accès, gérer toutes les opérations sur la structure générale, …)
    - le client est une interface que les applications utilisent pour communiquer avec le service de répertoire

JNDI

- Java Naming and Directory Interface
- Fournit des fonctionnalités pour nommer et répertorier des informations
- Ce n'est pas un service de nommage, mais une interface commune pour d'autres services existants : DNS, CORBA, RMI, …
- Cache tous les détails d'implémentation des différents services, permet à plusieurs services de cohabiter, toutes les informations semblent être fédérées sous un seul service de nommage

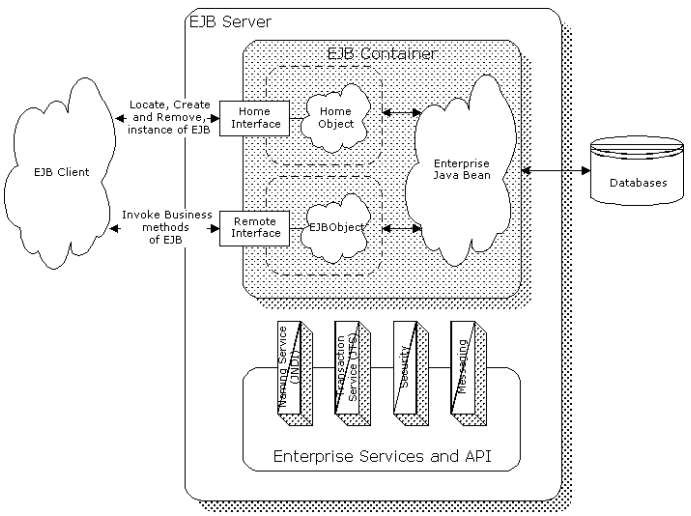
JavaBeans

- C'est un modèle générique qui définit un environnement qui supporte des composants réutilisables
- Chaque objet doit être conforme à un certain nombre de règles de base, comme avoir un constructeur par défaut, être sérialisable, exporter des propriétés, des événements et des méthodes
- Doit aussi capable de gérer des événements et de fournir des méthodes pour ajouter ou enlever des «écouteurs d'événements »

L'architecture EJB

- L'architecture consiste en
  - un serveur
  - des conteneurs
  - des objets EJB et des EJB
  - des clients d'EJB
  - des services auxiliaires (JNDI, JTS, sécurité, …)

Le modèle EJB 2



Le serveur d'EJB

- Environnement d'exécution dans lequel s'exécutent les conteneurs : conteneur de conteneur
- Gère le multiprocessing, la répartition de la charge et l'accès aux périphériques
- Peut aussi proposer d'autres fonctionnalités propres au vendeur

Les conteneurs

- Interface entre un EJB et les fonctionnalités de bas niveau, spécifiques à la plate-forme
- C'est une entité qui gère une ou plusieurs classes d'EJB, en mettant à disposition les différents services : gestion des transactions, gestion des instances, gestion de la persistance, gestion de la sécurité
- Tout accès à un bean se fait via des méthodes de classes générées par le conteneur, qui elles-mêmes appellent les méthodes du bean
- Deux types de conteneurs :
  - session containers : contiennent des EJB volatiles (transcient), non persistants
  - entity containers : qui contiennent les EJB persistants, dont les états doivent être sauvegardés entre les différentes invocations

L'interface Remote et l'objet EJB

- L'ensemble de toutes les méthodes propres au métier est défini dans l'interface Remote
  - les clients n'invoquent jamais directement les méthodes de la classe du Bean
- Cette interface représente la connaissance qu'a le client de l'EJB
- Elle est implémentée par une classe automatiquement générée et dont l'EJBObject est une instance
  - celui-ci est géré par le conteneur
- Les appels de méthodes sont interceptés par le container, afin d'assurer le traitement middleware implicite
  - une fois ce traitement effectué, le container appelle les méthodes de la classe du bean
- Le développeur de composant se concentre sur la logique, ignore la partie middleware

L'EJB

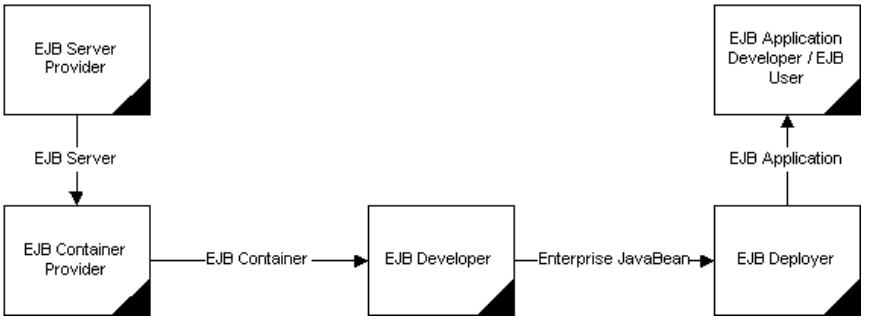
- C'est un objet contenu dans le conteneur
- Il est géré par le conteneur et ne doit jamais être adressé directement
- La classe du Bean : implémente une interface précise et qui respecte certaines règles
  - il s'agit de l'implémentation du bean lui-même,
    - logique métier, calculs, transfert de compte bancaire, saisie de commandes, etc…
    - logique orientée donnée, par exemple comment changer le nom d'un client, diminuer un compte bancaire…
    - logique orientée message, traitement après réception d'un ordre d'achat d'actions boursières…

Les clients

- Le client d'un Bean peut être une servlet, une applet, une application classique, un autre bean
- Un client identifie un conteneur d'EJB spécifique via l'interface JNDI, puis utilise l'EJB grâce aux méthodes spécifiées dans l'interface Remote
- Lorsque le client demande l'exécution d'une méthode métier, l'objet EJB reçoit une requête qu'il délègue au bean correspondant

Les acteurs

- Implication de plusieurs experts
- La spécification répartit les responsabilités et les rôles
- Le développement n'implique pas toujours la même personne



Les fournisseurs

- Le fournisseur du serveur
  - implémente et fournit les accès à un service compatible JNDI, et un service de transactions
  - souvent aussi le fournisseur de conteneurs
- Le fournisseur de conteneurs
  - fournit les logiciels pour installer un EJB sur un serveur
  - est responsable de fournir les classes qui prennent en charge les différents services, entre autre
    - les classes permettant l'accès aux instances des EJB,
    - et les classes permettant leur référencement dans le service JNDI

Le développeur d'EJB

- Il doit avoir une certaine connaissance de la spécification
- Il est responsable de coder toute la logique métier du côté serveur : implémente des classes qui se focalisent sur le métier en utilisant les spécifications EJB
- Il doit comprendre comment fonctionnent les transactions
- Il est responsable de stipuler au déployeur tous les besoins des différentes méthodes des EJB en matière de transaction

Le déployeur d'EJB

- Il doit installer correctement les EJB sur le serveur
- Il ne lui est pas nécessaire de connaître Java, ni les règles du domaine que les EJB implémentent
- Il doit connaître le cadre général de l'application
- Il doit connaître l'environnement d'exécution du serveur (BD, localisation, …)
- Le développeur lui fournit ses besoins => doit s'assurer que l'EJB est correctement référencé et accessible via JNDI, et installé avec les attributs corrects

Le développeur d'application

- Il écrit la partie client, en utilisant des EJB pré-construits : application Java, applet, servlet, application CORBA
- A juste à intégrer les EJB sous forme de plug-in, sans avoir besoin de les tester
- Se concentre sous des fonctionnalités de haut-niveau, comme la représentation de l'information, sans se préoccuper de comment l'information est réellement obtenue

Différents types d'EJB

- Entité , Session , Message
- Entity : utilisé pour modéliser une entité du métier
- Session : intérêt plus général : représente un traitement côté serveur
- Message : consommateur asynchrone de messages

Entity Bean

- Un EB représente des données persistantes du domaine, ainsi que les méthodes qui les manipulent
  - il correspond à un enregistrement. Dans une BDR : une ligne d'une table
  - possède un état qui va persister entre les différentes invocations de l'EJB
  - sa durée de vie n'est pas limitée à celle de la machine virtuelle => survit aux arrêts et crash système
- En EJB2, chaque EB est identifié par un objet qui lui est associé : sa clé primaire
- Plusieurs clients peuvent partager le même EJB
- En EJB2, la persistance peut être gérée par l'objet EJB ou par son conteneur
  - container–managed persistence : à la charge du conteneur : lors du déploiement on précise tous les champs qui seront persistants
  - bean-managed persistence : l'EB est responsable de la sauvegarde de son état, et le conteneur ne génère aucun appel à la BD => implémentation moins portable car la persistance est codée « en dur » dans le bean

Session Bean

- Sert à décrire les interactions entre autres beans ou à implémenter des tâches particulières
- Il est créé par un client et ne dure que pendant une session
  - il n'est pas récupéré après système crash ou shutdown
  - il est associé à un client, qui doit le créer et le détruire
  - il peut contenir une information privée propre au client qui l'a créé, mais il est anonyme
- Ne représente pas des données partagées dans une base mais peut accéder à ces données (pour lire, mettre à jour, en ajouter)
  - il s'occupe des interactions entre les beans entité en décrivant comment ils collaborent pour accomplir une tâche spécifique
- Peut être sans état ou maintenir un état « conversationnel » :
  - sans état (stateless) : collection de services liés, qui correspond à un but général et qui sont généralement utilisables
    - peut être utilisé dans un pool de SB, par plusieurs clients simultanément
  - avec état (statefull) : extension de l'application client. Il effectue des tâches à la place du client et maintient un état en rapport avec ce client. Cet état est appelé un état de conversation car il représente une conversation continue entre le bean et le client
    - est sujet à l'activation/passivation
    - associé à un seul client
    - possède un état qui traduit l'avancement du traitement, de la session

Les freins au succès des EJB 2

- Un modèle de programmation trop lourd
  - un composant de type EJB implique le développement de pas moins de trois classes (quatre pour certains EJB entité)
  - pas d'héritage, pas de polymorphisme
- Un modèle de persistance pas très efficace
  - mapping sommaire : une entité = une table
- Un modèle de déploiement complexe et pas encore totalement indépendant des serveurs d'application
  - lourdeur de la syntaxe , manque d'outils productifs, descripteurs souvent liés au serveur d'application

Les avantages des EJB 3

- Simplification du déploiement : la configuration peut se faire via les annotations
  - seul le paramétrage des contextes de persistance doit se faire via un fichier XML (persistence.xml)
- Configuration par exception
  - tout est paramétré par défaut pour faciliter et accélérer le développement
- Facilité de développement : l'application ne travaille qu'avec des objets « simples » (POJO)
- Injection de dépendance
  - l'utilisation d'annotations permet de faciliter les couplages entre les EJB, le conteneur web et le conteneur d'applications clientes

Les annotations

- Nouveauté de Java 5
- Permettent d'annoter le code avec une liste de mots clés, extensible selon les besoins
- On associe à ces mots clés un traitement spécifique pour automatiser certaines tâches
- Elles peuvent être utilisées à la compilation ou à l'exécution
- En général l'annotation est placée devant la déclaration de l'élément qu'elle marque

L'injection de dépendance

- Design pattern « Inversion Of Control » (IOC)
  - principe : découpler les éléments fonctionnels de leur environnement d'exécution et des paramètres fluctuant d'une exécution à l'autre
- Conséquence : on peut écrire un programme sans se soucier d'un certain nombre de paramètres extérieurs à son bon déroulement
- Permet de réduire le couplage d'une architecture grâce à un conteneur spécialisé
  - l'injection est paramétrée dans un descripteur de dépendance
  - le conteneur joue le rôle principal dans l'application du design pattern IoC
    - il est responsable de la création (instanciation) des objets
    - il résout les dépendances entre les objets qu'il gère

Les beans session sans état

- Facile à développer
- Toute activité qui peut être accomplie par un appel de méthode est susceptible d'être transformée en un BS sans état
- Peuvent avoir des variables d'instance, mais qui leur sont propres et qui ne seront jamais d'aucune utilité pour le client, ni même accédées
- L'instance du bean peut servir n'importe quel client qui fait l'invocation d'une de ses méthodes
- Utilisés généralement pour la génération de rapports

Les beans session avec état

- Etat conversationnel : variables d'instances qui contiennent des données relatives au client à conserver entre 2 invocations de méthodes.
  - les méthodes du bean manipulent les données de cet état qui sont partagées en toutes les méthodes
- Alternative entre les BS sans état et les BE
  - dédiés à un client pendant toute leur durée de vie : agissent en tant qu'agent du client
  - servent à diminuer la taille d'un client => système plus facile à gérer
  - ne sont pas persistants
  - ne sont pas partagés entre les clients
  - peuvent posséder une période d'expiration, ou bien être détruits explicitement par le client

Exemple de bean Stateless - Interface

```
import javax.ejb.Remote;

@Remote
public interface CoursManager
{
        public void create (Cours c) ;
        public Cours findCours(String coursId) ;
        public Collection findAll();
        ...
}
```

Exemple de bean Stateless – Classe

```
import javax.ejb.Stateless;

@Stateless
public class CoursManagerBean implements CoursManager
{
        public void create (Cours c) {…}
        public Cours findCours(String coursId){…}
        public Collection findAll() {…}
        …
}
```


Interface Remote ou Local

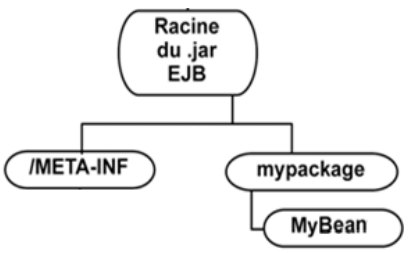
- Plusieurs types de clients : les locaux (local) et les distants (remote)
- L'interface remote est destinée aux clients distants
  - ne s'exécute pas dans la même machine virtuelle que celle du serveur
  - tous les appels de méthodes sont susceptibles de remonter des exceptions de type java.rmi.RemoteException qui pourraient être levées suite à une erreur de connectivité entre le client et le conteneur d'EJB
- L'interface local est destinée aux clients locaux
  - s'exécutent dans la même machine virtuelle que le serveur
    - d'autres EJB ou une servlet s'exécutant sur le même serveur
  - ses méthodes n'ont pas à remontrer d'exception, RMI n'intervenant pas dans le processus de communication avec un client local

Les méthodes de l'interface métier

- Leur nom est libre tant qu'il ne commence pas par « ejb » pour éviter tout conflit avec les méthodes callback interceptors
- Elles doivent être déclarées comme étant public
- Elles ne doivent pas être déclarées comme final ou static
- Dans le cas d'un accès de type remote, le type de retour et le type des arguments doivent être compatibles avec des transports de type RMI/IIOP

Déploiement

- On construit une archive .jar qui contient les fichiers compilés, et un répertoire META-INF (vide à ce stade)



Le cycle de vie

- Instanciation : le conteneur s'occupe de cette instanciation par l'appel de la méthode newInstance() à partir de l'objet Class lié à la classe du Session Bean
  - la classe d'implémentation du Session Bean doit disposer d'un constructeur public, sans argument
- Analyse de l'instance afin de déceler les éventuelles injections de dépendance à effectuer
  - permet au conteneur d'initialiser des propriétés de l'EJB automatiquement
  - grâce à des annotations
- Exécuter les méthodes callback interceptors
  - @PostConstruct qui intervient après toutes les dépendances d'injection effectuées par le conteneur et avant le premier appel de la méthode métier
  - @PreDestroy qui est appelé au moment où l'instance du Bean est détruite (lors de la suppression du Bean ou à l'arrêt de l'application)
  - @PrePassivate (Stateful uniquement) permet de spécifier une méthode appelée par le conteneur EJB lorsque qu'il juge nécessaire de passer le Bean dans un état sérialisable
  - @PostActivate (Stateful uniquement) : permet de spécifier la méthode qui sera appelée lorsqu'un Bean devra être réactivé de son état de passivation

Le client

- Le client utilise l'API JNDI pour localiser les beans
  - JNDI fonctionne comme la recherche d'un nom dans un annuaire
- Il connait l'interface Remote
  - il utilise les méthodes définies dans cette interface

Utilisation de JNDI

- Dans l'API JNDI, la classe InitialContext est définie et représente que point de départ de toute recherche.
  - la méthode lookup de cette classe retourne un Object, son paramètre est une chaîne de caractères : le nom de l'interface
- Il faut ensuite convertir le type de cet objet pour qu'il correspond réellement à l'interface recherchée
- Exemple : CoursManager cm= (CoursManager) initcont.lookup("CoursManager") ;
- Remarque : ne pas oublier de gérer les exceptions qui peuvent être levées
- On utilise ensuite les méthodes métier proposées par l'interface

Les beans entités

- **Données (dont information persistante) + comportement : règles métier associées à ces données**
- **Indépendance vis à vis du moyen de stockage, conservation de l'information persistante => favorise l'évolutivité**
- **Une instance de bean correspond à un enregistrement dans un BD => synchronisation entre la création d'un bean et l'ajout dans la base, et une modification dans le bean et une action sur la base**
- **En EJB3, développement très simple grâce aux annotations et à la configuration par exception**
  - **suppression des interfaces liées aux Entity Beans**
  - **utilisation d'un Session Bean pour gérer l'accès aux données**
  - **impossible d'accéder directement à l'Entity Bean depuis l'application cliente**

La classe de l'entité

- Les Entity Beans sont des POJO
  - l'état persistant est représenté par les variables d'instances
- Mais cette classe doit respecter certaines règles
  - peut être abstraite ou concrète, peut aussi hériter d'une classe entité que d'une classe non-entité
  - les méthodes, les propriétés et la classe ne doivent pas être finales
  - doit implémenter java.io.Serializable si une instance est envoyée à un client
  - doit posséder un constructeur sans argument public ou protégé
    - peut aussi posséder des constructeurs surchargés
    - but : simplifier l'instanciation de la classe par le conteneur

```
@Entity
public class Cours {
//...
}
```


Annotations d'un bean entité

- @Entity : information pour le conteneur
  - situé au niveau de la classe
  - permet de définir le nom de l'entité via l'attribut name
    - doit être unique dans une application
- Mapping base de données
  - par défaut, un Entity est mappé sur une table de même nom que la classe
  - annotation @Table avec l'attribut name permet de spécifier un nom de table différent

Les champs persistants

- Toute variable d'instance non «static» et non «transient» est automatiquement considérée comme persistante par le conteneur
  - il n'est pas nécessaire d'annoter les propriétés pour les rendre persistantes
  - pour spécifier un champ non persistant, il faut annoter son getter avec @Transient
- 2 types d'annotations : liées aux propriétés ou liées aux colonnes
  - peuvent se placer directement sur les champs ou sur les accesseurs
- La clé primaire pour les types simples : annotation @Id devant le champ concerné
- Annotation @Column pour surdéfinir les valeurs par défaut
  - name : nom de la colonne liée (nom de la propriété par défaut)
  - unique : si la valeur est unique dans la table ou non
  - nullable : précise si la colonne accepte des valeurs nulles ou non

Unité et contexte de persistance

- Unité : boîte noire qui permet de rendre persistants les Entity Beans
  - caractérisée par un ensemble d'Entity Beans, un fournisseur de persistance et une source de données
  - son rôle :
    - savoir où et comment stocker les informations
    - s'assurer de l'unicité des instances de chaque entité persistante
    - gérer les instances et leur cycle de vie via le **gestionnaire d'entité**
  - la persistance des informations doit être configurée
- Le contexte de persistance regroupe un ensemble d'entités en garantissant que, pour une même entité, il n'y aura qu'une seule instance d'objet

L'entity Manager

- Certains objets sont persistants et d'autres restent temporaires
  - le développeur décide de sauvegarder, modifier, supprimer les instances
- La manipulation des données se réalise via un Session Bean, qui accède au contexte de persistance et travaille ensuite avec les EB via l'Entity Manager
- Plusieurs manières d'obtenir un objet EntityManager
  - injection par annotation : par défaut, le conteneur instancie l'Entity Manager et gére son cycle de vie : container-managed entity manager
  - possible de gérer manuellement le cycle de vie : application-managed entity manager, utilisation de la fabrique EntityManagerFactory

```
@Stateless
public class MonServiceBean implements MonService {
@PersistenceContext(unitName="MonUniteDePersistance")
protected EntityManager em;
//...
}
```


Utiliser l'Entity Manager

- Enregistrer les entités : méthode em.persist(Object o)
  - l'instance devient managée et son insertion en base est mise dans la file d'attente de l'Entity Manager
- Retrouver les entités : à partir de leur clé primaire ou avec les requêtes EJB-QL
  - deux méthodes avec les mêmes paramètres : la classe de l'entité et l'instance de la clé primaire
  - <T> T find(Class<T> entity, Object primaryKey) retourne null si aucune entité associée à la clé primaire demandée
    - exemple : em.find(Cours.class, 1);
  - <T> T getReference(Class<T> entity, Object primaryKey) lance une javax.persistence.Entity-NotFoundException si aucune entité associée à la clé primaire
- Modifier les entités : deux façons
  - instance attachée : la charger depuis la base de données (via find(), getReference() ou par une requête) puis la modifier au sein de la transaction
  - instance détachée : méthode merge
  - les modifications sont réellement affectées en base à la fin de transaction ou sur appel explicitement de flush()
- Supprimer les entités : méthode em.remove()
  - suppression effective à l'appel de flush() ou à la fermeture du contexte de persistance => possible d'annuler la suppression entre-temps
- Synchronisation explicite : méthode flush
  - avec les méthodes persist(), merge() ou remove(), les changements ne sont pas synchronisés immédiatement

Cycle de vie du contexte de persistance

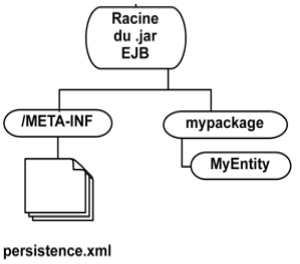
- Soumis à une durée de vie, lié à un Session Bean
- Un Entity Bean est attaché à un contexte de persistance ou détaché
  - attached : les modifications appliquées à l'objet sont automatiquement synchronisées avec la base de données, via l'Entity Manager
  - detached : l'Entity Bean n'a plus aucun lien avec l'Entity Manager
- Deux comportements différents selon le Session Bean :
  - transaction-scoped persistence context : durée de vie liée à une seule transaction
    - cycle de vie géré par le conteneur => totale transparence pour l'application
    - s'initialise lorsque l'application demande un Entity Manager au sein d'une transaction et s'arrête quand la transaction est validée ou annulée
    - si un Entity Manager est appelé en dehors de toute transaction, le contexte est créé et supprimé respectivement au début et à la fin de la méthode
  - extended persistence context : durée de vie liée à celle d'un Stateful Session Bean (et donc plusieurs transactions)
    - durée de vie gérée par l'application de la création jusqu'à la suppression

Configurer l'unité de persistance

- Fichier persistence.xml
- Balise racine <persistence> ne contient que des balises <persistence-unit>
- <persistence-unit> : déclare une unité de persistance
  - attribut name affecte un nom unique à cette unité dans l'application
    - utilisé pour identifier l'unité de persistance dans l'annotation @PersistenceContext pour la création d'EntityManager
  - attribut type : si l'unité de persistance est gérée dans une transaction Java EE (JTA par défaut) ou si gestion manuelle des transactions
- Balises à l'intérieur de persistence-unit
  - <provider> : définit la classe d'implémentation du fournisseur (unique) de persistance ; possibilité d'utiliser le fournisseur par défaut du SA
  - <jta-data-source> : définit le nom JNDI de la BD paramétrée sur le serveur
  - <properties> : configure des attributs dépendant du fournisseur de persistance
    - ensemble de propriétés <property> attributs name et value

Déploiement

- L'unité de persistance est assemblée dans une archive EJB .jar
  - le fichier persistence.xml dans le répertoire META-INF
  - les classes doivent être à la racine du fichier JAR
- Le conteneur analyse l'ensemble des classes du fichier JAR pour identifier les classes annotées avec @Entity



Transaction-scoped persistence context

```
@Stateless
public class UserSB implements GestionUser {
        @PersistenceContext(unitName="ExempleUP")
        EntityManager em;

        public User createUser(User user) {
                em.persist(user);
                user.setValid(false);
                return user;
        }
```

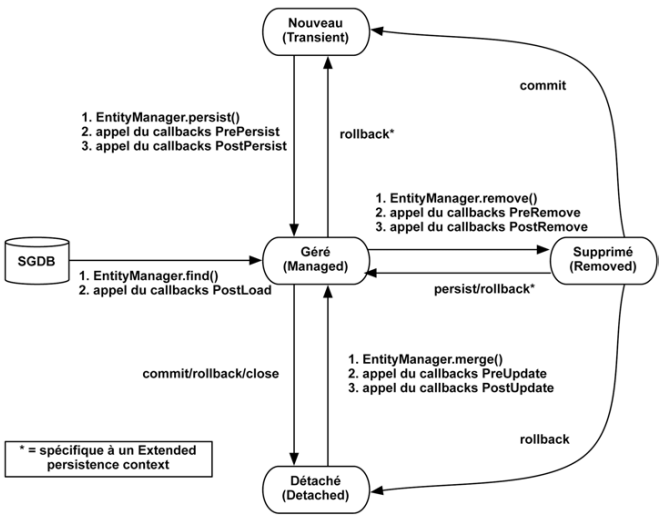
Extended persistence context

- Il est possible de définir un contexte de persistance avec une durée de vie s'étalant sur plusieurs transactions

```
@PersistenceContext(unitName="BiblioPU",type=PersistenceContextType.EXTENDED)
```

- Chaque instance d'Entity Bean managée par ce type de contexte le reste même après la fin d'une transaction
  - les instances sont détachées à la fermeture du contexte
- La durée de vie d'un contexte de persistance étendu doit être gérée par l'application (de la création jusqu'à la suppression)
- Cette gestion peut être automatiquement gérée avec un Stateful Session Bean
  - ce composant garde un état conversationnel entre les appels de méthodes

États d'un Entity Bean



Les méthodes de rappel

- Ces méthodes seront appelées par le fournisseur de persistance quand une instance passe à une autre étape de son cycle de vie
  - avant ou après insertion ou suppression ou modification de la base ...
- Ces méthodes appartiennent à une classe entité
  - elles sont alors sans argument et ne retournent rien (void)
- Ces méthodes peuvent être définies dans une classe *listener* associée
  - elles prennent alors en paramètre l'entité associé
  - il faut assigner cette classe listener à l'entité en utilisant l'annotation @EntityListeners sur la classe de celui-ci
    - @EntityListeners({Classe_entité.class})

Annotations des callbacks

- Pour spécifier au conteneur à quel moment du cycle de vie doivent être appelées ces méthodes, il faut les annoter
- @PrePersist : quand persist (ou merge) s'est terminé avec succès
- @PostPersist : après l'insertion dans la BD
- @PreRemove : quand remove est appelé
- @PostRemove : après suppression dans la BD
- @PreUpdate : avant modification dans la BD
- @PostUpdate : après modification dans la BD
- @PostLoad : après la lecture des données de la BD pour construire une entité
- Une méthode peut être annotée par plusieurs de ces annotations

Activation et passivation

- Synchronisation : maintenir l'équivalence entre les données enregistrées dans la base et celles contenues dans le bean
- Passivation : processus de sauvegarde du bean et déchargement
- Activation : processus de rechargement du bean et restauration de son état
- L'opération de Load est systématiquement invoquée avant l'exécution d'une méthode métier (afin de s'assurer d'avoir la dernière version des infos de la base en mémoire)
- Il peut être nécessaire de définir un code si besoin de mise à jour de données non persistantes

Lazy loading

- Les propriétés peuvent être chargées plus tard, uniquement lorsque le programme tente d'y accéder
  - par exemple lorsqu'un champ est de type collection
  - fonctionnement par défaut pour les propriétés multi-valuées
- Si une propriété simple persistante doit être chargée à la demande, on utilise l'attribut fetch de l'annotation @Basic afin de forcer le chargement en lazy

```
@Basic(fetch=FetchType.LAZY)
public String getCVContent() { return cvContent; }
```

- Problème lorsqu'un client veut accéder à une propriété "lazy"
  - pour pallier cela, on s'assure que toutes les propriétés de l'entité utilisée par le client sont initialisées avant que l'objet soit détaché du contexte de persistance
  - par exemple en appelant une méthode d'une collection pour l'initialiser

L'EJB-QL

- Entreprise JavaBean Query Language : spécification de langage de requêtes, intégré au système EJB
- Portable : utilisé à l'identique quelle que soit la version du SGBD
- Repose sur une abstraction du langage SQL et est traduit en « vrai » SQL lors de son exécution
- On utilise les objets des Entity Beans directement dans les requêtes
- Le fournisseur de persistance traduira ces requêtes en « vrai » SQL lors de l'exécution

Le schéma abstrait

- EJB-QL s'appuie sur le schéma abstrait des Entity Beans
- C'est une représentation interne des entités et de leurs relations, utilisée par EJB-QL pour naviguer entre les propriétés persistantes et relationnelles
- Le terme abstrait distingue ce schéma du schéma physique de la base de données
  - dans une base de données relationnelle, le schéma physique correspond à la structure des tables et colonnes
  - en EJB-QL on travaille avec un schéma abstrait et non avec le nom des tables du schéma physique
- Par défaut, le nom de la classe d'un bean entité est utilisé

Le langage EJB-QL 3

- Une requête EJB-QL peut permettre une sélection (SELECT), modification (UPDATE) ou suppression (DELETE).
- Une requête EJB-QL est similaire à du SQL, avec les clauses suivantes :
  - SELECT : liste les Entity Beans et les propriétés retournées par la requête
  - FROM : définit les Entity Beans utilisés. Ceux-ci doivent être déclarés via l'expression AS
  - WHERE : permet d'appliquer des critères de recherche
    - on spécifie aussi bien des types Java ayant leur équivalent dans les bases de données (String, Integer, Double…) mais également des Entity Beans
    - dans ce dernier cas, lors du passage en requête native, le critère s'appliquera sur la clé primaire
- L'opérateur . pour naviguer entre propriétés et relations

  SELECTe.nom, e.salaire FROM Employé as e WHERE **e.id=5**

  SELECT e FROM Employé AS e WHERE **e.employeur.id=5**
- Pas de ; à la fin !

Les opérateurs

- BETWEEN : SELECT e FROM Employé AS e WHERE e.id BETWEEN 1500 AND 2000
- LIKE : SELECT e FROM Employé AS e WHERE e.prénom LIKE 'jean%'
- IS NULL : SELECT e FROM Employé AS e WHERE e.login IS NULL
- MEMBER OF : SELECT s FROM Société AS s WHERE :user MEMBER OF s.salariés
- EMPTY : SELECT s FROM Société AS s WHERE s.salariés IS EMPTY
- La clause ORDER BY permet de ranger par ordre les résultats d'une requête
- Les fonctions d'agrégation pour effectuer des opérations sur des ensembles

  SELECT count(user) FROM User AS user
- La clause GROUP BY /Having

  SELECT s.raison_sociale, count(s.salariés) AS nbrSalariés FROM Société AS s GROUP BY s.raison_sociale HAVING nbrSalariés > 10

Les jointures

- **Inner join:**

  select p from Personne p join p.enfants e where p.prénom = 'Jean'

  select p from Personne p, in(p.enfants) e where p.prénom = 'Jean'

- **Left outer join:**

  select p.Nom, e.age from Personne p left join p.enfants e

- **Fetch join:**

select distinct p from Personne p left join fetch p.enfants where p.prénom = 'Jean'

Le polymorphisme

- EJB-QL supporte nativement le polymorphisme, c'est-à-dire les requêtes portant sur une hiérarchie d'objets

- L'exécution d'une requête sur une classe de base retourne tous les enregistrements liés à cette classe et donc de ses sous-classes

L'API Query

- Jonction entre l'application et l'exécution de requêtes EJB-QL : javax.persistence.Query
- List getResultList() : exécute la requête et retourne l'ensemble des résultats de celle-ci

```
Query query = entityManager.createQuery("SELECT u.id, u.lastName FROM User As u");
List<Object[]> listUsers = query.getResultList();
for(Object[] valueArray : listUsers){
    Integer id = (Integer) valueArray[0];
    String name = (String) valueArray[1];
}
```

- Object getSingleResult() : exécute la requête et retourne un unique résultat
  - si le nombre de résultats est > 1, une exception NonUniqueResultException est levée
  - si aucun résultat n'est trouvé, une exception EntityNotFoundException est levée
- Query setMaxResults(int max) et Query setFirstResult(int first) : définissent respectivement le nombre maximal de résultats et l'index du premier élément à retourner

Utilisation de paramètres

Query setParameter(String pNom, Object val)

Query setParameter(int pIndex, Object val)

- Ces deux méthodes affectent la valeur val respectivement au paramètre pNom ou au paramètre placé à l'index pIndex
- Les paramètres indexés sont placés dans la requête via un point d'interrogation suivi d'un entier, dont la valeur commence par 0

  SELECT user FROM User AS user WHERE user.id=?0
- Les paramètres nommés sont placés grâce aux deux points suivis d'un identifiant

  SELECT user FROM User AS user WHERE user.id=:userId
- Ces méthodes retournent l'objet Query sur lequel elles travaillent, ce qui permet de les appeler en cascade

```
Query query = entityManager.createQuery("
            SELECT user FROM User AS user WHERE user.id=:userId AND user.firstName=?0");
List<User> listUsers =
            query.setParameter("userId", new Integer(5)).setParameter(0, "Alex").getResultList();
```

Modification de la base

- Possibilité d'exécuter des requêtes « UPDATE » (modification) ou « DELETE » (suppression)
- Méthode executeUpdate() : exécute la requête qui doit être de type UPDATE ou DELETE
  - retourne le nombre d'enregistrements supprimés ou modifiés

```
Query query =
entityManager.createQuery("DELETE FROM User user
                                        WHERE user.login = '%s%'");
System.out.println(query.executeUpdate() + " enregistrement(s) supprimé(s)");
```


Les requêtes nommées

- Contrairement aux requêtes dynamiques (vues précédemment), elles ont l'avantage de pouvoir être « précompilées » lors du déploiement et donc de s'exécuter plus rapidement
- On utilise l'annotation @javax.persistence.NamedQuery ; définit un nom et une requête EJB-QL associée

```
@NamedQuery ( name="findAllUsers",
        query="SELECT user FROM User user WHERE user.lastName LIKE :lastname")
@Entity public class User { ... }
```

- Pour exécuter cette requête, on utilise la méthode EntityManager.createNamedQuery(), qui prend en paramètre le nom de la requête nommée

```
@PersistenceContext
public EntityManager em;
public List<User> findUserByName(String name) {
    List<User> userList =
    em.createNamedQuery(" findAllUsers",User.class).setParameter("lastname",name).getResultList();
    return userList;                                    }
```

- Pour définir plusieurs requêtes nommées pour un même bean entité, on le regroupe via @NamedQueries

```
@NamedQueries ( { @NamedQuery( name="findAllUsers", query="SELECT user FROM User AS user"),
                  @NamedQuery(name="findUserByLogin", query="…") } )
```

Gestion de transactions

- Service clé pour le développement côté serveur qui permet à des applications de fonctionner de manière robuste
- Transaction : opération atomique, bien que composée de plusieurs petites opérations
  - Ex : transfert de compte à compte bancaire : on enlève sur un compte, on dépose sur l'autre…
- Une solution possible : traitement par exceptions
  - problèmes : si on échoue à nouveau en remettant en cohérence, si le nombre d'opérations imbriquées est important, si le problème est dû à une panne réseau, SGBD ou machine (impossible de savoir si modification avant ou après le crash), ...
- Partage de données : on ne veut pas de perte d'intégrité des données

Les propriétés ACID

- **A**tomicité
  - la suite d'opérations est indivisible, tous les participants indiquent si la transaction s'est bien passée (« commit à 2 phases »)
- **C**onsistance
  - le système demeure cohérent après l'exécution d'une transaction
- **I**solation
  - empêche les transactions concurrentes de voir des résultats partiels, chaque transaction est isolée des autres
- **D**urabilité
  - garantit que les mises à jour sur une BD peuvent survivre à un crash (BD, machine, réseau)
  - en général, on utilise un fichier de log qui permet de revenir dans l'état avant le crash

Modèles de transactions

- Deux modèles
- *Flat transactions* ou transactions à plat : le plus simple
  - supportées par les EJBs
  - on démarre une transaction avant d'effectuer les opérations ; si toutes les opérations sont ok, transaction et opérations sont validées (permanent changes), sinon elle échoue et opérations annulées (rolled back)
- *Nested transactions* ou transactions imbriquées (ex billets d'avion)
  - non supportées par les EJBs pour le moment
  - une transaction peut inclure une autre transaction
  - abort de la globale => abort de toutes les imbriquées mais abort d'une imbriquée n'implique pas l'abort des autres transactions imbriquées

Gestion des transactions avec les EJB

- Le code (éventuel) écrit reste à un très haut niveau
  - simple choix d'un commit ou d'un abort
  - c'est généralement le container qui fait tout le travail
- Trois façons de gérer les transactions
  - par programmation : contrôle très fin possible
    - le concepteur du bean décide dans son code du begin, du commit et du abort
  - de manière initiée par le client
    - le concepteur du client décide du begin, du commit et du abort
    - ajoute une couche de sécurisation en plus, qui permet de détecter les crashes
  - de manière déclarative

La gestion déclarative

- Tout le traitement est à la charge du container
  - le bean est automatiquement enrôlé (enrolled) dans une transaction
- Le concepteur du bean utilise les annotations ou le descripteur de déploiement pour configurer ses choix : attributs transactionnels
  - on peut spécifier des attributs pour le bean entier ou méthode par méthode, ou les deux : le plus restrictif est appliqué
  - chaque méthode doit être traitée (globalement ou individuellement), comportement par défaut prédéfini
- Intérêt :
  - facile à programmer mais granularité importante
  - le code transactionnel n'est pas écrit dans la logique métier => respect la séparation des niveaux, augmente la clarté du code et favorise la maintenance
- Portée transactionnelle :
  - désigne l'ensemble des beans (session et entité) qui participent à la même transaction

Transactions et entités

- Une entité n'accède pas à la BD à chaque appel de méthode, mais à chaque transaction
- Si une entité s'exécute trop lentement, la cause est peut-être qu'une transaction est démarrée pour chaque appel de méthode de l'entité, impliquant des accès BD
- Solution : inclure plusieurs appels de méthodes de l'entité dans une même transaction

Attributs transactionnels

- Required
  - la méthode du bean doit être invoquée dans la portée d'une transaction. Si le client fait partie d'une transaction, le bean définissant la méthode est inclus dans la portée transactionnelle. Sinon, une nouvelle transaction démarre qui ne couvre que le bean et ceux auxquels il accède. Elle se termine à la fin de la méthode
  - comportement par défaut
- NotSupported :
  - l'invocation d'une méthode ayant cet attribut transactionnel suspend la transaction jusqu'à ce que la méthode soit terminée
  - après exécution la transaction originale reprend son exécution
- Supports :
  - inclut la méthode dans la portée transactionnelle si elle est invoquée dans une transaction => le bean qui définit cette méthode et tous ceux auxquels il accède dans la méthode font partie de la transaction originale, si elle existe.
  - si elle n'existe pas, pas de portée transactionnelle ni pour le bean ni pour ceux auxquels il accède

Attributs transactionnels - suite

- RequiredNew
  - une nouvelle transaction est toujours démarrée, que le client fasse partie d'une transaction ou pas. Si le client fait partie d'une transaction, celle-ci est suspendue jusqu'au retour de la méthode. La nouvelle portée transationnelle ne couvre que le bean contenant cette méthode RequiredNew et ceux auxquels il accède; elle s'arrête à la fin de la méthode.
- Mandatory
  - La méthode doit toujours faire partie de la portée transactionnelle du client. Si celui-ci ne fait pas partie d'une transaction, échec de l'invocation et lancement de javax.transaction.TransactionRequiredException
- Never
  - La méthode ne doit jamais être invoquée dans une transaction. Si c'est le cas, lancement de RemoteException. Sinon, exécution de la méthode sans transaction.

Exemple

```
@Stateless
@TransactionManagement(javax.ejb.TransactionManagementType.CONTAINER)
public class CompteBean implements SessionSynchronization {
        @PersistenceContext(unitName="MonUniteDePersistance")
        protected EntityManager em;
        @TransactionAttribute(javax.ejb.TransactionAttributeType.REQUIRED)
        public void transfertDeFonds(...){
        ...
        }
}
```


Attributs et types de beans

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

La gestion par programmation

- Plus complexe à manipuler mais plus puissante
- Le développeur doit utiliser Java Transaction API (JTA)
- Dans une transaction de nombreux partis sont impliqués : le driver de DB, le bean, le container,
  - premier effort pour assurer les transactions dans un système distribué : le service CORBA Object Transaction Service (OTS)
  - ensemble d'interfaces pour le gestionnaire de transactions, le gestionnaire de ressources, …
- Sun Microsystems a encapsulé OTS en deux API distinctes
  - JTS s'adresse aux vendeurs d'outils capables d'assurer un service de transactions, elle couvre tous les aspects complexes d'OTS,
  - JTA s'adresse au développeur d'application (vous) et simplifie grandement la gestion de transactions en contexte distribué.

Java Transaction API (JTA)

- JTA permet au programmeur de contrôler la gestion de transaction dans une logique métier
- Il pourra faire les begin, commit, rollback, etc…
- Il peut utiliser JTA dans des EJB mais aussi dans n'importe quelle application cliente
- JTA se compose de deux interfaces distinctes
  - une pour les gestionnaires de ressources X/Open XA (hors sujet)
  - une pour le programmeur désirant contrôler les transactions : javax.transaction.UserTransaction

L'interface UserTransation

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


Transactions initiées par le client

- Servlet par exemple
- Il est nécessaire d'obtenir une référence sur un objet UserTransaction, fourni par JTA
  - Via JNDI !

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

Niveau d'isolation

- Le niveau d'isolation limite la façon dont les transactions multiples et entrelacées interfèrent les unes sur les autres dans une BD multi-utilisateurs
- Trois types de violations possibles
  - lecture impropre (ou brouillée) : une transaction lit des données écrites par une transaction concurrente non validée
  - lecture ne pouvant être répétée : une transaction relit des données qu'elle a lu précédemment et trouve que les données ont été modifiées par une autre transaction (validée depuis la lecture initiale)
  - lecture fantôme : une transaction ré-exécute une requête renvoyant un ensemble de lignes satisfaisant une condition de recherche et trouve que l'ensemble des lignes satisfaisant la condition a changé du fait d'une autre transaction récemment validée
- Choix du niveau d'isolation en fonction de ce qu'on sait des exécutions possibles de l'application
  - on doit toujours choisir entre deux stratégies
  - pessimiste : on pense qu'il va y avoir des problèmes, on prend donc un verrou lors des accès BD, on travaille sur les données, puis on libère le verrou
  - optimiste : on espère que tout va bien se passer, et si la BD détecte une collision, on fait un rollback de la transaction

Comment spécifier ces niveaux ?

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

```
DataSource ds = jndiCtxt.lookup("java:comp/env/jdbc/mabd");
ds.getConnection();
```

- Transactions gérées par le container
  - impossibilité de spécifier le niveau d'isolation, ni par annotation ni par descripteur !
  - on le fera via le driver JDBC, ou via les outils de configuration de la DB ou du container
  - problèmes de portabilité

Rollback et bean Stateful

- Informer le client en cas de rollback
  - on peut envoyer une exception au client pour le tenir au courant de ce qu'il se passe pendant l'exécution d'une transaction
- Que faire de l'état conversationnel du bean ?
  - on risque d'avoir un état incorrect
  - lors du design, il faut prévoir la possibilité de restaurer un état correct
  - le container ne peut le faire car le traitement est en général spécifique à l'application, mais peut aider à réaliser cette tâche

L'interface SessionSynchronization

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

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

- Uniquement pour les session beans stateful dont les transactions sont gérées par le container
- Le container appelle afterCompletion() que la transaction se soit terminée par un commit ou par un abort
  - le paramètre de la méthode signale dans quel cas on se trouve

Exemple

```
@Stateful
public class CompteBean implements SessionSynchronization {
    private int val;
    private int oldVal;
  public CompteBean(int val) {
      this.val=val;
      this.oldVal=val;
    }
  public void afterBegin() { oldVal = val;}
  public void beforeCompletion() {}
  public void afterCompletion(boolean commited) {
      if (! commited)
        val = oldVal;
    }
  public int Compte() {
      return ++val;
    }
}
```


La gestion des exceptions

- **Deux types d'exceptions : système ou applicative**
- **Les exceptions système sont de type java.lang.RuntimeException ou java.rmi.RemoteException et de leurs sous-types, y compris EJBException**
- **Une exception applicative correspond à toute exception qui n'étend pas java.lang.RuntimeException ou java.rmi.RemoteException.**
- **Les transactions**
  - **sont automatiquement révoquées si une exception système est lancée à partir d'une méthode d'un bean**
  - **ne sont pas automatiquement révoquées si une exception applicative est lancée**
- **Dans le cas où les transactions sont gérées par le bean, les méthodes de la classe UserTransaction (commit, rollback, ...) permettent d'adopter l'attitude appropriée en cas d'exception**
- **Dans les autres cas (transactions gérées par le conteneur), on utilisera**
  - **soit une annotation sur la classe de l'exception qui annule la transaction dans tous les cas**
  - **soit l'EJBContext qui fournit une interface à la transaction**

Transaction annulée par exception

```
public class MonException extends RuntimeException {}
Ou
@ApplicationException(rollback=true)
public class MonException extends Exception {}

public void traitement() throws MonException {
    …
    if(problem) throw new MonException();
}
```


Transaction annulée via le contexte

- Lorsque la transaction est gérée par le conteneur
- On utilise la méthode du contexte setRollbackOnly()
- Accéder au contexte du Bean :

  ```
  @Resource SessionContext ctx;
  @Resource EntityContext ctx;
  ```

- Marquer la transaction comme annulée

  ```
  ctx.setRollbackOnly();
  ```

- Consulter l'état

  ```
  ctx.getRollbackOnly()
  ```

- Remarque : le code de la transaction se termine, annulation à la fin

Configuration d'environnement

- On utilise le descripteur de déploiement : fichier ejb-jar.xml dans le répertoire META-INF

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar version="3.0" xmlns=http://java.sun.com/xml/ns/javaee
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee ../../../testapp/ejb-jar_3_0.xsd ">
    <enterprise-beans>
        <session>
            <ejb-name>BeanQuiUtiliseLaRessource</ejb-name>
            ...
        </session>
    </enterprise-beans>
</ejb-jar>
```

- Définir des ressources externes/entrées d'environnement : valeurs similaires à des propriétés que le bean peut lire lors de l'exécution
  - balise env-entry
    - env-entry-name : on y accèdera en utilisant le chemin complet java:comp/env dans une recherche via JNDI
    - env-entry-type : peut être String ou un des types enveloppe (Integer, Long, …)
    - env-entry-value
- Le bean retrouve ces entrées de l'environnent en utilisant JNDI, ou en utilisant une annotation

```
@Resource(name = "nomVariableEnvironnement")
int NbMaxAutorisé;
```


Les relations entre beans entités

- Une entité ne travaille généralement pas seule ; elle est reliée à d'autres entités
- Pour définir un champ relationnel au sein d'une entité, il suffit de créer une propriété dont le type est une entité (cardinalité 1) ou un ensemble d'entité (cardinalité n
- Une relation est dite unidirectionnelle si une seule partie connaît la relation, ou bidirectionnelle si les deux parties la connaissent

Un à Un (One to One)

- Pour associer deux entités avec ce type de relation, il faut utiliser l'annotation @OneToOne
- Mappé de plusieurs manières dans la base de données
  - on utilise les mêmes valeurs pour les clés primaires des deux entités
    - on précise ce choix via l'annotation @PrimaryKeyJoinColumn (jointure par clé primaire)

```
@Entity public class User {//...
@OneToOne
@PrimaryKeyJoinColumn
    public AccountInfo getAccountInfo() {
        return accountInfo;
    } //... }
```

  - on utilise une clé étrangère d'un côté de la relation
    - la colonne de cette clé doit être marquée comme unique
    - on utilise l'annotation JoinColumn

```
@OneToOne
@JoinColumn(name="account_id", referencedColumnName="id")
    public AccountInfo getAccountInfo() {
        return accountInfo;
    }
```

Champ dans le bean AccountInfo

Un à Plusieurs et Plusieurs à Un

- On utilise l'annotation @ManyToOne et dans le cas d'une relation bidirectionnelle, l'autre côté doit utiliser l'annotation @OneToMany
- L'atttribut mappedBy spécifie le champ propriétaire de la relation dans le cas d'une relation bidirectionnelle

```
@Entity
public class User implements Serializable {
//...
    private Collection<Account> accounts;

    @OneToMany(mappedBy = "user")
    public Collection<Account> getAccounts() {
        return accounts;
    }
//...
```

Plusieurs à Plusieurs

- On utilise des propriétés multi-valuées de chaque côté de la relation (si bidirectionnelle) et on les annote avec @ManyToMany
  - en relationnel, cette relation impose l'utilisation d'une table d'association

```
@Entity
public class User { //...
    private Collection<Hobby> hobbies;
    @ManyToMany
    @JoinTable(name = "USER_HOBBIES",
                joinColumns=@JoinColumn(name= user_id", referencedColumnName="id"),
                inverseJoinColumns=@JoinColumn(name="hobby_id",referencedColumnName="id"))
    public Collection<Hobby> getHobbies() {
    return hobbies;
    }
//... }
```


Plusieurs à Plusieurs - suite

```
@Entity
public class Hobby {
//...
        private Collection<User> users;
        @ManyToMany(mappedBy="hobbies")
        public Collection<User> getUsers() {
                return users;
        }
        public void setUsers(Collection<User> users)
        {
                this.users = users;
        }
}
```