EJB 3.1

Université Française d'Egypte
Richard Grin
Version 0.16 – 13/10/12

Plan du support (1/2)

- Généralités
- Fonctionnement des EJB
- Types d'EJB
- Beans session sans état
- Beans session avec état
- Concurrence, appels asynchrones
- Injection
- Contexte des EJB

Plan du support (2/2)

- Transactions
- Exceptions
- Descripteur de déploiement
- Variables d'environnement
- Compléments
- Clients des EJB
- API pour container embarqué

Généralités

Entreprise Java Beans

- Fait partie de la spécification Java EE
- Standard pour développer des objets métier, les EJB, composants Java pour les serveurs

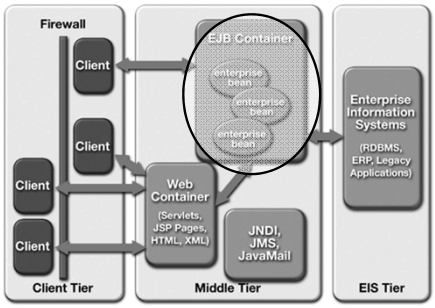
Qu'est-ce qu'un EJB

- Composant réutilisable du « *middle tiers* », placé sur le serveur, entre la base de données et les clients
- Classe annotée par une annotation ou désignée dans un fichier XML de configuration **`ejb-jar.xml`**

Interfaces EJB

- Un EJB « s'expose » à des clients (classes Java qui vont l'utiliser) par des interfaces qui contiennent les méthodes que les clients pourront appeler
- Le container EJB joue les intermédiaires entre les clients et le composant EJB

Application multi-tiers - containers



Serveur d'application

- Il permet l'utilisation d'EJB par des clients pendant l'exécution des programmes
- Il fournit des services aux EJB et gère les conteneurs d'EJB
- Il fournit au moins les services
  - de nommage
  - d'authentification
  - d'accès HTTP
  - d'accès aux EJB

Services fournis par les serveurs d'application

- Nommage, utilisable par injection ou par JNDI : les programmes peuvent désigner les objets et les ressources par un nom, sans connaître leur emplacement
- Authentification : oblige les utilisateurs à prouver leur identité quand ils se connectent
- Accès HTTP : permet l'accès aux servlets et JSP (contenus dans un container Web)

Services offerts par un container d'EJB

- Gestion du cycle de vie : création et suppression des EJB (avec possibilité de callbacks) ; gestion de pools
- Gestion de la sécurité : seuls les clients autorisés peuvent appeler les méthodes des EJB
- Gestion de la persistance : conserver des informations d'une session à l'autre, même en cas de redémarrage du serveur
- Gestion des threads : création, répartition (équilibrage), appels asynchrones de méthodes

Services offerts par un container d'EJB

- Appels de méthodes à distance
- Gestion de la concurrence
- Gestion des transactions : les programmes peuvent démarrer et arrêter des transactions JTA ; les transactions peuvent utiliser plusieurs bases de données de différents types
- Gestion de la charge : les EJB peuvent être distribuées sur plusieurs machines
- Interception lors des appels de méthodes

Services offerts par le container

- Injection de dépendance
- Réception de messages JMS

Avantages des EJB

- Le développeur peut se concentrer sur les traitements « métier »
- Il n'a (presque) plus à s'occuper des problèmes non fonctionnels, techniques, liés aux transactions, à la sécurité, aux threads,…
- Les composants sont portables ; ils peuvent être utilisés avec tous les serveurs d'applications qui respectent la spécification EJB

Intégration des EJB dans une application Web

- Les pages JSF (Java Server Faces) se chargent de l'interface avec l'utilisateur
- Les servlets contrôlent le flot d'exécution
  - aiguillage des requêtes de l'utilisateur
  - choix des pages JSF à présenter à l'utilisateur (servlet « JSF » caché à l'utilisateur)
- Les EJB s'occupent de la partie « métier » et de l'interface avec les bases de données (en utilisant les entités persistantes)

Intégration des EJB dans une application Web

- Des Java Beans (POJO) sont utilisés par les servlets et les EJB pour
  - faire certains traitements
  - encapsuler des données qui sont échangées entre les divers éléments (servlet, JSF, EJB) (*data transfert objects*, DTO)

Interfaces des EJB

Principe de fonctionnement des beans

- Les clients n'accèdent jamais directement aux méthodes implémentées dans le bean
- Le container intercepte toujours les messages pour fournir ses services ; il délègue ensuite au bean pour l'exécution du code spécifique
- Les clients envoient donc toujours les messages « aux interfaces » et pas au bean lui-même
- Ce sont des objets d'une classe créée par le container qui recevront les messages

Autour d'un EJB

- L'EJB proprement dit, ou « bean Entreprise », est la classe qui contient l'implémentation (écrite par le développeur)
- L'interface distante (*remote*) ou locale est l'interface de l'EJB exposée aux clients (optionnelle en EJB 3.1)

Exemple

```
@Remote
public interface Hello {
  public String ditHello(String nom)
         throws RemoteException;
}
```

```
@Stateless
public class HelloBean implements Hello {
  public String ditHello(String nom){
    return "Hello, " + nom;
  }
}
```

Même si la classe a d'autres méthodes publiques, elles ne seront pas utilisables par les clients de l'EJB

Interface « Remote »

- L'interface doit être annotée avec **`@Remote`** si on veut que ses méthodes soient utilisables à distance (depuis une autre JVM)
- Annoter une interface avec **`@Remote`** implique que le passage des paramètre de type non primitif se fera en copiant les valeurs des objets (sérialisation cf. RMI)
- En fait, lorsque l'EJB est sur la même JVM, des serveurs d'application peuvent faire un appel local de la méthode mais ça n'est pas assuré

Interface locale

- Si l'interface n'est pas annotée par **`@Remote`**, elle est locale (on peut aussi l'annoter avec **`@Local`**)
- Une interface locale ne permet l'accès que depuis la même JVM où s'exécute l'instance de l'EJB
- Elle offre des meilleures performances puisqu'elle évite de passer par le réseau et qu'elle permet de passer des objets par référence, sans sérialisation

Vue « no-interface »

- Depuis EJB 3.1, un bean session peut ne pas implémenter d'interface
- En ce cas, le bean expose toutes les méthodes publiques de la classe (« *no-interface view* ») comme une interface locale

Exemple – vue « no-interface »

Pas d'**`implements`**

```
@Stateless
public class HelloServiceBean {
  public String ditHello(String nom){
    return "Hello, " + nom;
  }
}
```

La méthode **`ditHello`** peut être appelée par les clients de l'EJB (annotation de la classe par **`@LocalBean`** est optionnelle)

Locale ou distante, il faut choisir

- Une interface ne peut être à la fois locale et distante (remote)
- Si on veut une interface qui puisse servir pour les 2 cas, le plus simple est de créer une interface « ordinaire » qui contient toutes les méthodes et de créer 2 interfaces, une locale et l'autre distante, qui héritent de cette interface ordinaire

@LocalBean

- Si le bean expose au moins une interface et souhaite tout de même exposer ses méthodes publiques, il doit être annoté avec **`@LocalBean`** :

```
@Stateless
@LocalBean
public class Bean implements HelloService {
  public String ditHello(String nom) {
    return "Hello, " + nom;
  }
  public String autreMethode(...){...}
}
```


@Remote pour le bean

- Le bean lui-même peut être annoté avec **`@Remote`** avec le nom de l'interface en paramètre
- Exemple

```
@Stateless
@Remote(RemoteBanque.class)
public class Banque ...
```

- Ça peut être utile dans le cas où il implémente une interface qui n'est pas annotée (car créée sans penser aux EJB)

Présentation rapide des différents types d'EJB

2 types d'EJB

- Session (3 types : avec ou sans état, singleton)
- Dirigé par les messages (*Message-Driven Bean*, MDB)

Classe d'un bean session

- Doit être public, pas final ou abstract
- Doit avoir un constructeur sans paramètre public
- Ne doit pas avoir de méthode finalize()
- Les méthodes « métier » (qui implémentent les interfaces « métier ») ne doivent pas commencer par « ejb » et ne peuvent être final ou static
- Les arguments et les valeurs de retour des méthodes « remote » doivent être des types RMI (types primitifs ou sérialisables)

3 types d'EJB session

- Avec ou sans état et singleton

Bean session sans état (1)

- Pour exécuter une tâche avec une seule méthode
- L'interaction avec le client commence par l'appel d'une méthode de l'interface « métier » et se termine avec la fin de la méthode
- Un bean session ne conserve aucune information entre 2 appels de méthode ; toutes les informations nécessaires au traitement doivent être passées en paramètre des méthodes appelées
- Exemple d'utilisation : afficher le solde du compte du client numéro 1345567

Bean session sans état (2)

- Fait partie d'un pool dont les instances sont partagées entre plusieurs clients
- Un bean session sans état est annoté avec **`@Stateless`**

Bean session avec état (1)

- Pour exécuter une tâche qui nécessite plusieurs appels de méthode
- Correspond à une conversation engagée par un client avec le serveur
- La conversation commence au moment où le client obtient une référence vers le bean session et se termine lorsque le client abandonne cette référence
- Conserve des informations sur le client entre 2 appels de méthode

Bean session avec état (2)

- Ne peut être partagé entre 2 clients
- Peut être temporairement enlevé de la mémoire centrale (passivation - activation)
- Exemple : un caddy qui contient les produits achetés par un client pendant la session
- Il est bon de le supprimer lorsque la session est finie (pas toujours possible ; sinon, il est supprimé au bout d'un certain temps par le serveur)
- Un bean session avec état est annoté avec **`@Statefull`**

Singleton (1)

- Ajouté dans le norme EJB 3.1
- Instancié qu'une seule fois par container EJB
- Annoté avec **`@Singleton`**
- Peut être instancié au démarrage du serveur d'applications (intéressant si long à initialiser) (**`@Startup`**)
- On peut spécifier un ordre dans l'instanciation des singletons s'ils sont interdépendants (**`@DependsOn`**)

Singleton (2)

- Possible de spécifier le comportement pour les accès concurrents (`@ConcurrencyManagement`) :
  - Géré par le container (on peut indiquer le comportement pour chaque méthode avec `@Lock`)
  - Géré par le bean (le programmeur du bean gère)
  - Interdit (exception levée si accès concurrent)

Bean dirigé par les messages

- Message Driven Bean (MDB)
- Écoute les messages asynchrones (JMS) qui lui sont envoyés
- Sans état
- Sans interface ; les clients y accèdent directement
- Etudié dans un autre support de cours

EJB Lite (1)

- Sous –ensemble des EJB qui exclut
  - Les MDB
  - Les interfaces distantes
  - Les services Web
  - Les appels asynchrones
  - Les timers
  - Le support des EJB 2.x
  - L'interopérabilité avec CORBA
- Le plus souvent une application Web peut se contenter des EJB Lite

EJB Lite (2)

- Plus facile à implémenter (le container embarqué ne supporte que EJB Lite)
- Application Web peut être distribuée dans un simple fichier war si elle ne contient que des fonctionnalités de EJB Lite (fichier ear si elle contient d'autres fonctionnalités de EJB)

Distribution des applications

- Les applications sont distribuées dans des fichiers qui ont le format « jar » de base, avec des ajouts
- Une application Web qui n'utilise que les EJB Lite peut se distribuer avec un seul fichier « war » qui contient la partie Web (JSF en particulier) et les EJB
- Sinon, l'application doit être distribuée dans un fichier « ear » qui contient 0 ou plusieurs fichiers war pour la partie Web et 0 ou plusieurs fichiers jar pour la partie EJB

Bean session sans état

Exemple – classe du bean

```
@Stateless
public class HelloServiceBean
      implements HelloService {
  public String ditHello(String nom){
    return "Hello, " + nom;
  }
}
```


Cycle de vie d'un EJB session

- Le serveur d'applications décide quand il doit créer ou supprimer un bean session
- Pour le créer il utilisera le constructeur sans paramètre
- Une instance d'un bean session sans état peut être réutilisée par plusieurs clients

Initialisation

- Le constructeur n'est pas assuré de disposer des ressources du serveur d'application
- Pour les initialisations du bean qui nécessitent des services du serveur (par exemple une connexion à une base de données) on n'utilise donc pas le constructeur
- On utilise plutôt une méthode callback annotée par **`@PostConstruct`** qui sera appelée par le serveur d'applications après la création d'une instance du bean et quand tous les services seront disponibles

A la suppression

- Une méthode annotée par **`@PreDestroy`** permet de libérer les ressources éventuelles que le bean aurait acquises durant son existence
- Toutes les ressources acquises dans la méthode **`@PostConstruct`** qui nécessitent une fermeture doivent être rendues dans cette méthode avec la fermeture appropriée

Déclaration en XML

- Toutes les annotations relatives aux EJB peuvent être remplacées par des déclarations dans les fichiers XML de description de déploiement (fichier META-INF/ejb-jar.xml du jar qui contient les EJB)
- Il est plus simple d'utiliser les annotations
- En cas de contradiction, les déclarations dans les fichiers XML l'emportent sur les annotations

Exemple de déclaration XML

```
<ejb-jar>
  <entreprise-beans>
    <session>
      <ejb-name>HelloBean</ejb-name>
      <local>HelloService</local>
      <ejb-class>fr…HelloServiceBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
    ...
  </entreprise-beans>
  ...
</ejb-jar>
```

Bean session avec état

Interfaces

- Comme pour un bean session sans état, la classe du bean implémente une ou plusieurs interfaces locales ou distantes (ou aucune interface depuis EJB 3.1)

Création et suppression

- Comme pour un EJB session sans état, un EJB session avec état peut comporter des méthodes **`@PostConstruct`** et **`@PreDestroy`**

Passivation (1)

- La correspondance 1 client – 1 bean session avec état peut occasionner l'existence de nombreux beans à un moment donné
- Le container peut sauvegarder l'état du bean pour le retirer momentanément de la mémoire centrale (passivation) et le réactiver plus tard (activation) ; la classe doit donc être sérialisable

Passivation (2)

- Les ressources utilisées par le bean qui ne peuvent être sérialisées doivent être acquises à l'activation et fermées à la passivation avec les méthodes annotées par **`@PostActivate`** et **`@PrePassivate`**

`@Remove`

- Invoquer une méthode annotée par **`@Remove`** supprime le bean session avec état

@StatefulTimeout

- Prend en paramètre le nombre de millisecondes le bean peut rester sans appel de méthode de la part du client
- Exemple :

```
@Stateful
@StatefulTimeout(50000)
public class Caddy {
```


Interface SessionSynchronization

- Si un bean stateful implémente cette interface il est averti quand une transaction commence et se termine
- Cette interface est rarement utilisée mais peut être utile dans certains cas particuliers
- Les méthodes peuvent lancer les exceptions javax.ejb.EJBException (non contrôlée), java.rmi.RemoteException (contrôlée)

Interface SessionSynchronization

- L'interface contient les méthodes
  - afterBegin() : lancée au début d'une transaction ; peut être utilisé par lire des données de la base et les garder dans un champ du bean (optimisation)
  - beforeCompletion() : lancée juste avant la fin de la transaction ; peut être utilisé pour écrire dans la base les données « cachées » par afterBegin
  - afterCompletion(boolean) ; le paramètre est à true s'il y a eu un commit, à false s'il y a eu un rollback

Concurrence
Appels asynchrones

Concurrence

- La spécification EJB interdit au code écrit pour un EJB de gérer les threads (créer, suspendre, reprendre, changer le nom ou la priorité d'un thread est interdit)
- En effet, la gestion des threads doit être laissée au container qui les utilise au mieux pour administrer l'environnement d'exécution
- Depuis Java EE 6 il est cependant possible et simple d'indiquer qu'une méthode doit être exécutée en parallèle (en laissant la gestion du nouveau thread au container)

@Asynchronous

- Une méthode (ou la classe) d'un bean session peut être annotée par **@Asynchronous**
- La méthode (ou toutes les méthodes de la classe annotée) sera alors appelée d'une manière asynchrone : le container lancera l'exécution de la méthode dans un autre thread et redonnera immédiatement la main au code qui a appelé la méthode
- Intéressant pour les méthodes qui sont longues à exécuter comme les envois d'emails ou les impressions

Contexte transmis

- Le contexte de sécurité est transmis à la méthode asynchrone (pour savoir si l'utilisateur a le droit d'exécuter les méthodes appelées)
- Le contexte transactionnel n'est pas transmis ; un attribut REQUIRES va donc créer une nouvelle transaction et un attribut MANDATORY va lever une TransactionRequiredException

Type retour

- Le type retour de la méthode ne peut être que **`void`** ou **`Future`** (interface du paquetage **`java.util.concurrent`**)
- **`Future`** sert à récupérer une valeur calculée par la méthode (méthode **`get`** ; voir cours sur la concurrence) ou une exception lancée par la méthode (enveloppée dans une **`ExecutionException`**)
- EJB fournit la classe **`javax.ejb.AsyncResult`** qui implémente **`Future`**

Interruption de la méthode (1)

- **`Future`** permet d'empêcher le démarrage de la méthode si elle n'a pas déjà commencé avec la méthode **`cancel`**
- Cette méthode prend un paramètre égal à **`true`** si on veut indiquer qu'on veut essayer d'interrompre la tâche même si elle a déjà commencé

Interruption de la méthode (2)

- Si la tâche a déjà commencé et que le client a voulu interrompre une tâche déjà commencée, celle-ci peut utiliser la méthode **`wasCancelCalled()`** de la classe **`SessionContext`** (le plus souvent injectée par **`@Resource`** dans le code du bean session) pour savoir si le client a essayé d'interrompre la tâche
- Si c'est le cas, la tâche peut s'interrompre « gentiment » (voir exemple à suivre)

Exemple

```
@Resource SessionContext sc;
...
public Future<Integer>
envoyerCommande(Commande commande) {
  if (sc.wasCancelCalled()) {
    return new AsyncResult<Integer>(-1);
  }
  // Traitement de la commande
  ...
  return statutTraitement;
}
```


Injection

Avec ou sans CDI

- La spécification CDI offrent de nombreuses possibilités pour injecter des instances, avec de nombreuses fonctionnalités ; un support de cours à part étudie CDI
- La spécification EJB 3.1 offre aussi des possibilités moins importantes pour injecter des beans ou des ressources

Injection d'EJB

- L'annotation @EJB (paquetage javax.ejb) permet d'injecter un EJB dans une classe gérée par un container
- Le plus souvent, on injecte ainsi :

```
public class MaClasse {
  @EJB
  private InterfaceEJB monEjb;
```

- L'EJB sera injecté et enregistré dans JNDI avec le nom suivant :
  <nom-complet-de-MaClasse>/monEjb

Injection d'EJB avec XML

- On peut aussi injecter avec les fichiers de configuration XML
- Ca permet d'écraser une éventuelle annotation @EJB au moment du déploiement
- Voir <ejb-local-ref> des fichiers ejb-jar.xml (pas étudié ici)

Annotation @EJB

- Elle permet d'injecter simplement un EJB
- Avec des attributs elle permet aussi d'indiquer une référence vers un EJB dont on donne le nom JNDI, ce qui permettra de le rechercher par un lookup dans le registre, dans le code Java de la classe qui est annotée

Exemple

```
import javax.naming.InitialContext;
@Stateful
@EJB(name="ejbs/unNomJNDI",
     beanInterface=UneInterface.class,
     beanName="AutreEJB")
public class MonEJB implements MonInterface {
  ...
  InitialContext ctx = new InitialContext();
  UneInterface ejb = (UneInterface)
    ctx.lookup("java:comp/env/ejbs/unNomJNDI");
```


Injection de ressource

- **@Resource** sert essentiellement pour injecter une source de données mais peut aussi servir pour injecter d'autres types de ressources comme un serveur d'emails
- Exemple :

```
@Resource(name="jdbc/mabd")
private DataSource source;
```

Injection de ressource particulière

- Lié à JPA :
  - **`@PersistenceUnit`** : fabrique d'entity manager
  - **`@PersistenceContext`** : entity manager
- Lié aux service Web : **`@WebServiceRef`**

Contexte des EJB

Interface **`EJBContext`**

- Interface **`javax.ejb.EJBContext`**
- Interface avec le contexte extérieur :
  - l'utilisateur qui a appelé l'EJB (**`getCallerPrincipal`**, **`isCallerInRole`**)
  - si la transaction en cours a été marquée pour un rollback (**`getRollbackOnly`**) ou permet de la marquer pour un rollback (**`setRollbackOnly`**)
  - récupère le service timer (**`getTimerService`**)
  - ou récupère une ressource JNDI (**`lookup`**)
- Interface fille **`SessionContext`** pour les EJB session et **`MessageDrivenContext`** pour les MDB

Exemple

- Pour avoir le nom de l'utilisateur qui est actuellement connecté :

```
@Resource // Injection de ressource
private SessionContext context;
...
public void m(...) {
  String user =
    context.getCallerPrincipal().getName();
  ...
```


Interface **`SessionContext`**

- Sous interface de l'interface **`EJBContext`**
- Permet de savoir si un client a souhaité stopper l'exécution d'une méthode asynchrone de l'EJB session (wasCancelCalled), ou l'interface (locale ou distante) à travers laquelle l'EJB a été appelé (getInvokedBusinessInterface)
- Permet aussi d'avoir un « this » de l'EJB

« this » d'un EJB

- La méthode **`SessionContext.getBusinessObject`** est utilisée dans le code d'un EJB pour passer à un autre objet une référence à « lui-même », en fait à l'EJB Object
- On lui passe en paramètre l'interface locale ou distante (ce qui permettra le renvoi d'une référence qui convient)
- Ce « this » peut être passé à d'autres beans

Transactions dans les EJB

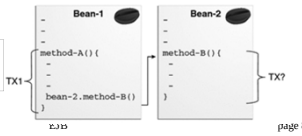
2 types

- Les transactions peuvent être gérées
  - par le container (CMT : *Container Managed Transaction*)
  - par le bean (BMT)

CMT

- Il faut indiquer ce que doit faire le container si une éventuelle transaction est en cours au moment de l'appel d'une méthode, ou si aucune transaction n'est en cours
- Cette indication peut être donnée par une annotation ou par une balise dans un fichier de déploiement XML

Image du tutoriel Java EE 6



CMT avec annotation

- Le plus simple est d'annoter les classes ou les méthodes avec **`@TransactionAttribute`**
- **U**ne annotation sur une classe donne une valeur par défaut pour toutes les méthodes de la classe
- Valeurs possibles : **`Never`**, **`NotSupported`**, **`Supports`**, **`Required`** (valeur par défaut), **`RequiresNew`**, **`Mandatory`**

Exemple

```
@TransactionAttribute(NOT_SUPPORTED)
@Stateful
public class Bean1 { ...
  @TransactionAttribute(REQUIRES_NEW)
  public void methode1() {...}

  @TransactionAttribute(REQUIRED)
  public void methode2() {...}

  public void methode3() {...}
}
```


CMT avec fichier de déploiement

- Il est aussi possible d'utiliser le descripteur de déploiement en donnant la valeur de l'attribut **`trans-attribute`** aux méthodes
- Ces valeurs l'emportent sur les annotations

Exemple dans le descripteur de déploiement ejb-jar.xml

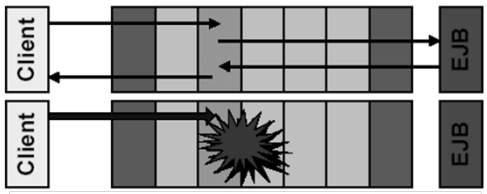
```
<container-transaction>
  <method>
    <ejb-name>rep/truc</ejb-name>
    <method-name>*</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
  <method>
    . . .
  </method>
</container-transaction>
```


Bloc <container-transaction>

- On peut avoir plusieurs blocs **`<container-transaction>`**
- Ils sont dans le bloc **`assembly-descriptor`** (et pas dans le bloc **`enterprise-beans`**)
- On peut désigner toutes les méthodes, les méthodes qui ont un certain nom, une méthode particulière en donnant son nom et les types de ses paramètres
- La désignation la plus spécifique l'emporte en cas de conflit

Never

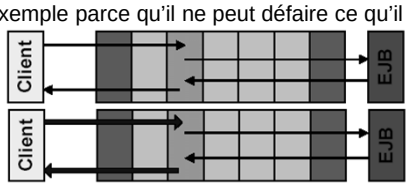
- Si le client appelle la méthode avec une transaction en cours, une exception sera levée



flèche noire : pas de transaction en cours, flèche bleue : une transaction en cours (container indiqué en gris foncé)

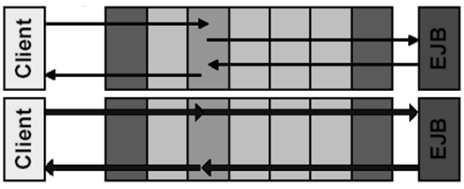
NotSupported

- Toute transaction en cours sera suspendue pendant l'exécution de la méthode
- Utile quand le développeur du bean sait que son code ne peut participer à une transaction, par exemple parce qu'il ne peut défaire ce qu'il a fait



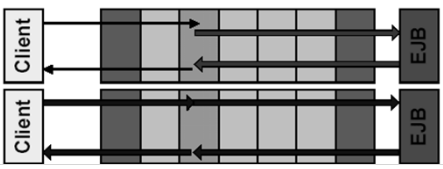
Supports

- Le bean peut supporter une transaction s'il y en a une mais très bien fonctionner sans
- A utiliser avec précaution…



Required

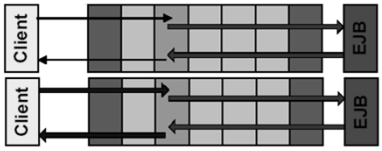
- Une nouvelle transaction sera créée par le container si aucune n'est en cours
- Si une transaction est en cours, elle sera utilisée par le bean



flèche rouge : une nouvelle transaction créée par le container

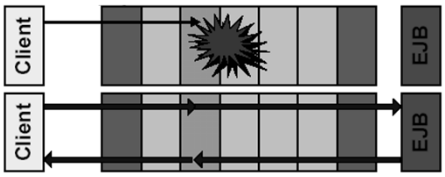
RequiresNew

- Une nouvelle transaction sera toujours créée par le container pour cette méthode
- Utile quand la méthode veut être sure que ses modifications seront validées ou non à la fin de son exécution



Mandatory

- Une exception sera levée si aucune transaction n'est en cours au moment de l'appel de la méthode



Nouvelle transaction en CMT

- Si une nouvelle transaction a été créée par le container au démarrage d'une méthode
- Au retour de la méthode le container la termine
  - par un rollback si une exception « système » a été levée par la méthode (voir section « Exceptions dans les EJB ») ou si la transaction a été marquée comme annulée (setRollbackOnly)
  - par un commit sinon

Marquer une transaction comme annulée

- Si le bean veut un rollback sans lever une exception « système », il peut marquer la transaction en cours comme annulée en appelant la méthode **`javax.ejb.EJBContext.setRollbackOnly()`**
- La transaction en cours ne pourra plus être validée par un commit

`setRollbackOnly`

- Cette méthode est réservée aux EJB dont les transactions sont gérées par le container
- Les autres beans peuvent appeler directement la méthode rollback ou setRollbackOnly de l'interface javax.transaction.UserTransaction
- Le container lancera une IllegalStateException si aucune transaction n'est en cours

`setRollbackOnly`

- Quand une transaction est marquée pour un rollback par setRollbackOnly, est-ce que les opérations qui suivent sont quand même exécutées ou est-ce que la méthode se termine tout de suite ?
- C'est au programmeur de s'arranger pour que les instructions suivantes ne soient pas exécutées (en lançant une exception par exemple) ; cette exception sera contrôlée car si elle était non contrôlée, on n'aurait pas besoin d'utiliser setRollbackOnly (voir section sur les exceptions)

Rollback et exceptions

- La levée d'une exception ne provoque pas toujours le rollback d'une transaction en cours ; voir section « Exceptions dans les EJB »

Rollback pour le client

- Quelle que soit la manière dont le rollback a eu lieu, le client recevra une exception qui lui indiquera qu'un rollback a eu lieu : **`javax.transaction.TransactionRolledbackException`** (classe fille de **`RemoteException`**, donc contrôlée)
- Si ce client a une transaction en cours, il peut décider d'attraper ou non cette exception
- S'il ne l'attrape pas, sa propre transaction sera invalidée car l'exception se propagera

BMT

- On choisit cette option quand on veut plus de précision ou de liberté dans la gestion des transactions
- Par exemple, si on veut des délimitations de transactions qui ne correspondent pas aux débuts et fins de méthodes

Contraintes pour le BMT

- Pour les beans sessions sans état et les beans dirigés par messages, les transactions doivent être validées ou invalidées avant le retour des méthodes vers le client
- Pour les beans sessions avec état, une transaction peut concerner plusieurs méthodes ; en ce cas, les méthodes doivent le plus souvent être appelées dans un ordre déterminé
- Si le bean géré par BMT est appelé avec une transaction en cours, celle-ci est suspendue

Niveau d'isolation des transactions

- On retrouve les 4 niveaux classiques :
  - Read uncommited
  - Read commited
  - Repeatable reads
  - Serializable

Inter-blocages

- Ils peuvent être détectés ou non ; ça dépend du serveur d'application
- Le serveur d'application peut, par exemple, les détecter et lèver une exception sur une des transactions impliquées, ce qui débloque les autres

Méthodes du contexte pour les transactions

- EJBContext :
  - boolean getRollbackOnly
  - void setRollbackOnly
  - javax.transaction.UserTransaction getUserTransaction()

Interface **`SessionSynchronization`**

- Si un bean stateful implémente cette interface il est averti au début et à la fin d'une transaction
- Cette interface a été étudiée dans la section « Bean session avec état »

Les clients et les transactions

- Souvent, c'est le client qui démarre ou stoppe une transaction
- En effet, les méthodes des EJB n'en savent souvent pas assez pour faire un rollback ou marquer la transaction comme annulée

Les clients et les transactions

- Le client doit commencer par récupérer une instance de javax.transaction.UserTransaction
- S'il peut faire de l'injection de ressource, c'est simple :

```
@Resource
private UserTransaction userTransaction;
```

- Ensuite le client peut gérer la transaction ; par exemple

```
userTransaction.begin();
```


JNDI et transaction

- Si le client ne peut faire de l'injection de ressource, la transaction peut aussi être obtenue par le service JNDI :

```
Context ctx = new InitialContext();
UserTransaction userTransaction =
 (UserTransaction)
  ctx.lookup("java:comp/UserTransaction");
```

- L'adresse JNDI dépend du serveur

`UserTransaction`

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

Exemple de code

```
userTransaction.begin();
try {
  delegue.transfert(5, 6, 100000000);
  userTransaction.commit();
}
catch(MouvementCompteException e) {
  System.out.println(e.getMessage());
  userTransaction.rollback();
}
finally {
  afficheCompte(delegue);
}
```


Exceptions dans les EJB

`RemoteException`

- Les EJB sont conformes au standard RMI-IOOP comme protocole de transport
- Toutes les méthodes qui peuvent être invoquées à distance doivent comporter **`throws java.rmi.RemoteException`**
- C'est une exception contrôlée par le compilateur (classe fille de **`IOException`**)

`EJBException`

- **`javax.ejb.EJBException`** est une exception « *runtime* » (pas contrôlée par le compilateur) spécialement dédiée aux problèmes sur les EJB
- Elle prévient qu'un problème inattendu n'a pas permis l'exécution normale de la méthode

Interception des exceptions

- Le container d'EJB « enveloppe » l'EJB ; il peut attraper une exception qui sort d'une méthode « métier » de l'EJB (définie dans une interface « métier ») ou d'un méthode callback (**`@PostConstruct`**, **`@PostActivate`**,…) pour l'envelopper avec une autre exception

2 types d'exception

- Exception liée au fonctionnement de l'application comme un compte bancaire qui n'est pas suffisamment approvisionné pendant un retrait d'argent
- Exception système comme un problème dans le fonctionnement du SGBD ou l'absence d'une ressource indispensable au traitement ; **`EJBException`** en est une
- Ces 2 types d'exceptions sont traitées différemment par le container d'EJB

Exception d'application

- Le container laisse passer une exception d'application telle quelle et ne fait rien de spécial
- Il ne lance pas un rollback, sauf si l'annotation @ApplicationException indique le contraire
- Le client de la méthode qui a lancé l'exception peut ainsi attraper l'exception et essayer de réparer le problème

Exception système

- Le container attrape les exceptions système, et
  - fait un rollback de la transaction en cours
  - met l'exception dans les log du serveur
  - enveloppe l'exception dans une **`RemoteException`** si le client est distant, ou une **`EJBException`** si le client est local (ou des sous-classes de ces classes)
  - relance cette nouvelle exception
  - le plus souvent le container enlève par précaution l'EJB du pool des EJB utilisables

Répartition des exceptions

- Normalement,
  - les exceptions contrôlées par le compilateur sont considérées comme des exceptions d'application
  - les exceptions non contrôlées sont considérées comme des exceptions système
- Pour changer cette répartition, il est possible d'annoter la classe d'une exception non contrôlée par @javax.ejb.ApplicationException

Annoter une classe d'exception

- **`@ApplicationException`** indique que l'exception doit être considérée comme une exception d'application (même si elle est non contrôlée)
- L'annotation a un attribut **`rollback`** qui peut avoir la valeur **`true`** ou **`false`** suivant qu'on veut que l'exception provoque ou non un rollback (**`false`** par défaut)

Envelopper avec **`EJBException`**

- Si on veut profiter du traitement automatique des exceptions système pour une exception contrôlée comme **`NamingException`** (qui est plutôt liée au système) dans le code Java on peut envelopper l'exception contrôlée dans une **`EJBException`**

Exception d'application et rollback

- N'entraîne pas nécessairement un rollback : le client doit pouvoir choisir par exemple un traitement substitutif pour terminer la transaction
- Si on veut forcer un rollback en CMT, on doit appeler la méthode **`setRollbackOnly`** avant de lancer l'exception
- L'attribut **`rollback`** de l'annotation **`@ApplicationException`** peut aussi déclarer que l'exception provoque automatiquement un rollback

Exemple

```
@ApplicationException(rollback = true)
public class TrucException extends Exception {
```


`TransactionRolledbackException`

- **`TransactionRolledbackException`** est une sous-classe de **`EJBException`**
- Elle indique que la transaction en cours a été annulée ou marquée pour l'annulation

Descripteur de déploiement

2 types d'information

- Information structurelle fournie par le fournisseur d'EJB : information de base sur les ejb qui composent l'application
- Information pour l'assemblage : par exemple pour définir les méthodes concernées par des intercepteur pour définir les autorisations d'accès à des méthodes

Fichier

- Le fichier **`ejb-jar.xml`** contient les informations
- La structure du fichier :

```
<ejb-jar>
  <entreprise-beans>
     . . .
  <entreprise-beans>
  <assembly-descriptor>
    . . .
  </assembly-descriptor>
</ejb-jar>
```


- Les descripteurs de déploiement sont bien plus légers que dans les premières versions de Java EE depuis l'arrivée des annotations

Sections pour les beans

- La section **`entreprise-beans`** contient les informations structurelles sur les beans
- Elle contient une section pour chaque bean de l'application
- Le nom de le section dépend du type du bean
  - **`session`**
  - **`entity`**
  - **`message-driven`**

Information sur chaque bean

- Pour chaque EJB, on a
  - son nom (**`ejb-name`**) : nom logique qui n'est pas forcément lié à l'adresse JNDI de déploiement
  - sa classe (**`ejb-class`**) : le nom complet de la classe qui implémente le bean
  - ses interfaces (**`remote`**, **`local`**) : noms complets des interfaces liées au bean

Informations particulières sur les bean

- Chaque type de bean a des informations particulières ; par exemple pour les beans sessions, son type (**`session-type`** peut avoir la valeur **`stateful`** ou **`stateless`**)

Variables d'environnement

- Le fournisseur d'EJB peut indiquer des variables d'environnement qui permettront de configurer le bean sans avoir à recompiler le bean
- Le déployeur d'EJB peut modifier ces valeurs lors du déploiement
- Ces valeurs sont rangées dans le descripteur de déploiement

Exemple

```
<env-entry>
  <env-entry-name>seuil</env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-
type>
  <env-entry-value>500</env-entry-value> </env-
entry>
<env-entry>
  <env-entry-name>pourcentage</env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-
type>
  <env-entry-value>10</env-entry-value>
</env-entry>
```

Injection d'une variable d'environnement

- Depuis EJB 3.0 il est possible d'injecter une variable d'environnement :

```
@Stateless public class EmployeeServiceBean
implements EmployeeService {
  ...
  // Suppose nom JNDI java:comp/env/seuil
  @Resource int maxExemptions;
  ...
  public void setMaxExemptions(int maxEx) {
    maxExemptions = maxEx;
  }
  ...
```


Code qui utilise une variable d'environnement

```
Context initial = new InitialContext();
Context env =
  (Context)initial.lookup("java:comp/env");
Double seuil = (Double)env.lookup("seuil");
Double pourcentage =
  (Double)env.lookup("pourcentage");
if (montant >= seuil.doubleValue()) {
  remise = pourcentage.doubleValue();
}
else {
  discount = 0.00;
}
```


Quelques compléments

Ce qui est interdit aux EJB

- Un EJB ne peut lancer de nouveaux threads (c'est le container qui gère les threads)
- L'utilisation de java.io est interdite depuis un EJB

Comment faire pour lire un fichier

- Comment faire si on veut qu'un EJB lise le contenu d'un fichier ? Par exemple un fichier qui contient des données pour initialiser l'application
- L'utilisation des API ordinaires pour les entrées-sorties est interdite
- A la rigueur, la lecture du fichier considéré comme une ressource (getResource) marchera le plus souvent
- Cependant il existe d'autres solutions

Autres solutions

- Ranger le contenu du fichier dans une base de données
- Associer un nom JNDI au fichier
- Un servlet peut lire le contenu et associer un nom JNDI à ce contenu. L'idée est de faire lire le contenu du fichier par la partie Web de l'application. Cette partie Web fait le travail ou bien passe les informations suffisantes à l'EJB pour faire le travail (en paramètre de la méthode de l'EJB appelée)

La sécurité

- La sécurité est traitée dans un support à part

API pour container embarqué

Utilité

- Surtout intéressant pour effectuer les tests unitaires des EJB sans la lourdeur du serveur d'applications
- Classe **`javax.ejb.embeddable.EJBContainer`**

Exemple (1)

```
public class HelloWorldServiceTest {
    private static EJBContainer ejbContainer;
    private static IHelloWorldService
                   helloWorldService;
    /**
     * Idéal pour démarrer l'EJBContainer et récupérer
     * les EJB à tester.
     */
    @BeforeClass
    public static void init() throws NamingException {
      // Code dans le transparent suivant
    }
```


Exemple (2 – méthode init)

```
Map<String, Object> properties =
  new HashMap<String, Object>();
// Modification éventuelle des propriétés
...
EJBContainer container =
  EJBContainer.createEJBContainer(properties);
Context ctx = container.getContext();
String helloWorldServiceName =
HelloWorldService.class.getSimpleName();
helloWorldServiceName = "java:global/classes.ext/"
                        + helloWorldServiceName;
helloWorldService = (IHelloWorldService)
                    ctx.lookup(helloWorldServiceName);
```

Exemple (3 – méthode de test)

```
/**
 * Vérifie que l'EJB HelloWorldService a bien été
 * récupéré et qu'il est fonctionnel
 */
@Test
public void should_say_hello_world() {
   Assert.assertEquals("Hello World",
                       helloWorldService.sayHello());
}
```


Exemple (4 – fermeture ressources)

```
/**
 * Fermer le contexte JNDI et l'EJBContainer.
 */
@AfterClass
public static void cleanup() throws NamingException {
   // Ferme aussi le contexte JNDI
   container.close();
}
```


Clients des applications EJB

Types de clients

- Ce sont presque toujours les éléments de l'application qui fournissent l'interface graphique avec l'utilisateur
- On trouve des
  - Clients « standalone » qui sont des applications Java qui utilisent des EJB placés du côté serveur
  - des clients légers sous forme de pages Web, HTML, JSF, JSP ou autres
  - des applets, à l'interface plus riche, utilisables dans un navigateur

Environnement pour les clients « standalone »

- Pour eux on doit installer un environnement minimum pour qu'ils utilisent les ressources et EJB placées du côté du serveur
- On doit tout d'abord initialiser JNDI pour qu'ils puissent obtenir des références aux EJB session qu'ils vont utiliser
- Il faut aussi leur fournir les bibliothèques pour utiliser les EJB

Bibliographie

- Enterprise JavaBeans 3.1, O'Reilly, de AndrewLee Rubinger et Bill Burke
- Beginning Java EE 6 Platform With Glassfish 3: From Novice to Professional de Antonio Goncalves, éditeur Apress, 2ème édition août 2010
- Tutoriel en ligne (gratuit) d'Oracle : http://download.oracle.com/javaee/6/tutorial/doc/ (ce cours utilise des images de ce tutoriel)
- Des images de ce cours ont été reprises du « Quick Start Guide » de JBoss 3.0 ; le nouveau guide pour 3.2 : http://jboss.org/docs/jbossj2ee.pdf