

Les applications n-tiers

La logique applicative

La logique applicative

La logique applicative est censée être principalement réalisée dans les **Enterprise Java Beans (EJB)**

... mais elle peut aussi être plus ou moins réalisée :

- par la couche Web pour les applications Web
- par la couche Client pour les clients riches / lourds
 - Web → Single Page Application, Rich Internet Application, ...
 - non Web (pas de couche Web)

Historique

Un peu d'histoire...

- première spécification des EJB : fin des années 1990
 - deuxième spécification : début des années 2000
 - problème : configuration très complexe car basée sur des fichiers XML (*deployment descriptors*)
 - troisième spécification : milieu / fin des années 2000
 - bases :
 - Hibernate pour persistance ==> **annotations** plutôt que fichiers XML
 - Spring pour CDI ==> injection de dépendances et de ressources
- ==> un EJB est un **POJO** / **POJI**

EJB et conteneur

Une entité JPA est un POJO qui peut s'exécuter sans être géré par un container JEE

- récupération manuelle d'un EntityManager
- gestion manuelle des transactions

A la différence des entités JPA, un EJB ne peut **pleinement** fonctionner que s'il est **déployé** dans un **container d'EJB**

- seules les applications JEE peuvent définir des EJB
- il faut un serveur d'application (JBoss/**WildFly**, **GlassFish**, IBM WebSphere, Apache TomEE, ...)

Test des EJB

Au fait :

- l'accès à la couche Business se fait :
 - par la couche Web en cas d'application Web
 - par la couche Client en cas d'application avec client lourd
- vous n'étudierez la couche Web que plus tard dans le module 6
- a priori vous ne disposez pas de client lourd

... comment faire pour tester nos EJB ?



QUIZZ

Test des EJB

Un EJB peut néanmoins être testé avec JUnit par exemple

- utilisation d'un framework de test (Arquillian, ...)
- utilisation d'un serveur d'application **embarqué** par JUnit
 - GlassFish en propose un, **mais pas** WildFly

Les types d'EJB

Deux types d'EJB à l'origine :

- **Session**
- Entity

L'EJB Entity est devenu l'entité JPA, ce n'est donc plus un EJB.

La version 2.0 de la spécification des EJB (2001) a introduit l'EJB **"Message-Driven"**

- service de communication par message (asynchrone)
- basé sur JMS (Java Message Service)

Les EJB Session

EJB Session : qu'est-ce qu'une **session** ?

Les EJB Session

Une **session** est un échange de durée finie entre un client (HTTP souvent, mais pas obligatoirement) et un serveur.

Exemples :

- un client HTTP de Topaidi demande la création d'un utilisateur
- un client HTTP de Topaidi demande la liste des idées enregistrées
- un client HTTP d'un site marchand réalise l'achat d'un article
- un client HTTP d'un site bancaire demande à effectuer une conversion de devises

Les EJB Session

Certains échanges ne requièrent pas l'enregistrement d'un état

- conversion de devises par exemple
 - le client envoie sa demande en indiquant un montant dans une première devise à convertir, l'identifiant de la première devise et l'identifiant de la seconde devise
 - le serveur retourne le montant converti dans la seconde devise
- si le client souhaite effectuer une nouvelle conversion, le même échange est reproduit : les 3 informations doivent être fournies de nouveau

Les EJB Session

D'autres échanges nécessitent l'enregistrement d'un état conversationnel

- achat d'un article sur un site marchand par exemple
 - l'utilisateur doit d'abord se connecter ==> une interaction
 - il demande à visualiser la liste des articles proposés ==> une autre interaction
 - il peut ensuite placer un article dans son chariot ==> encore une interaction
 - il réalise enfin le paiement de ce qu'il y a dans son chariot ==> encore une autre interaction

Les EJB Session

Qui gère cet état ?

QUIZZ

Les EJB Session

Le client aussi bien que le serveur peut gérer l'état conversationnel.

On verra en fin de module 6 les caractéristiques des systèmes selon l'endroit où est géré l'état.

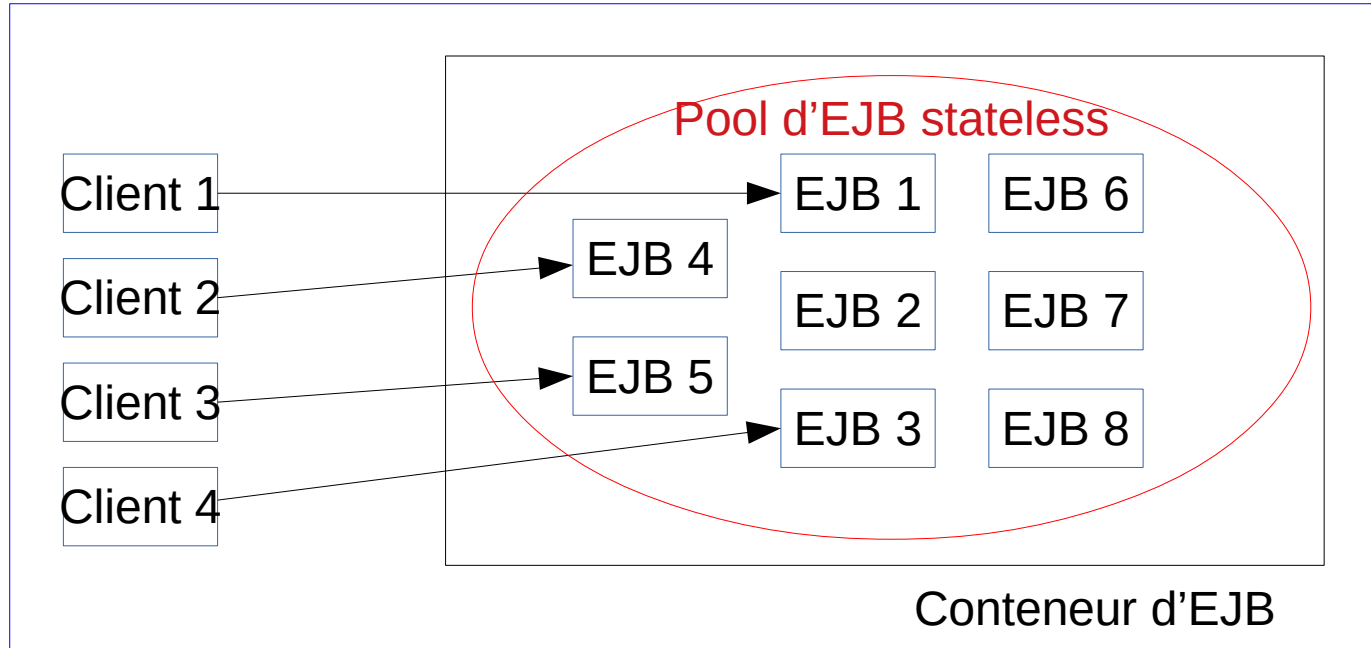
Les EJB Session

Il existe deux types principaux d'EJB Session :

- un EJB Session **stateful** enregistre l'état conversationnel entre un client et une application JEE
 - une instance d'un tel EJB est donc associée à un client précis
- un EJB Session **stateless** n'enregistre aucun état
 - une instance d'un tel EJB peut donc être utilisée par n'importe quel client

EJB Session stateless

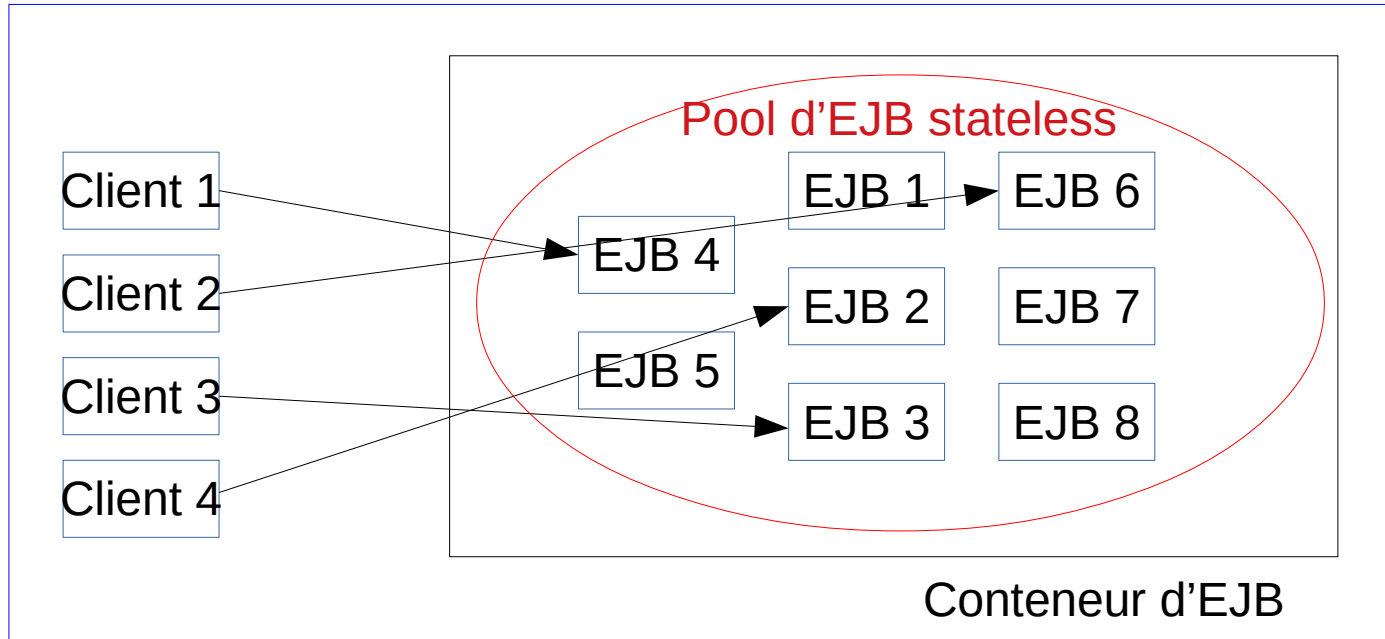
Premier échange



Serveur d'application

EJB Session stateless

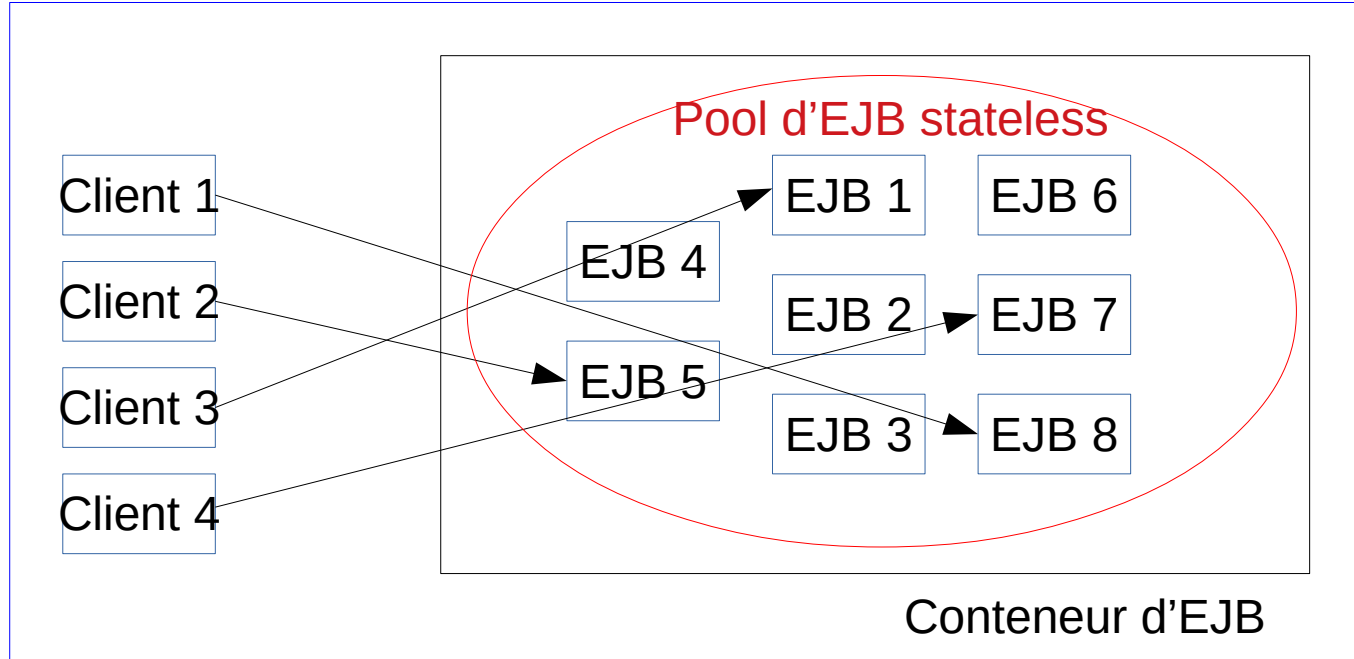
Deuxième échange



Serveur d'application

EJB Session stateless

Troisième échange



Serveur d'application

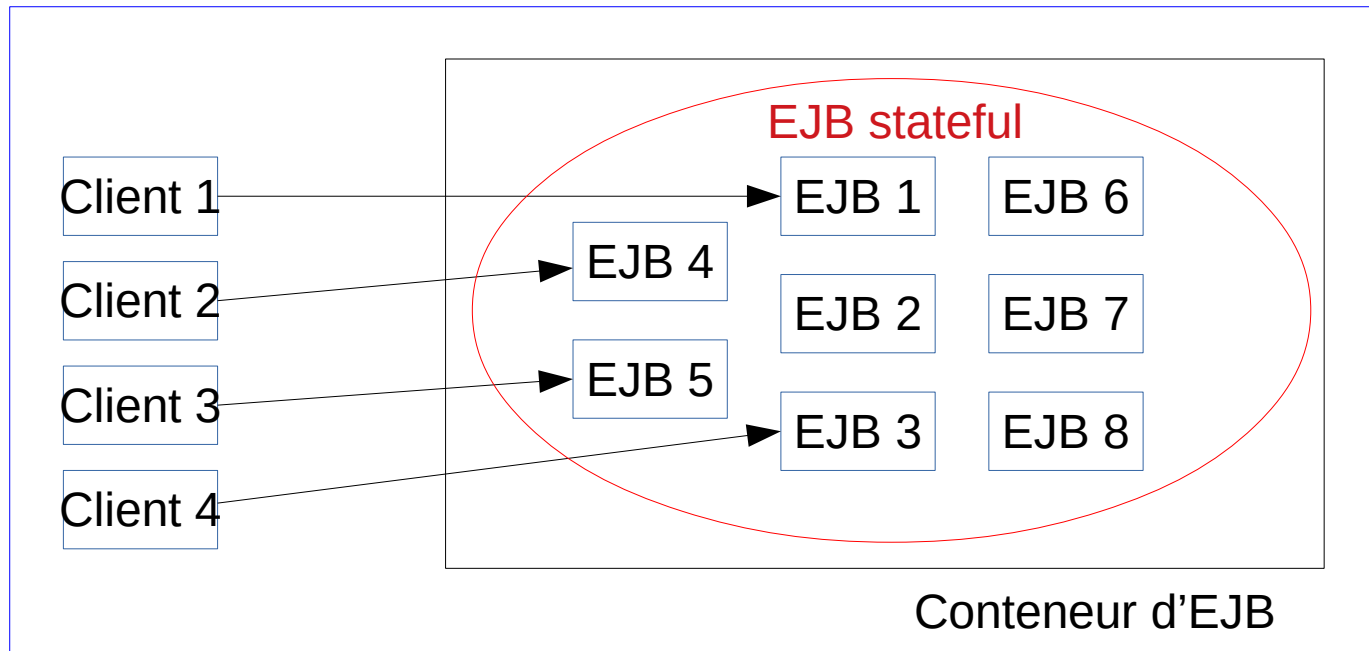
EJB Session stateful

Un EJB Session stateful associé à un client de l'application JEE sera toujours le même :

- tant que le serveur d'application n'est pas redémarré
- ou qu'un timeout (configurable) n'a pas été atteint.

EJB Session stateful

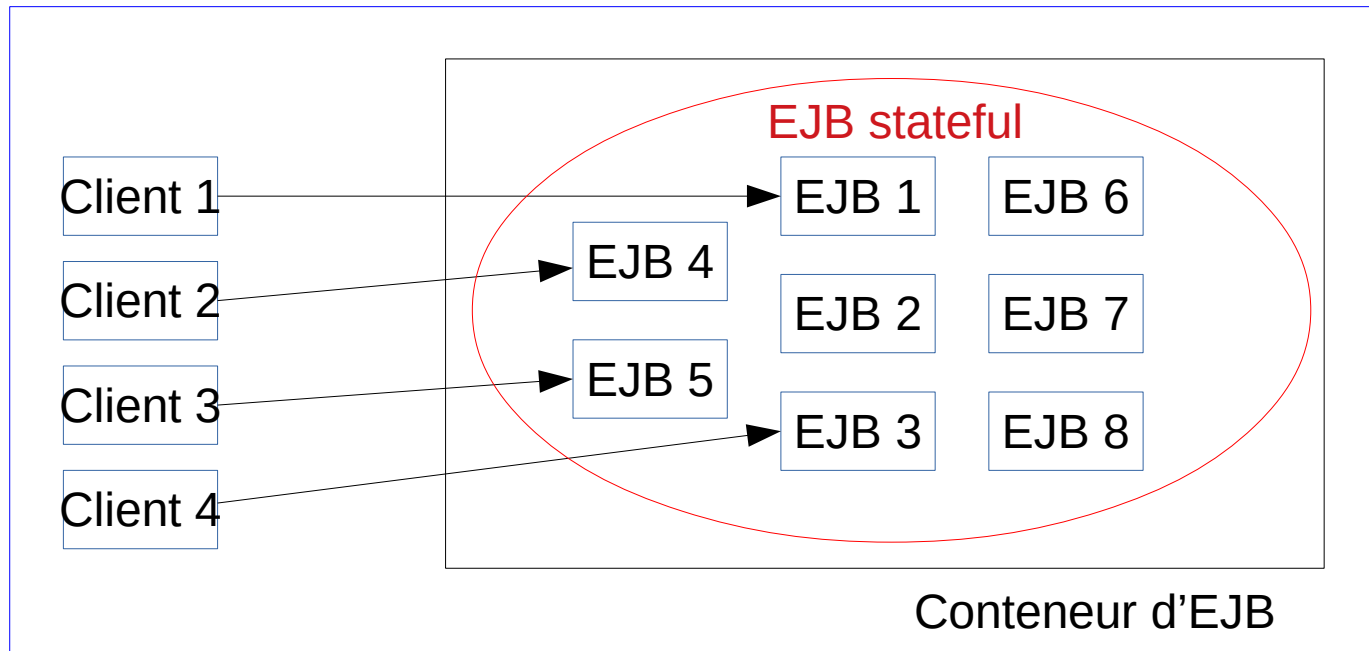
Premier échange



Serveur d'application

EJB Session stateful

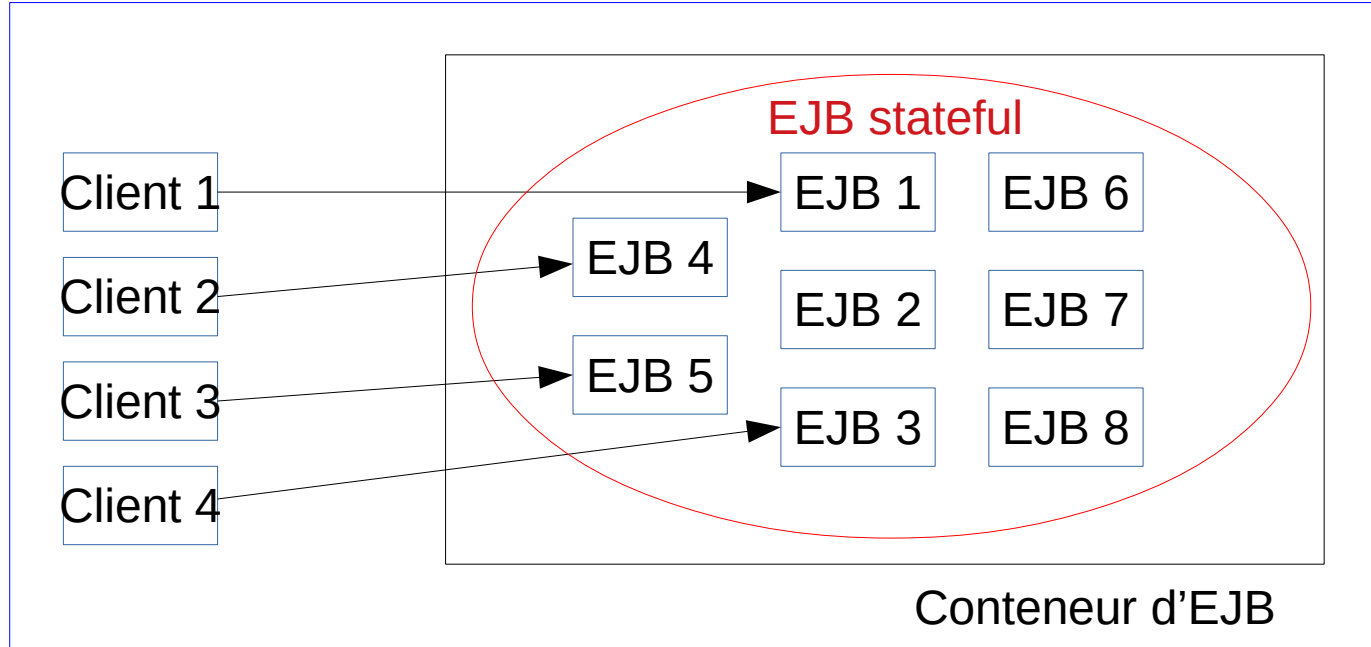
Deuxième échange



Serveur d'application

EJB Session stateful

Troisième échange



Serveur d'application

EJB Session stateful

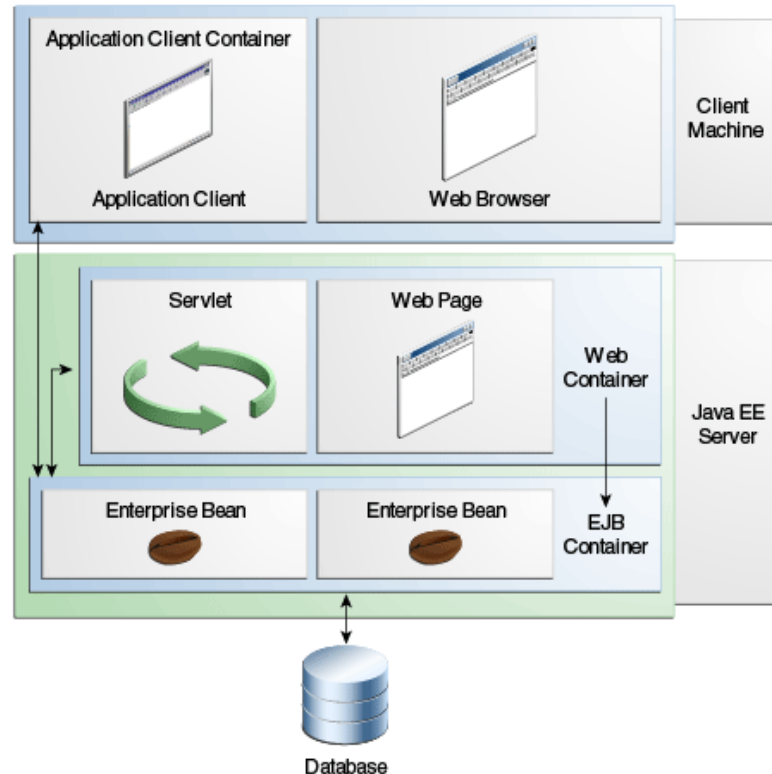
Un EJB Session stateful peut enregistrer des informations relatives au dialogue entre le client HTTP et l'application (**l'état conversationnel**) au moyen d'attributs de l'instance (variables de l'instance d'EJB).

Mais attention : le client de cet EJB n'est pas le client HTTP, c'est le composant **en relation directe** avec lui.

EJB Session stateful

Rappel de l'architecture d'une application n-tiers avec JEE

Dans ce schéma, les clients des EJB sont les composants gérés par le Web container ou l'application client container... pas le navigateur !



EJB Session singleton

Il existe un troisième type d'EJB Session : le **Singleton**.

Un singleton est spécifié au moyen de l'annotation `@Singleton`

Un EJB Session Singleton est un objet “applicatif métier” n'ayant qu'une instance dans l'application. Autrement dit, il est partagé par tous les objets EJB et entités de l'application.

Son état est persistant entre deux appels de service – mais pas après un arrêt du serveur.

Un singleton peut être créé automatiquement au lancement d'une application, au moyen de l'annotation `@Startup`

EJB Session singleton

@Singleton  *Déclaration d'un singleton*

@Startup  *Création du singleton au lancement de l'application*

```
public class MonSingleton {  
    private String etat;  
  
    public void setEtat(String etat) {  
        this.etat = etat;  
    }  
    public String getEtat() {  
        return this.etat;  
    }  
}
```

EJB Session singleton

N'y a t'il pas un risque lié à l'utilisation d'un singleton, si l'on considère qu'il possède un état ?

Rappel : la philosophie de Java est d'utiliser le multi-threading...

QUIZZ

EJB Session singleton

Vu qu'un singleton est visible de tous les objets de l'application, et qu'il possède un état, il y a le risque que des accès concurrents au singleton ne puissent aboutir.

Exemple : deux threads utilisent en parallèle un service du singleton qui modifie son état.

EJB Session singleton

Pour éviter cela, deux possibilités :

- laisser le container d'EJB contrôler les accès concurrents au singleton : annotation `@ConcurrencyManagement (CONTAINER)`
- contrôler par programme les accès concurrents : annotation `@ConcurrencyManagement (BEAN)`

Par défaut, la gestion des accès concurrents est faite par le conteneur d'EJB.

EJB Session singleton

Le contrôle de la concurrence peut être spécifié au niveau de chaque méthode, au moyen de l'annotation `@Lock` :

- `@Lock(READ)` : les clients peuvent accéder à la méthode en parallèle (typiquement, ils ne font que lire des données en parallèle, il n'y a donc pas de souci)
- `@Lock(WRITE)` : seul un client peut accéder à la méthode, les autres clients sont mis en attente; utilisé pour les méthodes qui modifient (write) l'état du singleton
 - la durée d'attente avant timeout peut être configurée au moyen de l'annotation `@AccessTimeout` (avec durée en paramètre)

EJB Session singleton

L'annotation `@Lock` peut être placée au niveau de la classe, auquel cas le type de contrôle s'applique à toutes les méthodes de la classe.

Exemple complet ci-après.

Exemple de singleton

```
@Singleton
@Startup
@Lock(READ)
@AccessTimeout(value=60000, unit=MILLISECONDS)
public class MonSingleton {
    private String etat;
    private byte[] donnees; // donnees
    private int compteur;    // compteur d'accès aux donnees
```

Exemple de singleton

```
MonSingleton(byte[] donnees) {    // constructeur
    this.donnees = donnees;
    this.compteur = 0;
    this.etat = "initial";
}

@Lock(WRITE)

public void setEtat(String etat) {
    this.etat = etat;
}

public String getEtat() {
    return this.etat;
}
```


Exemple de singleton

```
@Lock(WRITE)

public byte[] lireDonnees() {
    this.compteur += 1;
    return this.donnees;
}

public int getCompteur() {
    return this.compteur;
}
}
```

Les EJB Session

Définition d'un EJB Session stateless : au moyen de l'annotation
`@Stateless`

Définition d'un EJB Session stateful : au moyen de l'annotation
`@Stateful`

Les EJB Session : exercice

Réaliser l'exercice 7 (calculatrice basique)

Les EJB Session : exercice

Réaliser l'exercice 8 (calculatrice améliorée)

Bilan des exercices

Bilan de cet exercice ?

On a découvert beaucoup de choses :

- une application Web
- le choix d'un serveur d'application
- le déploiement d'une application sur le serveur
- la nécessité d'utiliser un serveur d'application embarqué
- ...

Bilan des exercices

JEE est un framework très complet, riche, et comportant de très nombreuses API.

L'intégration entre les implémentations des différents conteneurs / IDE / ORM service providers / drivers JDBC / ... n'est pas toujours parfaite, et l'investigation peut parfois être longue et fastidieuse.

Il est parfois nécessaire d'activer un ORM service provider ou un serveur d'application en mode debug... d'utiliser Maven pour investiguer les dépendances incompatibles entre elles du même composant... etc.

Amélioration de la calculatrice

Il reste néanmoins une question : comment faire pour que chaque client HTTP puisse disposer de sa propre calculatrice, et que sa mémoire soit partagée par toutes les servlets activées par ce même client HTTP ?

Amélioration de la calculatrice

Il faudrait que chaque client HTTP soit **associé** à une instance d'EJB **unique**, quelle que soit la servlet utilisée par ce client.

Est-ce qu'un singleton permettrait cela ?

QUIZZ

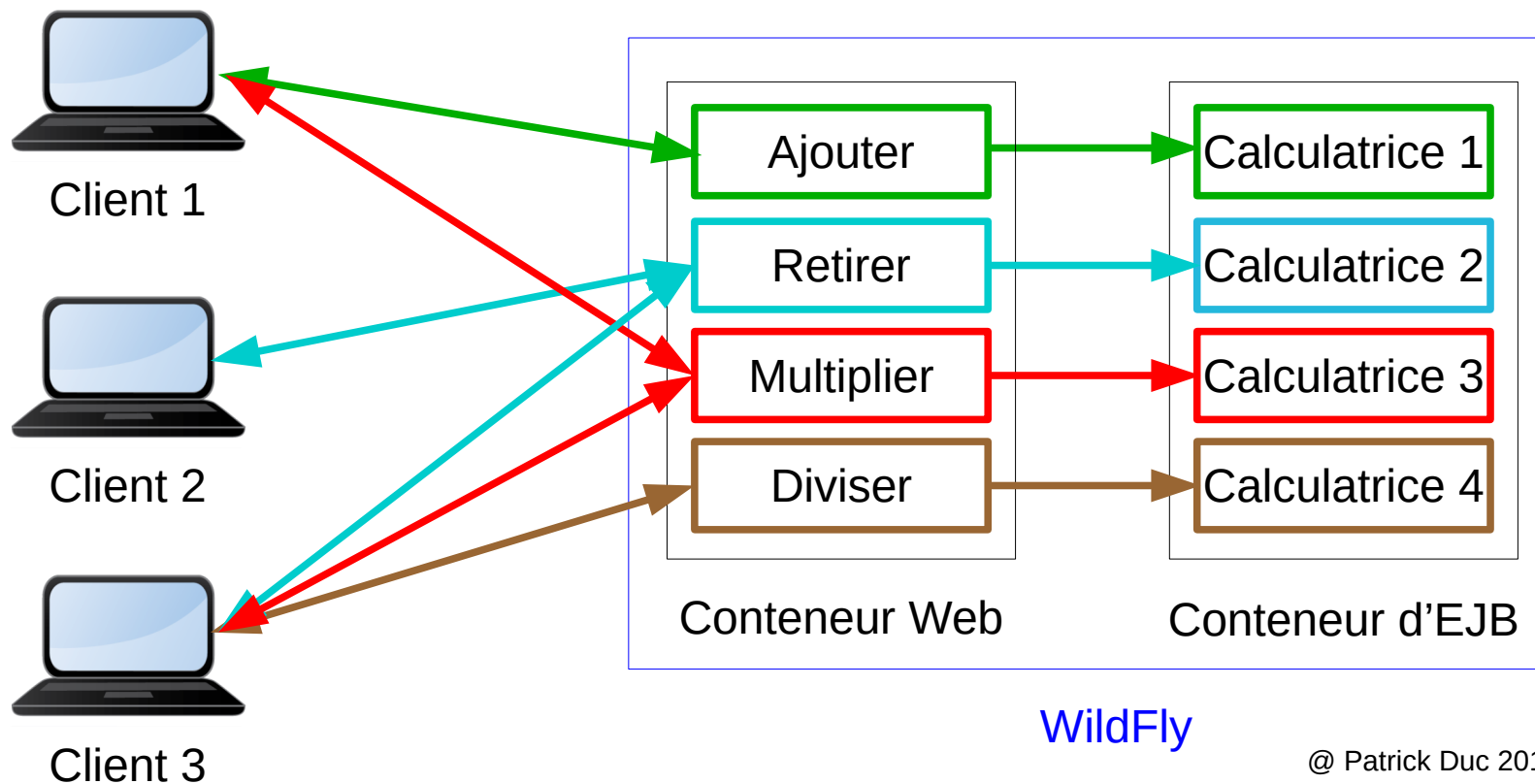
Amélioration de la calculatrice

Non : un singleton est une instance d'EJB unique pour une application JEE

⇒ tous les clients partageraient la même calculatrice

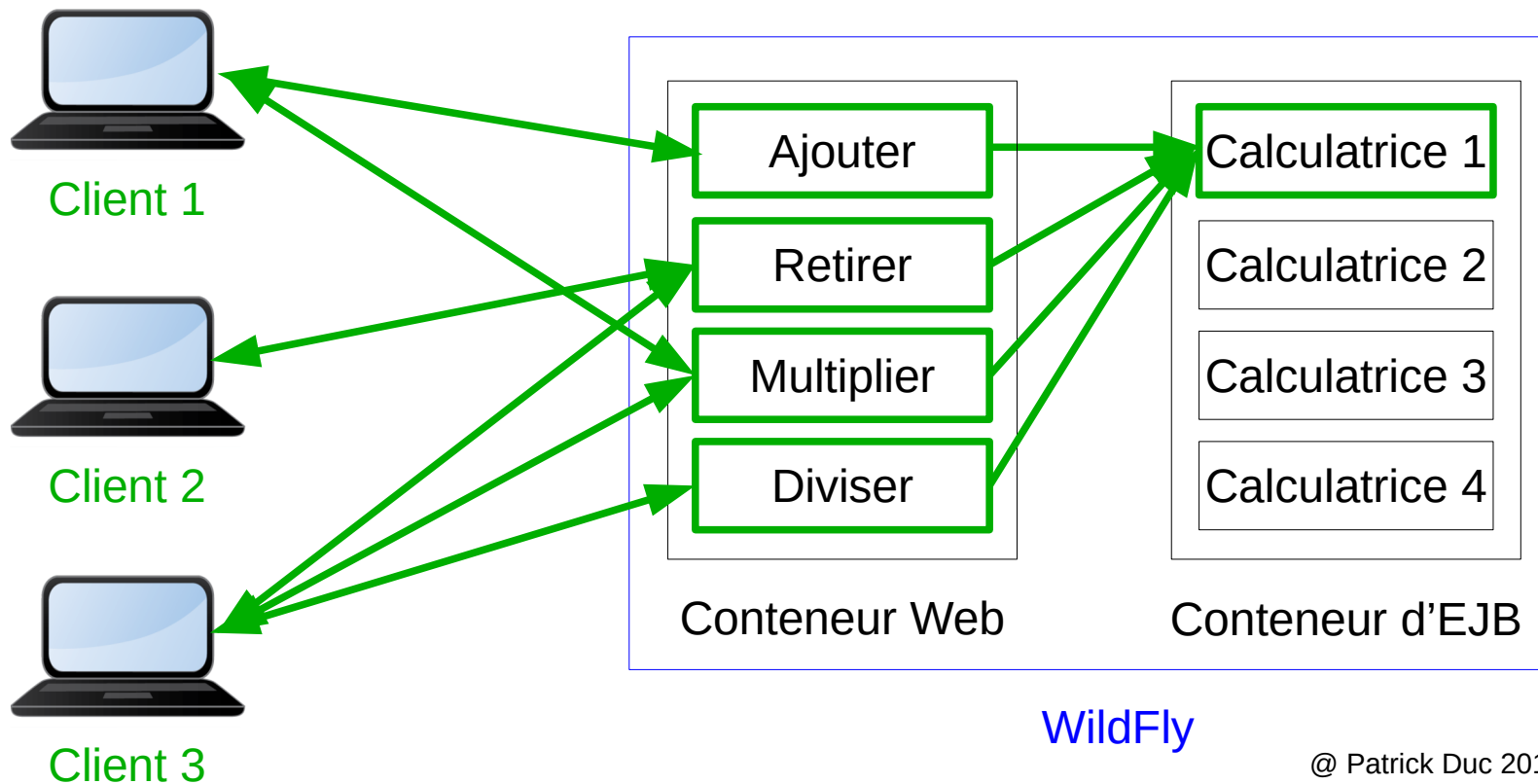
Situation en fin d'exercice 8

La situation actuelle : association servlet / EJB (car stateful)



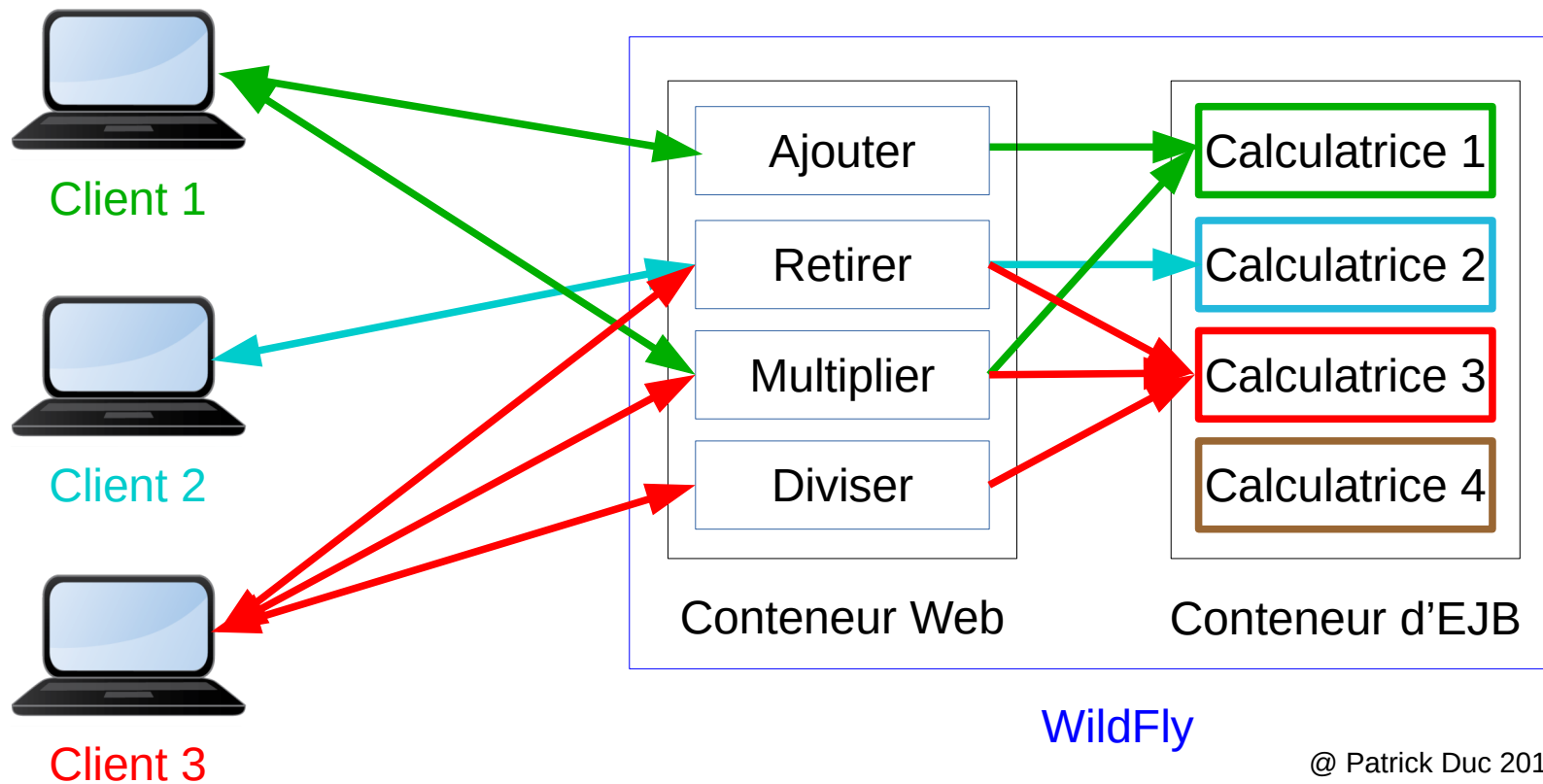
Situation si utilisation d'un EJB singleton

La situation avec un EJB Singleton : une seule calculatrice



Situation recherchée

Ce que nous voulons : association client / EJB



Session HTTP

Il faut donc que la servlet reconnaisse les clients HTTP et en fonction de cela choisisse l'EJB stateful.

Reconnaitre les clients HTTP au niveau du serveur implique de gérer une **session HTTP**.

Quelques informations à ce sujet...

Session HTTP en Java

HTTP est un protocole “sans état”, c’est-à-dire qu’une fois qu’un serveur a répondu à une requête HTTP, il ne conserve aucune information sur le client ou sa requête.

Le concept de session HTTP a été créé pour pouvoir mémoriser côté serveur HTTP des informations relatives à un client HTTP.

En Java, la création d’une telle session est faite par un composant Web. Une servlet mémorise des informations sur le client HTTP dont elle traite une requête au moyen de l’appel à la méthode `getSession()` d’un objet `HttpServletRequest`

Session HTTP en Java

Extrait du code d'une servlet

```
protected void processRequest(HttpServletRequest request,  
    HttpServletResponse response) throws ServletException,  
    IOException {  
    HttpSession laSession = request.getSession();  
  
    ...  
  
}  
  
// La session HTTP dans cette servlet n'existe pas tant  
// que cette méthode n'a pas été appelée
```

Session HTTP en Java

Un objet `HttpSession` permet entre autres :

- de récupérer l'identifiant unique de la session - cet identifiant est positionné par le serveur HTTP ayant reçu la requête HTTP, à partir d'informations de contexte :
 - couple adresse IP / port de l'émetteur de la requête
 - identifiant du « user agent » (navigateur, ligne de commande, ...)
⇒ `session.getId()`
- d'enregistrer ou récupérer un objet Java dans la session
⇒ `session.setAttribute()` ou `session.getAttribute()`

Session HTTP en Java

L'attribut enregistré dans une `HttpSession` est en fait un couple clé/valeur :

- la clé est une chaîne de caractères identifiant l'attribut
- la valeur est un objet Java

Exemple :

```
CalculatriceBean calculatrice = ...;
```

```
laSession.setAttribute("calculatrice", calculatrice);
```

Implémentation de la calculatrice

Vous avez maintenant ce qu'il faut pour implémenter une calculatrice par client :

- lors de la réception d'une requête, la servlet ouvre sa session HTTP, et essaye d'y récupérer un EJB `CalculatriceBean`
- si l'objet n'existe pas, c'est que c'est un nouveau client HTTP; dans ce cas :
 - la servlet utilise l'EJB `CalculatriceBean` qui lui a été injecté pour effectuer le calcul
 - elle enregistre ensuite cet EJB dans sa session HTTP
- si l'objet existe, alors la servlet effectue simplement le calcul.

Amélioration de la calculatrice

NB. Il est en fait inutile d'injecter l'EJB `CalculatriceBean` dans la servlet, puisqu'il est peut-être déjà disponible dans la session HTTP.

La meilleure solution consisterait à obtenir **si nécessaire** une instance de cet EJB par un appel à JNDI (Java Naming and Directory Interface, alias JNDI, le service d'annuaire Java)

⇒ on reviendra sur JNDI plus tard

Gestion des utilisateurs de Topaidi

Réaliser l'exercice 9 (gestion des utilisateurs)

Cycle de vie des EJB Session

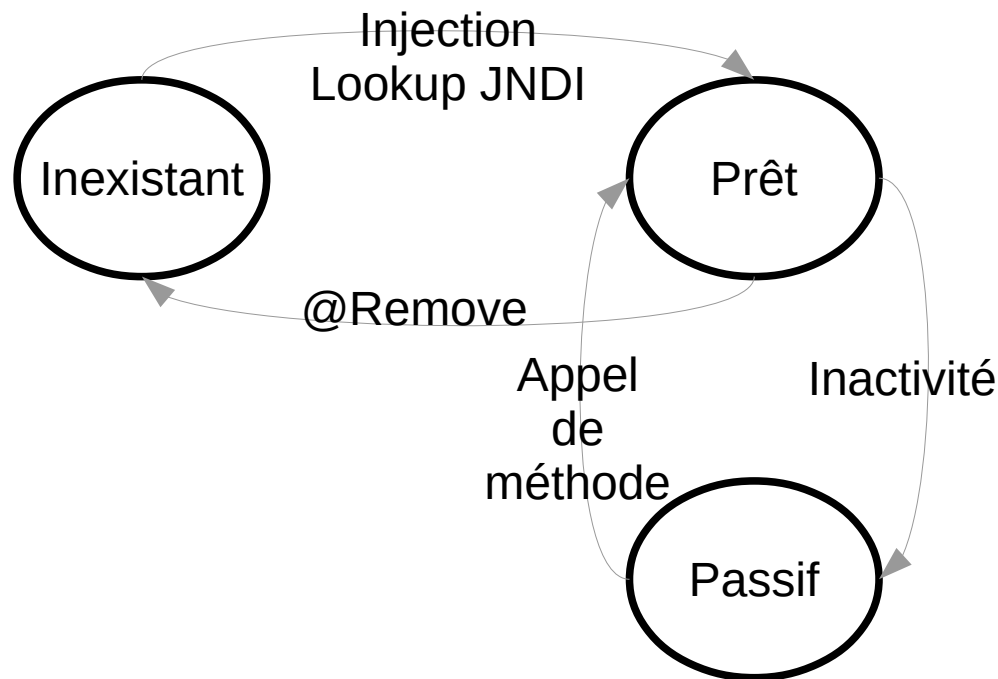
Selon leur type, les EJB Session suivent un cycle de vie :

- **contrôlé par le conteneur d'EJB** pour les EJB Session stateless et singleton
- **contrôlé par l'utilisateur** pour les EJB Session stateful.

Pourquoi cette logique ? **QUIZZ**

Cycle de vie des EJB Session stateful

Le cycle de vie des EJB Session stateful



Cycle de vie des EJB Session stateful

Le timeout avant passivation d'un EJB Session stateful peut être spécifié au moyen de l'annotation `@StatefulTimeout`.

Un EJB Session stateful peut être détruit explicitement par son utilisateur en plaçant l'annotation `@Remove` sur une ou plusieurs de ses méthodes

- à l'issue de l'exécution de la méthode, l'EJB sera supprimé
 - sauf si la méthode lève une exception, auquel cas la destruction ou non de l'EJB peut être spécifiée par l'argument booléen `retainIfException` de l'annotation

Cycle de vie des EJB Session stateful

A noter que la passivation d'un EJB Session stateful se traduit par sa sérialisation et son enregistrement dans un fichier

- si l'état conversationnel d'un tel EJB comporte des ressources non sérialisables (socket, connexion vers une base de données, etc.), alors le développeur a la possibilité de fermer ces ressources
 - ⇒ appel d'une callback marquée par une annotation `@PrePassivate`
- inversement, appel d'une méthode d'activation lorsque l'EJB sera réactivé, pour réouvrir les ressources (socket, connexion...)
 - ⇒ appel d'une callback marquée par une annotation `@PostActivate`

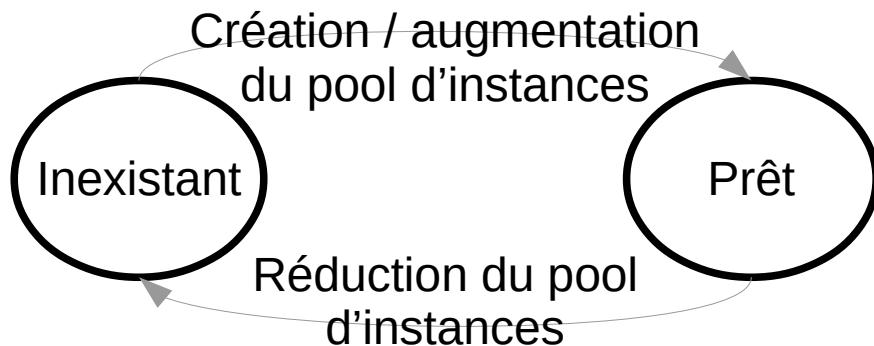
Cycle de vie des EJB Session stateful

Ces callbacks sont appelées des intercepteurs (interceptors en anglais).

Elles seront listées plus loin.

Cycle de vie des EJB Session stateless

Le cycle de vie des EJB Session stateless



Cycle de vie des EJB Session stateless

Un EJB Session stateless n'est jamais passivé, il est détruit par le conteneur d'EJB si ce dernier veut réduire son pool d'instances.

Le développeur d'un EJB Session stateless peut spécifier différents intercepteurs liés au cycle de vie de cet EJB.

Cycle de vie des EJB Session singleton

Un EJB Session singleton a le même cycle de vie que celui d'un EJB Session stateless.

De même, on peut spécifier différents intercepteurs liés au cycle de vie de cet EJB.

Les intercepteurs

Les annotations suivantes permettent de définir des **callbacks** appelées lors des changements d'état des EJB Session :

`@PostConstruct` ==> après construction d'un EJB Session

`@PreDestroy` ==> avant destruction d'un EJB Session

`@PrePassivate` ==> avant passivation d'un EJB Session
(seulement stateful)

`@PostActivate` ==> après activation d'un EJB Session
(seulement stateful), mais avant appel de la méthode "métier"
sur cet EJB

Les intercepteurs

NB. Il y a d'autres intercepteurs, par exemple ceux liés aux événements relatifs aux transactions dans lesquelles un EJB est impliqué.

Les façades d'entités JPA

On a vu dans la partie Persistance comment accéder à des objets Métier (entités JPA).

C'est de la responsabilité des utilisateurs (i.e. composants) de ces entités de gérer leur état dans la base de données :

- les enregistrer
- les modifier
- les lire
- les supprimer

Les façades d'entités JPA

Les composants utilisateurs doivent de plus obtenir un Entity Manager (par injection ou par consultation de l'annuaire).

S'il y a beaucoup d'utilisateurs, cela amène de la **duplication de code**, et cela rend l'utilisation des entités plus **complexe**.

On peut proposer un design pattern permettant d'éviter cette duplication, tout en supprimant la nécessité pour les utilisateurs d'accéder à l'Entity Manager : la **Façade Entité**.

Une **Façade** est un design pattern qui vise à simplifier une API, rendre plus simple l'utilisation d'une librairie, etc.

Les façades d'entités JPA

NetBeans supporte ce design pattern : via le wizard “New Session Beans for Entity Classes” d'un projet JEE, il suffit de préciser pour quelle entité JPA on veut créer une façade et NetBeans crée un EJB Session stateless dérivant d'une `AbstractFacade`.

Les façades d'entités JPA

`AbstractFacade` est une classe abstraite générique fournissant les services CRUD suivant sur l'entité qu'elle gère :

- `create()` : enregistrement d'une instance dans la base
- `edit()` : modification d'une instance dans la base
- `remove()` : effacement d'une instance dans la base
- `find()` : recherche d'une instance dans la base par son identifiant
- `findAll()` : récupération de la liste des instances
- `findRange()` : récupération d'un sous-ensemble d'instances
- `count()` : comptage du nombre d'instances dans la base

Les façades d'entités JPA

Il reste à implémenter la méthode abstraite `getEntityManager()` qui récupère l'Entity Manager permettant d'accéder à la base de données concernée.

Cet Entity Manager est injecté dans la classe dérivée (la façade concrète), `getEntityManager()` retourne simplement la variable correspondante.

Les façades d'entités JPA

@Stateless

```
public class IdeeEntityFacade extends AbstractFacade<IdeeEntity> {  
    @PersistenceContext(unitName = "FilRouge-ejbPU")  
  
    private EntityManager em;    // injection de l'entity manager  
  
    @Override  
  
    protected EntityManager getEntityManager() {  
        return em;  
    }  
  
    ...  
}
```

Les façades d'entités JPA

Les utilisateurs de l'entité JPA peuvent donc y accéder par des méthodes de haut niveau, et n'ont pas à spécifier l'Entity Manager gérant cette entité.

Exemple dans le cas d'une servlet page suivante.

Les façades d'entités JPA

```
@WebServlet(name = "ListUsers", urlPatterns =  
{"/ListUsers"})
```

```
public class ListUsers extends HttpServlet {  
    @EJB                // injection de la façade  
    private UserEntityFacade userEntityFacade;  
  
    protected void processRequest(HttpServletRequest  
request, HttpServletResponse response) throws  
ServletException, IOException {
```

Les façades d'entités JPA

```
...  
  
List users = userEntityFacade.findAll();  
  
for (Iterator it = users.iterator(); it.hasNext();) {  
    UserEntity elem = (UserEntity) it.next();  
  
    out.println("<b>" + elem.getEmail() + "</b><br />");  
  
    out.println(elem.getPassword() + "<br /> ");  
  
}  
  
...
```

L'accès aux entités JPA

Réaliser l'exercice 10 (façade vers une entité JPA)

Les différents types de middleware

Middleware = *couche de communication entre différentes applications distantes*

Des applications distantes sont des applications qui ne s'exécutent pas sur la même machine.

Le middleware est ce qui permet à des applications distantes de communiquer entre elles, en tant qu'**applications** – ce n'est donc pas Internet, ou TCP, ou le Wifi.

Les différents types de middleware

Il y a deux grands types de middleware :

- les middlewares orientés “appels de procédure” ou RPC (Remote Procedure Call)
- les middlewares orientés “messages” ou MOM (Message Oriented Middleware)

Appel de procédure

Invoquer une méthode d'un objet Java, ou appeler une fonction, présente les caractéristiques suivantes :

- l'appelant donne le **nom** de la fonction / méthode appelée
- il passe les **paramètres** nécessaires à la fonction / méthode, en utilisant les bons **types** et parfois les bons **identifiants**
- il est **bloqué** tant que la fonction / méthode ne se termine pas
- quand la fonction / méthode se termine, il récupère (ou pas ==> procédure) une **valeur** d'un type **précis** qu'il peut traiter
- il peut **alors** continuer son traitement.

Appel de procédure

Autrement dit, l'appelant doit savoir pas mal de choses sur la fonction / méthode qu'il appelle :

- son nom
- le nombre et le type de ses paramètres
- le type de la valeur de retour.

De plus, il ne peut rien faire tant que la fonction / méthode n'est pas terminée.

Appel de procédure

==> On parle de **couplage fort**, car :

- les **identifiants** et **types** des données doivent se correspondre **parfaitement** des deux côtés de la communication
- et les traitements sont **synchrones**.

Appel de procédure distante

Ce qui a été dit concerne l'appel à une fonction ou méthode locale, c'est-à-dire (en gros) dans le **même** programme, sur la **même** machine.

Le type de middleware dit RPC vise à permettre de faire des appels de fonction dans d'**autres programmes** s'exécutant sur d'**autres machines**.

L'idée est attirante, car cela permet de reproduire un schéma qui fonctionne bien en local.

Appel de procédure distante

Exemples de technologies qui fonctionnent sur le principe des RPC :

- les RPC s.s. (d'origine Sun, dans les années 80)
- CORBA (années 90, but = intégration de systèmes hétérogènes)
- DCOM (monde Windows)
- SOAP (technologie de web services)
- RMI (middleware Java)
- ...

Appel de procédure distante

Souci ==> le couplage fort est étendu au **réseau**, qui rend les choses plus délicates :

- est-ce que la machine sur laquelle s'exécute l'autre programme est bien allumée ?
- est-ce que l'autre programme s'exécute bien sur l'autre machine ? (problématique de déploiement des applications)
- est-ce que ce que je sais des types et identifiants à préciser est bien correct ? (problématique de gestion de version des déploiements)
- en plus, il peut se produire un problème lié purement à la communication (problème réseau).

Appel de procédure distante

L'avant-dernier point rend les évolutions des deux parties en communication potentiellement délicates, car elles peuvent impacter l'interface, rendant les deux parties **incompatibles**.

Ce schéma de communication repose généralement sur une **dissymétrie** entre l'appelant et l'appelé :

- l'appelant est un client
- l'appelé est un serveur.

En règle générale, les implémentations de ce schéma de communication ne permettent pas au serveur d'appeler le client.

Les middlewares orientés message

Il existe un autre modèle de communication, non basé sur la simulation d'appel de procédure : les échanges de message.

Dans ce modèle, les deux parties sont équivalentes : on parle de communication **peer-to-peer**, car chaque client de messagerie peut envoyer un message à n'importe quel autre client de messagerie – il n'y a plus à proprement parler de client et de serveur.

On parle d'ailleurs uniquement de composants « client » de MOM, pas de composant « serveur ».

Les middlewares orientés message

De plus, les informations à connaître sont plus limitées côté émetteur :

- la **destination** du message (qui n'est **pas** l'identifiant du destinataire)
- le **format** du message.

Côté destinataire, il n'y a rien besoin de savoir sur l'appelant, un message est simplement délivré, et il est traité.

De plus, l'opération de réception peut être effectuée bien après l'opération d'envoi : il n'y a pas plus de **synchronicité**.

Les middlewares orientés message

Ceci permet à l'émetteur d'envoyer un message et d'effectuer immédiatement d'autres traitements, sans devoir attendre que l'autre partie soit disponible pour recevoir le message.

Globalement, le couplage est donc plus lâche entre les deux parties communicantes.

Les composants logiciels qui permettent ce genre de communication forment ce qu'on appelle un MOM : **Message Oriented Middleware**.

Les middlewares orientés message

Il existe de plus deux styles de communication par message :

- le **point à point**, qui permet l'échange d'un message entre un émetteur et un destinataire au travers d'une **queue**
- le **publish / subscribe**, qui permet à un émetteur de distribuer un message sur un certain sujet (**topic**) qui sera relayé à tous les clients s'étant abonnés à ce sujet.

Les différents types de middleware

Bien entendu, la distinction entre ces deux modèles est parfois floue :

- des middleware orientés RPC peuvent parfois autoriser les traitements en partie asynchrones, dans le sens que l'appelant fait un appel RPC qui retourne immédiatement; le résultat du traitement sera disponible plus tard
- inversement, des MOM peuvent nécessiter dans certains cas de figure que la distribution du message soit synchrone de l'envoi.

Les différents types de middleware

Exemple : HTTP est un protocole qui présente certaines caractéristiques de communication RPC (dissymétrie, synchronicité à l'envoi) et certaines caractéristiques de MOM (envoi d'un message plutôt qu'appel de procédure, envoi synchrone mais traitement possiblement asynchrone).

JMS API

Java propose une **API** permettant la communication entre composants au travers d'un MOM : JMS API (Java Message Service API).

Une implémentation de MOM supportant la JMS API est appelée un **provider JMS**.

Cette API est générique et n'adresse pas un service de messagerie particulier.

La JMS API supporte le mode **point à point** et le mode **publish/subscribe**.

Concepts JMS

Composants mis en oeuvre dans une application JMS :

- **provider JMS** : implémente l'API JMS et fournit des outils d'administration de la communication
- **clients JMS** : utilisent l'API pour *produire* et *consommer* des messages
- **messages** : support de l'échange d'information entre clients JMS
- **objets administratifs** : fabriques de connexions et destinations

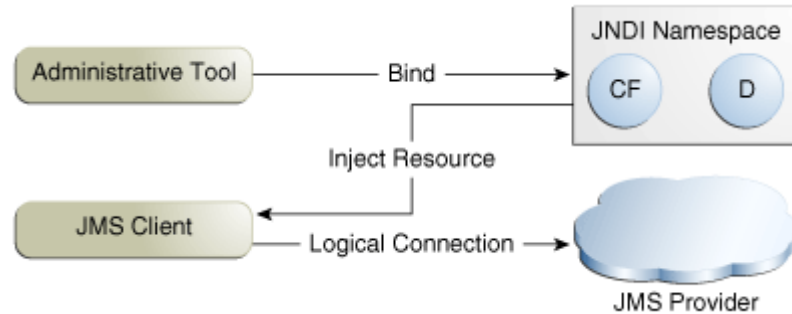
Concepts JMS

Deux informations capitales doivent être définies pour mettre en œuvre une communication JMS :

- une **fabrique de connexions** (JMS connection factory) : l'objet utilisé par un client JMS pour établir une connexion avec un provider JMS
- une **destination** : l'objet qui permet à un client JMS d'indiquer la destination des messages qu'il envoie et la source des messages qu'il reçoit.

Concepts JMS

Ces deux objets reçoivent un identifiant JNDI qui permettra de les récupérer par leur nom dans le code du client.



Définition de ressources JMS

The screenshot shows the GlassFish Server Administration Console interface. The top navigation bar includes 'Home', 'About...', 'User: admin', 'Domain: domain1', 'Server: localhost', and 'Help'. Below this, the title 'GlassFish™ Server Open Source Edition' is displayed. The left sidebar contains a tree view of the server's configuration, with 'JMS Resources' expanded to show 'Connection Factories'. The main content area is titled 'JMS Connection Factories' and includes a descriptive paragraph: 'Java Message Service (JMS) connection factories are objects that allow an application to create other JMS objects programmatically. Click New to create a new connection factory. Click the name of a connection factory to modify its properties.' Below the text is a table titled 'Connection Factories (4)' with columns for 'Select', 'JNDI Name', 'Logical JNDI Name', 'Enabled', 'Resource Type', and 'Description'. The table lists four connection factories, all of which are enabled.

JMS Connection Factories

Java Message Service (JMS) connection factories are objects that allow an application to create other JMS objects programmatically. Click New to create a new connection factory. Click the name of a connection factory to modify its properties.

Select	JNDI Name	Logical JNDI Name	Enabled	Resource Type	Description
<input type="checkbox"/>	jms/_defaultConnectionFactory	java:comp/DefaultJMSConnectionFactory	✓	javax.jms.ConnectionFactory	
<input type="checkbox"/>	jms/TopicFactory		✓	javax.jms.TopicConnectionFactory	Factory pour test de topics
<input type="checkbox"/>	jms/NewMessage2Factory		✓	javax.jms.QueueConnectionFactory	JMS Queue factory pour test
<input type="checkbox"/>	jms/NewMessageFactory		✓	javax.jms.QueueConnectionFactory	Autre JMS Queue factory pour test

Définition de ressources JMS

[Home](#) [About...](#)
User: admin | Domain: domain1 | Server: localhost
GlassFish™ Server Open Source Edition

Common Tasks

Domain

server (Admin Server)

Clusters

Standalone Instances

Nodes

Applications

Lifecycle Modules

Monitoring Data

Resources

Concurrent Resources

Connectors

JDBC

JMS Resources

Connection Factories

jms/_defaultConnectionFactory

jms/NewMessage2Factory

jms/NewMessageFactory

Destination Resources

jms/NewMessage2

jms/NewMessage

JNDI

JavaMail Sessions

Resource Adapter Configs

Configurations

JMS Destination Resources

JMS destinations serve as the repositories for messages. Click New to create a new destination resource. Click the name of a destination resource to modify

Destination Resources (3)

New...

Delete

Enable

Disable

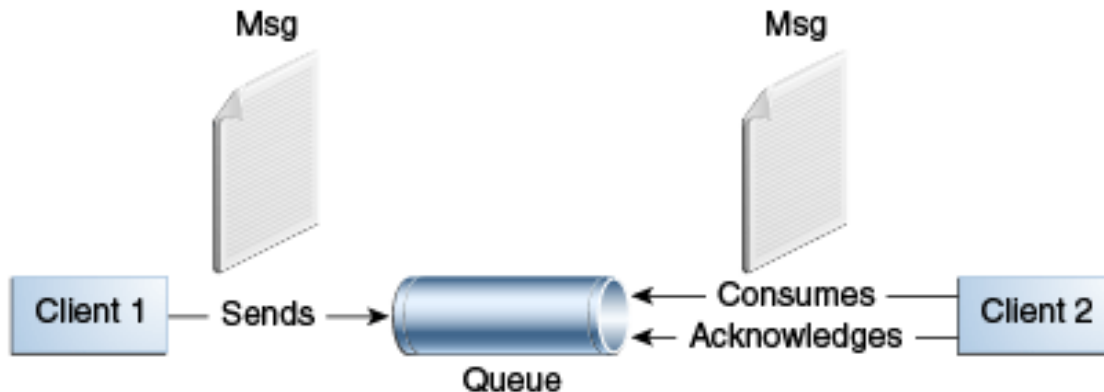
Select	JNDI Name	Enabled	Resource Type	Description
<input type="checkbox"/>	jms/NewMessage2	✓	javax.jms.Queue	
<input type="checkbox"/>	jms/NewMessage	✓	javax.jms.Queue	Queue pour les tests
<input type="checkbox"/>	jms/WeatherForecast	✓	javax.jms.Topic	Topic pour les prévisions météo

@ Patrick Duc 2018

Styles de communication par message

La communication point à point : envoi / lecture de messages sur une queue

- garantie de **livraison** du message
- garantie de lecture **unique** du message

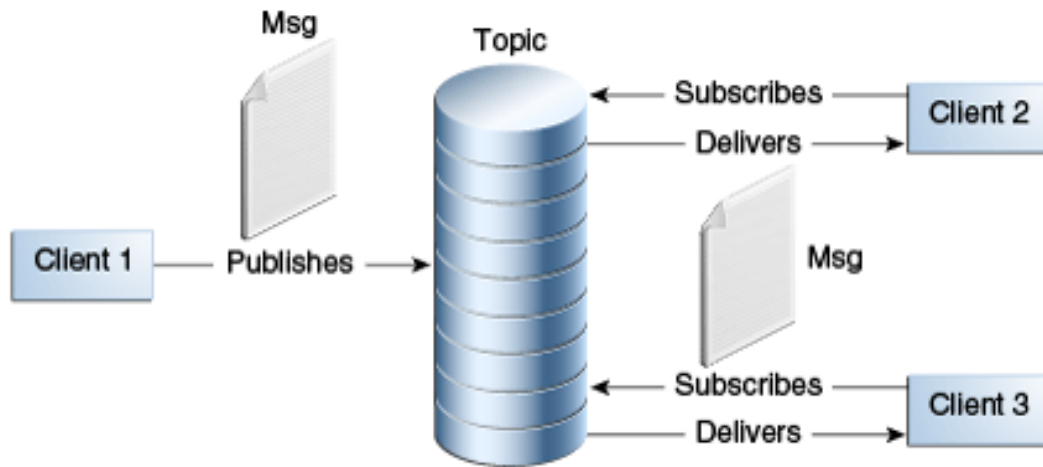


Styles de communication par message

La communication publish / subscribe : publication sur / abonnement à un sujet (**topic**)

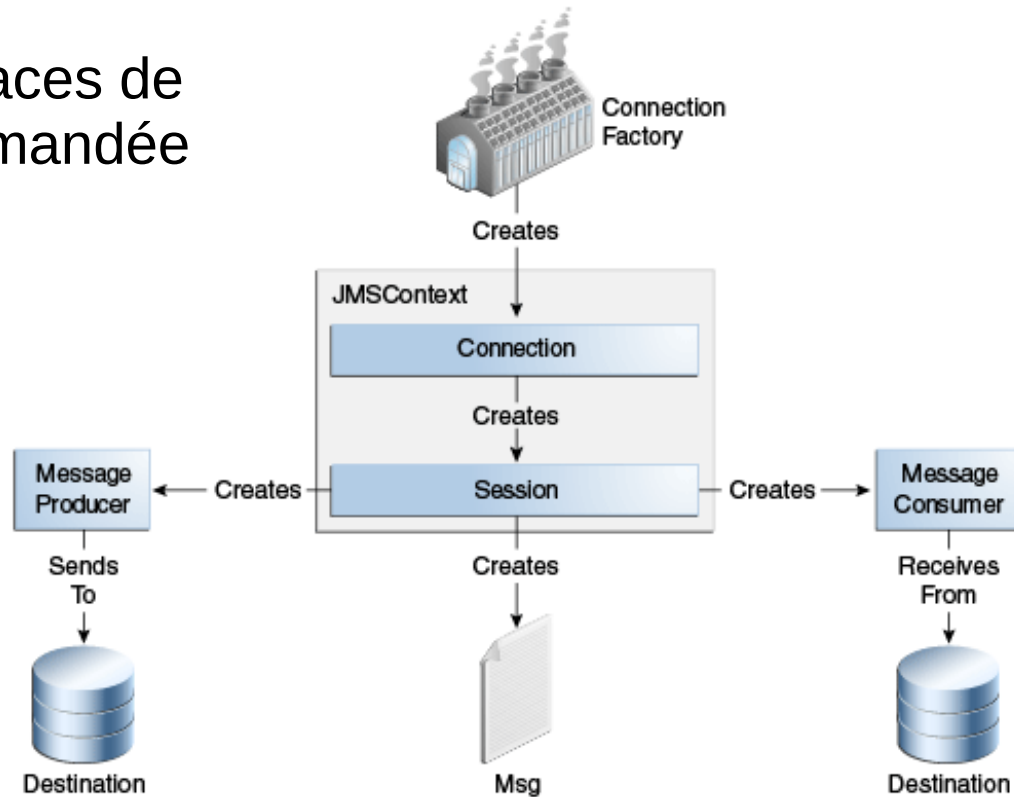
- garantie de **livraison**

du message



L'API JMS

Principales interfaces de l'API JMS recommandée



Scénario de mise en œuvre de JMS

Séquence d'opérations chez un client JMS :

- Recherche par JNDI d'une `ConnectionFactory`
- Recherche par JNDI d'un ou plusieurs objets `Destination` (`Queue` ou `Topic`)
- Utilisation de la fabrique pour créer un objet `JMSContext`
- Utilisation de l'objet `JMSContext` pour créer les objets `JMSConsumer` et `JMSProducer`
- Production et consommation de messages via ces objets

Les EJB “message-driven”

La JMS API est disponible pour les applications Java SE aussi bien que JEE.

Dans le contexte JEE, un provider JMS doit supporter un second type d'EJB : les EJB “message-driven”.

De plus, un tel provider doit permettre la mise en oeuvre des échanges de message en tant qu'éléments d'une **transaction JTA**.

Les EJB “message-driven”

Ils permettent le traitement de messages JMS a priori, mais pas seulement.

Ils ne doivent pas gérer d'état conversationnel car :

- le conteneur d'EJB peut faire traiter un message par n'importe quelle instance d'EJB MD
- un EJB MD peut traiter les messages en provenance de plusieurs émetteurs.

Les EJB “message-driven”

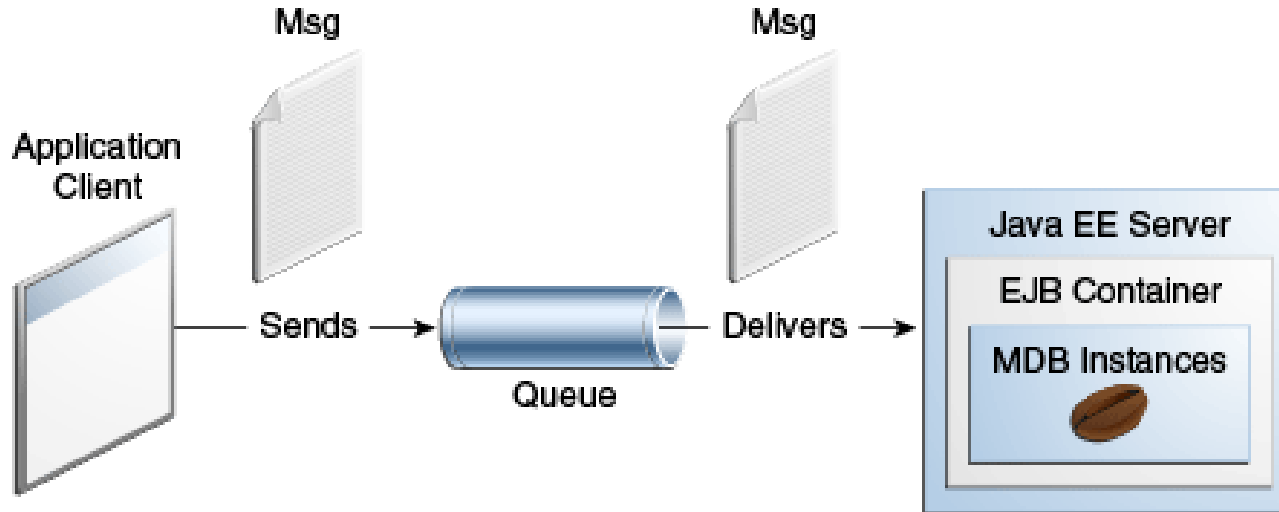
Un EJB MD est activé de manière asynchrone par rapport à l'arrivée d'un message

- à la différence des EJB Session qui peuvent consommer (ou produire d'ailleurs) des messages, mais pas de manière asynchrone.

Un EJB MD peut intervenir dans une transaction ou définir une transaction.

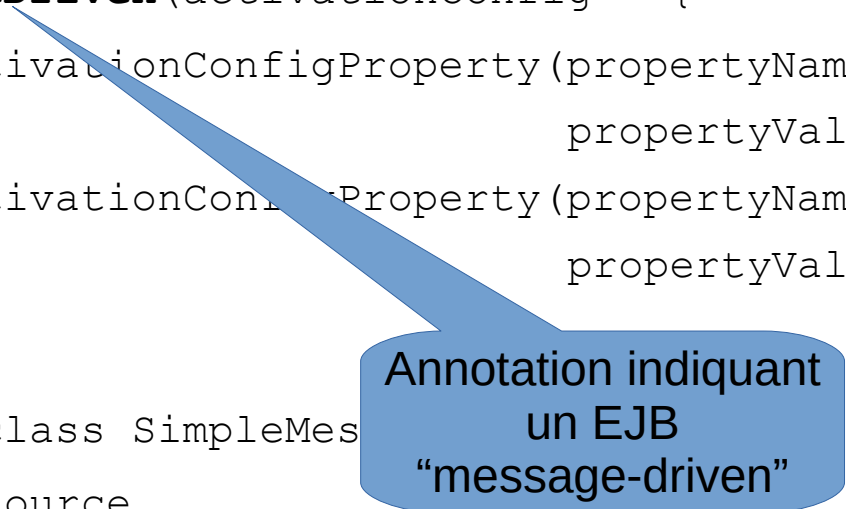
Les EJB “message-driven”

Exemple de mise en oeuvre d'EJB MD



Exemple d'EJB "message-driven"

```
@MessageDriven(activationConfig = {  
    @ActivationConfigProperty(propertyName = "destinationLookup",  
                                propertyValue = "jms/MyQueue"),  
    @ActivationConfigProperty(propertyName = "destinationType",  
                                propertyValue = "javax.jms.Queue")  
})  
  
public class SimpleMessageBean implements MessageListener {  
    @Resource  
    private MessageDrivenContext mdc;  
  
    static Logger logger = Logger.getLogger("SimpleMessageBean");  
}
```



Annotation indiquant
un EJB
"message-driven"

Exemple d'EJB "message-driven"

```
@MessageDriven(activationConfig = {  
    @ActivationConfigProperty(propertyName = "destinationLookup",  
                                propertyValue = "jms/MyQueue"),  
    @ActivationConfigProperty(propertyName = "destinationType",  
                                propertyValue = "javax.jms.Queue")  
})  
  
public class SimpleMessageBean implements MessageListener {  
    @Resource  
    private MessageDrivenContext mdc;  
  
    static Logger logger = Logger.getLogger("SimpleMessageBean");
```



Nom de l'EJB
"message-driven"

Exemple d'EJB "message-driven"

```
@MessageDriven(activationConfig = {  
    @ActivationConfigProperty(propertyName = "destinationLookup",  
                                propertyValue = "jms/MyQueue"),  
    @ActivationConfigProperty(propertyName = "destinationType",  
                                propertyValue = "javax.jms.Queue")  
})  
public class SimpleMessageBean implements MessageListener {  
    @Resource  
    private MessageDrivenContext mdc;  
  
    static Logger logger = Logger.getLogger("SimpleMessageBean");  
}
```

Informations
de configuration

Exemple d'EJB "message-driven"

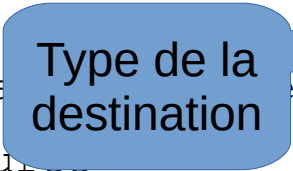
```
@MessageDriven(activationConfig = {  
    @ActivationConfigProperty(propertyName = "destinationLookup",  
                                propertyValue = "jms/MyQueue"),  
    @ActivationConfigProperty(propertyName = "destinationType",  
                                propertyValue = "javax.jms.Queue")  
})  
  
public class SimpleMessageBean implements MessageListener {  
    @Resource  
    private MessageDrivenContext mdc;  
  
    static Logger logger = Logger.getLogger("SimpleMessageBean");
```



Nom de la destination

Exemple d'EJB "message-driven"

```
@MessageDriven(activationConfig = {  
    @ActivationConfigProperty(propertyName = "destinationLookup",  
                                propertyValue = "jms/MyQueue"),  
    @ActivationConfigProperty(propertyName = "destinationType",  
                                propertyValue = "javax.jms.Queue")  
})  
  
public class SimpleMessageBean implements MessageListener {  
    @Resource  
    private MessageDrivenContext mdc;  
  
    static Logger logger = Logger.getLogger("SimpleMessageBean");
```



Type de la destination

Exemple d'EJB "message-driven"

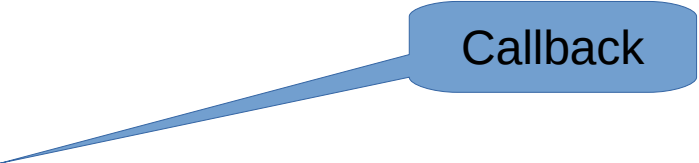
```
@MessageDriven(activationConfig = {  
    @ActivationConfigProperty(propertyName = "destinationLookup",  
                                propertyValue = "jms/MyQueue"),  
    @ActivationConfigProperty(propertyName = "destinationType",  
                                propertyValue = "javax.jms.Queue")  
})  
  
public class SimpleMessageBean implements MessageListener {  
    @Resource  
    private MessageDrivenContext mdc;  
  
    static Logger logger = Logger.getLogger("SimpleMessageBean");  
}
```



Listener de messages

Exemple d'EJB “message-driven”

```
public SimpleMessageBean() {  
    }  
  
    @Override  
    public void onMessage(Message message) {  
        if (message instanceof ObjectMessage) {  
            ObjectMessage msg = (ObjectMessage) message;  
            NewsEntity e = (NewsEntity) msg.getObject();  
            save(e);  
        }  
    }  
}
```



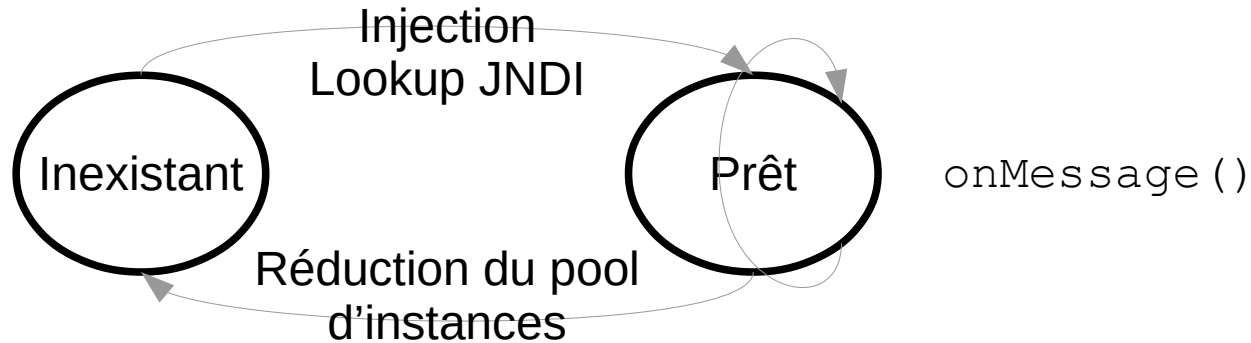
Exemple d'EJB “message-driven”

```
public SimpleMessageBean() {  
  
}  
  
@Override  
public void onMessage(Message message) {  
  
    if (message instanceof ObjectMessage) {  
        ObjectMessage msg = (ObjectMessage) message;  
        NewsEntity e = (NewsEntity) msg.getObject();  
        save(e);  
    }  
  
}
```

Logique
applicative

Cycle de vie des EJB “message-driven”

Le cycle de vie des EJB “message-driven”



Les messages JMS

Un message JMS est composé d'un **header** (entête), d'un **body** (corps) et éventuellement de **propriétés** (couples clé/valeur).

Le header comprend un grand nombre de champs dont la grande majorité sont remplis par le JMS provider.

Le corps peut être d'un type variable, selon le besoin.

Les propriétés peuvent permettre la compatibilité avec d'autres systèmes de messagerie (non-JMS) ou permettre de sélectionner les messages côté récepteur selon certaines propriétés.

L'entête d'un message JMS

Quelques champs de l'entête d'un message JMS :

- destination
- mode de livraison (persistent ou non si pb avec le provider JMS)
- heure d'envoi **plus éventuellement un délai choisi par l'expéditeur**
- priorité du message
- id du message
- type du message
- ...

Le corps d'un message JMS

Les types possibles d'un corps de message JMS :

- `TextMessage` (un message de type `String`)
- `MapMessage` (ensemble de couples clé/valeur)
- `BytesMessage` (suite d'octets sans structure)
- `StreamMessage` (suite de valeurs primitives)
- `ObjectMessage` (objet Java sérialisable)
- `Message` (pas de corps, seulement un header)
- ...

Propriétés d'un message JMS

Exemple de filtrage des messages par une **propriété**

```
@MessageDriven(activationConfig = {  
    @ActivationConfigProperty(propertyName =  
        "destinationLookup",  
        propertyValue="java:module/jms/newsTopic"),  
    @ActivationConfigProperty(propertyName =  
        "destinationType",  
        propertyValue = "javax.jms.Topic"),  
    @ActivationConfigProperty(propertyName =  
        "messageSelector",  
        propertyValue = "NewsType = 'Sports' OR NewsType =  
        'Opinion'"),  
})
```

Exemples de code

Exemple d'accès à une ressource `ConnectionFactory` par injection en utilisant l'annuaire JNDI :

```
@Resource(lookup = "jms/MyMessageFactory")  
private static ConnectionFactory connectionFactory;
```

Exemples de code

Exemple d'accès à une ressource `Queue` par injection en utilisant l'identifiant JNDI d'une destination :

```
@Resource(lookup = "jms/MyQueue")  
private static Queue queue;
```

On notera qu'il s'agit dans cet exemple d'une communication en point à point.

Exemples de code

Exemple d'accès à une ressource `Topic` par injection en utilisant l'identifiant JNDI d'une destination :

```
@Resource(lookup = "jms/MyTopic")  
private static Topic topic;
```

Il s'agit ici d'une communication en publish / subscribe.

Exemples de code

Exemple d'écriture d'un message texte simple dans une queue :

```
try (JMSContext context=connectionFactory.createContext())
{
    context.createProducer().send(queue, "Ceci est un
message.");
}
catch (Exception ex) {
    System.out.println("Exception reçue : " + ex);
}
```

Exemples de code

Exemple de lecture d'un message texte simple dans une queue :

```
try {
    if (message instanceof TextMessage) {
        System.out.println("Reading message: " +
message.getBody(String.class));
    } else {
        System.out.println("Message is not a TextMessage");
    }
} catch (JMSEException | JMSRuntimeException e) {
    System.err.println("Exception in onMessage(): " + e);
}
```

Exemples de code

Exemple d'écriture d'un message contenant un objet dans une queue :

```
JMSContext context = connectionFactory.createContext();  
ObjectMessage message = context.createObjectMessage();  
SomeEntity someEntity = new SomeEntity(x, y, z);  
message.setObject(someEntity);  
context.createProducer().send(queue, message);
```


Exemples de code

Exemple de lecture d'un message contenant un objet dans une queue :

```
try {  
    if (message instanceof ObjectMessage) {  
        msg = (ObjectMessage) message;  
        SomeEntity e = (SomeEntity) msg.getObject();  
    }  
} catch (JMSException | JMSRuntimeException e) {  
    System.err.println("Exception in onMessage(): " + e);  
}
```

Les EJB “message-driven” : exercice

Réaliser l'exercice 11 (communication en mode point à point par message simple)

Les EJB “message-driven” : exercice

Réaliser l'exercice 12 (communication en mode point à point par envoi d'objet Java)

Les EJB “message-driven” : exercice

Réaliser l'exercice 13 (communication en mode publish / subscribe par message simple)

Quizz !

Supposons que l'on me demande de réaliser une application Web permettant d'effectuer une conversion d'unités de mesure anglo-saxonnes vers le système métrique :

- *poppyseed* (graine de pavot)
- *barleycorn* (grain d'orge)
- *nail* (clou)
- etc.

Quel type d'EJB vais-je choisir a priori pour réaliser cette application ?

Quizz !

Mon client (MultiMegaPlex) souhaite obtenir une application sur PC permettant à ses clients de parcourir la liste des films proposés dans ses salles de cinéma et de réserver des places dans ses salles.

Quels types d'EJB vais-je choisir a priori pour réaliser ces fonctions ?

Quizz !

Vous êtes le fabricant des fameux processeurs SaBeugToulTan. Un de vos clients, un fabricant d'ordinateurs veut (enfin!) informatiser sa chaîne de production, en particulier pour pouvoir travailler en flux tendu. Il prévoit donc de vous envoyer des commandes H24, en fonction de ses commandes propres.

Vous le rencontrez pour discuter de l'interface entre vos deux systèmes.

Que pourriez-vous lui proposer comme organisation pour que vos systèmes ne soient pas trop dépendants l'un de l'autre, en terme d'échange de commandes ?

Autres informations

Les interfaces d'EJB Session

Habituellement, les EJB Session sont accédés au travers d'une **interface**.

Cette interface peut permettre d'accéder à un EJB **local** ou **distant** :

- annotation `@Local` pour spécifier une interface locale
- annotation `@Remote` pour spécifier une interface distante

Les interfaces d'EJB Session

L'accès à un EJB Session au travers d'une interface distante se fait via RMI (Remote Method Interface).

Les interfaces d'EJB Session

Dans les exercices, on n'a jamais utilisé d'interface pour accéder aux EJB → on parle d'EJB « no-interface ».

Quizz (niveau très avancé)

Un EJB « message-driven » est obligatoirement du type « no-interface ».

Pourquoi ?

Utilisation de JMS API

La JMS API peut être mise en œuvre en dehors du context JEE :

- création de producteurs
- création de consommateurs (obligatoirement synchrones)

Communication par messages hors JMS

Le but initial de JMS était de fournir un accès générique en Java à des MOM existants.

Il est possible de lire et/ou écrire des messages à destination de et/ou reçus par des MOM indépendants de la JVM

- communication de systèmes hétérogènes

Packaging des applications JEE

Côté serveur, une application JEE complète contient :

- un ou plusieurs modules Web (servlets, JSP, JSF)
- un ou plusieurs modules Métier (EJB Session, EJB MD, entités JPA).

Ces parties sont packagées (groupées) dans un fichier **.ear** qui est destiné à être déployé sur le serveur d'application.

Packaging des applications JEE

Si l'application ne contient que des modules Web, alors ils sont packagés dans un fichier **.war** qui doit bien sûr lui aussi être déployé sur le serveur d'application.

NB. Un simple conteneur de servlets peut suffire alors

- Tomcat
- Jetty
- ...

MERCI !

& SUIVEZ-NOUS !



HUMAN **booster**
•• VOTRE SOLUTION COMPÉTENCE

04 73 24 93 11

contact@humanbooster.com

www.humanbooster.com