Introduction aux Session Beans

Michel Buffa (buffa@unice.fr), UNSA 2002
modifié par Richard Grin (version 1.1, 21/11/11)

Plan du cours

- Introduction générale
- EJB : Les fondamentaux
- Introduction aux Session Beans
- Introduction aux Entity Beans
- Introduction aux Message-Driven Beans
- Concepts avancés sur la persistance
- Relations avec les Entity Beans
- Gestion des transactions

Session Bean : rappel

- Un Session Bean représente
  - une action, un verbe,
  - une logique métier, un algorithme,
  - Un enchaînement de tâches…
- Exemples
  - Saisie d'une commande,
  - Compression vidéo,
  - Gestion d'un caddy, d'un catalogue de produits,
  - Transactions bancaires…

3 types de beans sessions

- **Sans état** : traite les tâches qui peuvent être accomplies en un seul appel de méthode ; pas d'état maintenu entre 2 appels de méthode
- Exemple : afficher la liste des comptes bancaires d'un client
- **Avec état** : est associé à un seul client ; maintient un état entre plusieurs appels de méthodes ; pour les tâches accomplies en plusieurs étapes
- Exemple : remplir son caddy avec des articles dont les caractéristiques sont affichées sur des pages différentes

3 types de beans sessions

- **Singleton** : quand on veut être assuré qu'il n'y a qu'une seule instance du bean pour tous les utilisateurs de l'application
- Exemple : cache d'une liste de pays (pour améliorer les performances)

Stateless Session Beans

- Le client passe toutes les données nécessaires au traitement lors de l'appel de méthode
- Le container est responsable de la création et de la destruction du Bean
  - Il peut le détruire *juste après un appel de méthode*, ou le garder en mémoire pendant un certain temps pour *réutilisation*.
- Une instance de Stateless Session Bean n'est pas propre à un client donné, elle peut être partagée entre chaque appel de méthode
- Le serveur maintient un pool de beans sans état

Exemple de bean sans état

```
package fr.unice.ejb.conversion;

import java.math.BigDecimal;
import javax.ejb.*;

@Stateless
public class ConvertisseurBean {

  private BigDecimal tauxEuro =
    new BigDecimal("0.0093016");

  public BigDecimal yenVersEuro(BigDecimal yen) {
    BigDecimal val = yen.multiply(tauxEuro);
    return val.setScale(2, BigDecimal.ROUND_UP);
  }

}
```

Interface locale

- Si on ne fournit pas d'interface séparé à ConverterBean, il aura automatiquement l'interface composée de sa méthode publique et il ne pourra être utilisé que par les composants qui sont dans le même serveur d'application
- Ce qui correspondra au code du transparent suivant

Interface locale et implémentation

```
@Local
public interface Convertisseur {
  public BigDecimal yenVersEuro(BigDecimal yen);
}
```

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

- Donner une interface locale explicite permet de ne pas exposer aux clients toutes les méthodes publiques du bean

Interface distante et implémentation

- Pour que le bean session soit utilisable par un composant situé sur un serveur d'application, il faut lui ajouter une interface distante :

```
@Remote
public interface ConvertisseurDistant {
  public BigDecimal yenVersEuro(BigDecimal yen);
}
```

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

Stateful Session Beans

- Certaines conversations se déroulent sous forment de *requêtes* successives.
  - Exemple : un client surfe sur un site de e-commerce, sélectionne des produits, remplit son caddy…
- D'une requête HTTP à l'autre, il faut un moyen de conserver un *état* (le caddy par ex.)

Stateful Session Beans

- En résumé, un Stateful Session Bean est utile pour maintenir un état pendant la durée de vie du client
  - au cours d'appels de méthodes successifs.
  - Au cours de *transactions* successives.
  - Si un appel de méthode change l'état du Bean, lors d'un autre appel de méthode l'état sera disponible.
- Conséquence : une instance de Stateful Session Bean par client.

Exemple de bean avec état

```
@Stateful
@StatefulTimeout(300000) // 5 minutes
public class CaddyEJB {
  private List<Item> caddy =
    new ArrayList<Item>();

  public void addItem(Item item) {
    ...
  }

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

Problème de ressource

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

Passivation / Activation

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

Activation/Passivation callbacks

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

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

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

Accès concurrents

- Le code des beans session n'a pas besoin d'être thread-safe puisque le container ne permettra jamais l'accès par plusieurs requêtes
- Ca n'est pas nécessairement le cas pour les EJB singletons (configurable) ; on peut laisser le container gérer la concurrence en donnant des annotations (@Lock ou @ConcurrencyManagement)