Travaux pratiques UE Infrastructures Logicielles - UCE Architectures logicielles

Utilisation de JMS

Configuration de l'environnement de travail

On va utiliser OpenMQ, l'implémentation de JMS proposée avec le serveur JavaEE Glassfish.

A – Si votre PATH ne contient pas le répertoire des exécutables de Glassfish, ajouter son chemin à la variable d'environnement PATH : dans le fichier .profile de votre répertoire de connexion

```
export PATH=$PATH:/vbox/archives/glassfish3/glassfish/bin
```

Configurer aussi le classpath en ajoutant

```
export CLASSPATH=$CLASSPATH:/vbox/archives/glassfish3/glassfish/lib/appserv-rt.jar
```

B – Génération d'un domaine Glassfish dans votre propre espace de travail

a – Dans votre espace de travail, créer un répertoire nommé domains

b – Création d'un domaine (une seule fois !)

asadmin create-domain --domaindir *chemin_absolu_du_repertoire_domains* domain1

Remarque : en *italique* les noms que vous choisissez ...

(si possible choisissez pas de mot de passe ...)

c – Pour démarrer Glassfish dans ce domaine

asadmin start-domain --domaindir *chemin_absolu_du_repertoire_domains*

d – Pour arrêter Glassfish dans ce domaine

asadmin stop-domain --domaindir *chemin_absolu_du_repertoire_domains*

Première partie : Point à Point

Pour cette partie, on va utiliser une (unique) queue de messages.

Pour cela configurez deux ressources JMS dans Glassfish :

- tout d'abord une ConnectionFactory, ressource de type QueueConnectionFactory, que vous nommerez de la forme jms/*NomChoisiParVousPourConnectionFactory*
- ensuite une Destination, ressource de type Queue, dont le nom JNDI sera de la forme jms/*NomChoisiParVousPourDestination* et dont le nom de la destination physique sera celui choisi pour la précédente ressource *NomChoisiParVousPourConnectionFactory* (sans jms !).

Travail à réaliser

Q1 – Dans un premier répertoire, écrire une classe Emetteur qui enverra un message texte : il faut retrouver via JNDI la ConnectionFactory, puis la queue Destination créées sous Glassfish,

créer et démarrer une connexion puis une session, créer un Sender puis un message, et l'envoyer. (Revoir le support de cours ...). La classe doit se terminer par un exit pour ne pas rester bloquer dans le thread.
Dans le même répertoire, écrire la classe Recepteur qui ne reçoit qu'un seul message.
La compilation peut se faire en ligne de commande : javac *NomClasse*.java
Et l'exécution aussi : java *NomClasse*

Q2 – Copiez le premier répertoire dans un second et transformez l'émetteur de la question précédente pour qu'il envoie 5 messages successifs correspondant à une chaîne passée en paramètre. On lancera une fois l'émetteur et plusieurs fois le récepteur.
Utilisez l'utilitaire QBrowser pour voir l 'état de la queue de messages, et la manipuler.

Q3 – On cherche à constater que les messages sont bien distribués par le MOM à des récepteurs différents, et que celui-ci scrute à la recherche du premier récepteur disponible (c'est-à-dire en attente de réception). Pour cela, transformez le récepteur de façon à ce qu'il accepte un premier paramètre qui représentera un nombre de millisecondes, et un second qui représentera le nom du récepteur destinataire. Après réception d'un message, le récepteur affiche le nom du destinataire et le message reçu, et ensuite attend un certain temps (les millisecondes du paramètre). Cette attente permettra à un récepteur lancé en parallèle de recevoir le(s) message(s) suivant(s) de la file d'attente.

Q4 – On transforme maintenant l'émetteur de façon à ce qu'il envoie à un destinataire désigné. Pour cela on modifie l'émetteur pour qu'il n'envoie qu'un seul message (premier paramètre) à un destinataire identifié : on définit une propriété *destinataire* du message (utilisation de setStringProperty). Dans le récepteur on placera un sélecteur qui permettra de ne recevoir que les messages pour ce récepteur destinataire (second paramètre de createReceiver). Le récepteur prendra (ou pas) en paramètre le nom du destinataire, ce qui permettra (ou pas) de filtrer la réception des messages. Un récepteur qui n'a pas été nommé (pas de paramètre) accepte tous les messages (pas de filtre). Attention, dans un sélecteur, les valeurs de type chaîne de caractères sont entre apostrophes (') et non pas entre guillemets (").
Transformez ensuite l'émetteur pour qu'il accepte en paramètre un destinataire ou pas, et le récepteur pour qu'il prenne en paramètre 0 ou plusieurs noms de destinataires.
Vous constaterez qu'un destinataire qui a placé un filtre reçoit prioritairement (par rapport à un récepteur non nommé) les messages qui lui sont destinés.

Q5 – On veut constater maintenant que les messages ne sont supprimés de la queue qu'après réception d'un accusé de réception de la part du destinataire. Pour cela, vous utiliserez QBrowser pour visualiser quand le message est réellement détruit dans la file d'attente.
On transforme ensuite le second paramètre de createQueueSession en CLIENT_ACKNOWLEDGE, de façon à ce que le récepteur envoie explicitement l'accusé de réception (utilisation de la méthode acknowledge sur le message reçu). Constatez à nouveau avec QBrowser.

Q6 – L'accusé de réception précédent est uniquement à destination du MOM.
On va maintenant faire en sorte que le récepteur renvoie un message à l'émetteur pour lui indiquer qu'il a bien reçu son message. Pour cela, l'émetteur crée une queue de réception temporaire (createTemporaryQueue), il précise dans son message à qui doit parvenir la réponse (setJMSReplyTo), puis se met en attente du message de réponse. Le récepteur crée aussi un émetteur de message (QueueSender) sans l'affecter à une queue de messages.
De plus, pour qu'il n'y ait pas de confusion entre 2 messages émis par le même émetteur, celui-ci peut affecter à son message un numéro (setJMSCorrelationID) que le récepteur peut éventuellement utilisé dans son message de réponse. Vous transmettrez ce numéro en paramètre à l'émetteur à chaque exécution.

Deuxième partie : Entreprise Beans

Dans cette partie on va construire un squelette d'application Web qui ira de l'envoi du message jusqu'au lancement d'un traitement associé à ce message, en utilisant des EJB.
Q7 – Transformez le récepteur de la question Q1 en un Message Driven Bean.
Q8 – Faites en sorte que le précédent MDB soit maintenant un destinataire nommé de façon à ce qu'il n'intercepte que les messages qui lui sont destinés. Utilisez pour cela l'émetteur de la question 4b. Vous pouvez constater que le message n'est plus visible avec QBrowser puisqu'il est tout de suite consommé. Essayez de désactiver (disable) le MDB sous Glassfish, le message sera maintenu dans la queue de messages tant que vous ne réactivez pas le MDB.
Q9 – Ecrivez maintenant le client sous forme de servlet : pour cela vous proposez un formulaire HTML (index.html) qui permet de saisir un message, et qui enchaîne sur une classe servlet ServletEmetteur qui envoie le message. Attention de ne plus utiliser System.exit dans votre code (ce qui aura pour effet de fermer Glassfish ...). Après l'envoi, ServletEmetteur affiche un message qui indique que le message a été envoyé.
Packagez le tout dans un war qui contient les 2 classes servlet et MDB dans le répertoire WEB-INF/classes.

Pour information le web.xml commence par

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance version="2.5"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
```

Q10 – Ecrivez maintenant un bean stateless TraiteMessage qui propose une méthode traitement. L'invocation de cette méthode correspondra au lancement d'un traitement associé à l'envoi du message. Dans notre cas, vous vous contenterez d'afficher "Le traitement du message ... est en cours" . Dans le MDB, utilisez une instance de TraiteMessage pour lancer le traitement dans la méthode onMessage.

Packagez le tout dans un war qui contient toutes les classes (servlet et EJB) dans le répertoire WEB-INF/classes.

Troisième partie : Publish-Suscribe

Pour cette partie, on va utiliser un (unique) topic.

Dans Qbrowser, pour visualiser les messages dans ce topic, il faudra y souscrire (susbribe).

Q11 – Ecrivez une classe Emetteur qui envoie 5 messages dans un topic. Ecrivez ensuite une classe Récepteur qui fonctionne en mode synchrone.

Constatez dans plusieurs cas de figure, éventuellement en lançant QBrowser : l'émetteur émet sans qu'aucun récepteur soit actif ; un ou plusieurs récepteurs sont lancés avant que l'émetteur ait émis.

Q12 – On va maintenant écrire un récepteur asynchrone : on suppose qu'il effectue un travail et il peut être interrompu pour consommer un message (Faites un sleep de 5 secondes pour simuler le travail qu'il effectue). Constatez encore dans différents cas de figure.

Q13 – Transformez l'émetteur de façon à ce que après avoir envoyer 5 messages, il envoie un message prioritaire. Constatez la réception dans l'émetteur.

Q14 – On va maintenant transformer l'émetteur de la question 12 pour qu'il devienne désormais durable, c'est-à-dire que, dans le cas où si il n'est pas actif, les messages lui sont conservés jusqu'à ce qu'il puisse les consommer (c'est-à-dire la prochaine fois qu'il sera activé).

Pour cela, commencez par écrire une classe Inscription qui enregistre un récepteur auprès d'un topic : elle prend en paramètre le nom du souscripteur. Transformez ensuite la classe Recepteur de façon à ce qu'elle prenne 0 ou 1 paramètre, le nom du souscripteur. Ajoutez ensuite une

classe Desinscription qui permet de désinscrire un souscripteur dont le nom est transmis en paramètre.
Avec QBrowser) : on peut aussi voir en détail le nombre de messages à distribuer à chaque souscripteur (click droit sur le topic +détails).

Q15 – Transformez maintenant la classe Emetteur de façon à ce qu'elle émette un message qui expirera au bout d'un certain nombre de secondes si il n'a pas été consommé.

Quatrième partie : Service Web et client C#

Q16 – On va reprendre les classes de la question Q10, et ajouter une classe WSEmetteur qui correspond à un service Web qui permet d'envoyer un message. Cette classe reprend en partie ce que fait la servlet : à la place de doPost, elle définit une méthode emettre qui renvoie une chaîne de caractères indiquant que le message a été envoyé (à la place de l'écrire de HTML sur le flux de sortie de la servlet).

Pour déployer le service Web :

- on conserve la structure des répertoires existantes, à savoir WEB-INF qui contient classes
- on compile la classe WSEmetteur et on place le .class dans WEB-INF/classes

```
javac -d WEB-INF/classes *.java
```

- on utilise l'utilitaire wsgen pour générer ensuite les fichiers qui permettront la communication avec le service Web qui sera déployé

```
wsgen -cp WEB-INF/classes:$CLASSPATH -d WEB-INF/classes -wsdl WSEmetteur
```

- on crée un war que l'on déploie, par exemple WSJMS.war

```
jar cvf WSJMS.war WEB-INF/* (avec éventuellement index.html)
```

- pour consulter le WSDL

```
localhost:8080/WSJMS/WSEmetteurService?WSDL
```

- pour tester le service

```
localhost:8080/WSJMS/WSEmetteurService?Tester
```

Q17 – Ecrire maintenant un client en C# : la classe déclare un objet de la classe du service Web WSEmetteurService et utilise la méthode emettre de cet objet pour envoyer un message.

Pour pouvoir communiquer avec le service Web, du côté du client on génère les classes avec l'outil wsdl du framework .NET:

```
wsdl http://NomMachineServeur:NumPortHTTP/WSJMS/WSEmetteurService?WSDL
```

Pour compiler le client

```
gmcs ClientWS.cs WSEmetteurService.cs -r:System.Web.Services
```