Master 2 - BDIA

BDED - Bases de Données et Environnements Distribués

Travaux Pratiques

Éric Leclercq, Annabelle Gillet
Annabelle.Gillet@depinfo.u-bourgogne.fr

Révision : 15 septembre 2020

Résumé

Ce document contient l'ensemble des exercices de TP du module de Bases de Données et Environnements Distribués pour la partie concernant les intergiciels (*middleware*) et les serveurs d'applications JEE. Tous les exercices sont normalement à réaliser et sont regroupés sous formes de chapitres s'étalant le plus souvent sur plusieurs séances de 2h. Les exercices qui illustrent les concepts du cours sont notés avec une étoile (*). Dans ces exercices les questions techniques et les questions plus générales qui leur sont associées sont essentielles pour la maîtrise des concepts.

Les exercices 2 et 3, ainsi que 18 et 19 ou bien 18 et 15 sont à rédiger et à rendre par binôme, ils constitueront la note de travaux pratiques pour le module BDED.

Table des matières

1 Java RMI

Notes :

Plusieurs versions de J2SE (jdk) sont installées sur les serveurs et sur l'ensemble des stations Linux Debian du département IEM. Les installations de la version officielle (SUN ou ORACLE) résident dans les répertoires `/usr/local` des stations de travail ou des serveurs. Afin de pouvoir utiliser la version choisie, vous devez positionner les variables d'environnement `PATH` et `JAVA_HOME` avec la commande `export`. Ces variables ne sont actives que dans le shell courant. Il n'est pas conseillé de les mettre dans votre `.bashrc` car vous pouvez avoir besoin de les changer assez souvent. Le système d'exploitation GNU/Linux fournit une version libre du jdk (openjdk) qui peut avoir un comportement légèrement différent de celui d'ORACLE/SUN.

Pour fixer les variables utiliser les commandes shell suivantes, en prenant garde à regarder (via un `ls -al /usr/local`) le nom réel des répertoires sous `/usr/local` :

```
export PATH=/usr/local/jdk-X.Y/bin:$PATH
export JAVA_HOME=/usr/local/jdk-X.Y
```

Pour compiler un code source java dans une version spécifique du JDK, deux options du compilateurs sont proposées : `-target` génère le byte-code dans la version spécifiée de Java, `-source` permet de vérifier la compatibilité de version du code source.

```
javac -target 1.4 -source 1.4 serveurImpl.java
```

Nom local	OS/Architecture	JDK
butor.iem	Solaris 10 x64	JDK 1.6
eluard.iem	Solaris 10 T1 Sparc	JDK 1.5
MI10X et Linux	Debian 9	JDK 1.4, 1.5, 1.6, 1.7, 1.8
aragon	Debian 9	JDK 1.4, 1.5, 1.6, 1.7, 1.8

Afin d'observer et de comprendre précisément le comportement de Java RMI, ne pas utiliser d'environnement de développement mais un simple éditeur comme vi, xemacs, nedit ou bluefish et une exécution en ligne de commande.

Le service `rmiregistry` écoute par défaut sur le port 1099. Si vous partagez une machine avec d'autres binômes, utilisez un autre port ou bien veillez à ne pas donner le même nom à vos objets distribués. Pour lancer le service avec un autre port `rmiregistry [port]`.

La machine `eluard` Solaris 10 Sparc T1, héberge le SGBD Oracle en version 11r2. Vos comptes sous Oracle sont login = login machine, mot de passe = login machine. Avant de lancer `sqlplus`, exécuter le script `/export/home/oracle/oraenv.sh` pour fixer les variables d'environnement et notamment `ORACLE_SID=ENSE2020`. Pour JDBC, l'URL de connexion depuis les salles de TP est `eluard:1521:ENSE2020`. Pour la connexion depuis l'extérieur, il est nécessaire d'utiliser le VPN du département IEM, de ce fait la chaîne de connexion reste inchangée et vos programmes peuvent s'exécuter directement. Cependant faire attention à utiliser les mêmes version de JDK.

Exercice 1. Prise en main de Java-RMI (*)

Le but de ce premier exercice (application directe du cours) est de prendre en main l'environnement Java-RMI et de se familiariser avec le processus de développement d'applications objets distribués (ici uniquement la compilation et l'exécution). Réaliser le programme client serveur `HelloWolrd` vu en cours.

1. Tester l'ensemble sur une seule machine.
2. Observer la génération du stub et du squelette avec les JDK 1.4, et 1.6 (ou $\geq$ 1.6), que constatez vous ? Quelles sont les évolutions d'un JDK à l'autre ? Compléter vos observation en utilisant les options du compilateurs `rmic` telle que `-keep`
3. Séparer le code du client et celui du serveur dans deux répertoires, quels sont les fichiers nécessaires à chacun pour la compilation et pour l'exécution ?
4. Recompiler client et serveur avec la même version de JDK, lancer le serveur et le client sur deux machines différentes de même architecture.
5. Répéter l'opération précédente sur deux machines d'architecture différentes en prenant garde d'activer la même version de la machine virtuelle via la variable `PATH` de votre environnement. (travail personnel) À cette étape, il est peut être nécessaire de créer un fichier `build.xml` pour `ant` en se référent au tutorial `https://ant.apache.org/manual/tutorial-HelloWorldWithAnt.html`. Un exemple appliqué à Java-RMI est disponible sur `https://github.com/AnnabelleGillet/Enseignement/tree/master/JavaRMI/HelloWorld`.
6. Peut-on utiliser 3 machines afin d'avoir un référentiel d'interfaces séparé, que constatez-vous ?
7. Expérimenter la méthode `LocateRegistry`, résumer son fonctionnement.
8. Qu'est ce que le service JNDI proposé dans Java EE ? Quelles sont les relations entre JNDI et Java RMI ?

Exercice 2. Patrons de conception : *object factory* (*)

Réaliser un programme de simulation de gestion de comptes bancaires avec le pattern *object factory*. Vous devez obligatoirement adopter une approche objet et définir une classe `compte` permettant la création d'un compte, proposant les opérations de dépôt, retrait, consultation du solde et permettant un archivage des opérations effectuées. Vous devrez fournir un fichier `build.xml` permettant de compiler le programme, de produire une archive jar autonome contenant le client et une autre contenant le serveur.

1. Réaliser les programmes.
2. Lancer plusieurs clients simultanément effectuant plusieurs opérations. Que se passe-t-il ? Comment est (doit-être) gérée la concurrence ?
3. Que se passe t-il côté client si le serveur est arrêté puis relancé ? Quelles solutions peut-on envisager pour résoudre ce problème ?

Exercice 3. Objets distribués, persistance et *object factory*(*)

On suppose que le nombre de comptes à gérer est important (plusieurs milliers) et qu'ils peuvent être manipulés par plusieurs portion de code applicatif, c'est-à-dire par différentes fonctionnalités d'un système d'informatique de gestion bancaire.

1. Reprendre l'exercice 2, stocker les valeurs des attributs d'un compte dans une base de données au moyen d'une connexion JDBC.

2. Proposer une solution pour éviter de créer trop de connexions JDBC.
3. Identifier des problèmes de concurrence et de synchronisation de processus induits par la solution que vous proposez en réponse à la question 2.
4. En conclusion de vos expériences, proposez quelques règles simples de programmation en environnement distribué avec persistance.

Exercice 4. Objets mobiles

— Réaliser le programme de serveur générique permettant la mobilité des objets Java (exemple du cours).
— Reprendre l'exercice précédent, envoyer à la demande du client un objet `compte` et effectuer le calcul du solde sur le client.
— Modifier le programme afin de réaliser un objet mobile pouvant se déplacer de serveur en serveur en utilisant une route décrite dans un fichier ou dans un tableau.
— Qu'en est-il de l'asynchronisme dans cette approche ?

Exercice 5. Chargement automatique

Modifier le code de votre application afin de mettre en place le téléchargement automatique de classes via un serveur Web. Utiliser votre répertoire `public_html` afin de permettre le partage des classes depuis l'un des serveurs `ufrsciencestech/kundera` ou `depinfo/genet`.

— Quelles classes peuvent être chargées ?
— Existe-t-il un moyen dans Java pour charger dynamiquement des classes au travers du réseau ?

Exercice 6. Activation

1. Utiliser les références persistantes du mécanisme d'activation. Le mettre en œuvre dans le cadre de la gestion des comptes. À quels types de problèmes sont-elles une solution ? Faire une expérimentation en lançant un client, puis en coupant la JVM entre deux appel à des méthodes de l'objet serveur et noter l'état du compte avant et après avoir tué la JVM.
2. Utiliser un objet de la classe `MarshalledObject` pour assurer la persistance dans un fichier. Réitérer l'expérience précédente.
3. Observer ce qu'il se passe lorsque le démon `rmid` est arrêté. Identifier le ou les problèmes et proposez une solution.
4. Comparer le pattern *Object Factory* et le mécanisme d'activation, les deux techniques peuvent elles être utilisées conjointement ?

2 CORBA et l'ORB Orbacus

Exercice 7. Programme HelloWorld client serveur (*)

Le but de cet exercice est de tester les capacités multi-langages et multi-plate-formes de CORBA avec l'implementation Orbacus (anciennement Object Broker) de Progress Software dont le code est accessible mais qui n'est pas un logiciel libre.

1. Reprendre l'exemple simpliste Hello World vu en cours, construire l'interface IDL, implémenter le serveur et réaliser un client (en C++), compiler chaque partie et exécuter serveur et client sur une même machine.
2. Répéter l'exécution en utilisant deux machines différentes.
3. Implémenter un client et un serveur en Java et tester l'exécution du client et du serveur Java.
4. Exécuter de façon mixte sur des architectures différentes (OS et processeur) : client Java/serveur C++, client C++/serveur Java.
5. Tester plusieurs clients simultanés sur un même serveur. Comment résoudre les problèmes de concurrence éventuel ?

Exercice 8. Mesure du temps d'invocations de méthodes à distance

On désire comparer le temps d'invocation de méthodes dans une application client serveur lorsque le client et le serveur sont sur la même machine puis, lorsque le client et le serveur utilisent deux machines différentes. Pour effectuer cette mesure on lancera 1000 fois l'invocation d'une méthode depuis un client vers le serveur. On mesurera le temps de ces 1000 invocations. La méthode du serveur peut se contenter d'incrémenter un compteur. Reprendre le même type de programme et mesure le temps d'acheminement avec JavaRMI.

Exercice 9. Invocation dynamique, ObjectFactory et Activation (*)

Les exemples développés jusqu'à présent utilisent exclusivement le mécanisme d'invocation statique.

1. Au moyen des services `CosNaming` et IFR de CORBA réaliser une invocation dynamique avec un client en Java et un serveur en C++.
2. Expliquer le rôle des référentiels.
3. Expérimentez le mécanisme téléchargement de références via un serveur web.
4. Comment réaliser le pattern ObjectFactory en Java ?
5. Tester le mécanisme d'activation
6. Comparez les mécanismes proposés dans CORBA avec ceux de Java RMI.

Les fonctions nécessaire à la réalisation du programme sont décrites dans le chapitre 7 du manuel de référence d'Orbacus.

Exercice 10. Pont RMI-CORBA

Il est possible depuis Java RMI d'invoquer des objets CORBA. Décrire et tester sur un exemple simple le mécanisme proposé par Java.

Indications et recommandations :

Object Broker (Orbacus) peut fonctionner sur les machines SUN Solaris (SPARC et x86) et Linux Debian 7. Sous Linux et sous Solaris la version 4.3.4 est installée dans le répertoire `/usr/local`.

Vous utiliserez le serveur obaldia et cocteau pour exécuter la partie Linux. Les programmes pour Solaris x86 ou SPARC peuvent tourner sur les serveurs SUN. Attention à bien séparer les codes objets et binaires dans des répertoires différents.

Suivant la machine il faut faire attention à mettre a jour les variables `PATH`, `JAVA_HOME`, `LD_LIBRARY_PATH`, pour désigner le répertoire d'installation d'Orbacus.

Les documentations sont accessibles sur le serveur depinfo `http://depinfo.u-bourgogne.fr/docs/`. Reprenez les exemples développés en cours dans la documentation spécifique à la version 4.

La commande `c++` est générique, la remplacer par `CC` pour utiliser le compilateur C++ constructeur (celui de SUN) ou par `g++` pour utiliser le compilateur C++ GNU. Sur les machines solaris, utiliser le compilateur constructeur.

Indications générales de compilation avec OB 4.3.4 en C/C++ :

```
export PATH=$PATH:/usr/local/OB-4.3.4/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/OB-4.3.4/lib

c++ -c -I. -I/usr/local/OB-4.3.4/include --no-implicit-templates \
   -Wall -Wno-return-type Hello.cpp
c++ -c -I. -I/usr/local/OB-4.3.4/include --no-implicit-templates \
   -Wall -Wno-return-type Hello_impl.cpp
c++ -c -I. -I/usr/local/OB-4.3.4/include --no-implicit-templates \
   -Wall -Wno-return-type Hello_skel.cpp
c++ -c -I. -I/usr/local/OB-4.3.4/include --no-implicit-templates \
   -Wall -Wno-return-type Server.cpp

c++ --no-implicit-templates -Wall -Wno-return-type -L/usr/local/OB-4.3.4/lib -o server \
   Hello.o Hello_skel.o Hello_impl.o Server.o -lOB -lsocket -lnsl

c++ -c -I. -I/usr/local/OB-4.3.4/include  --no-implicit-templates \
   -Wall -Wno-return-type Client.cpp

c++ --no-implicit-templates -Wall -Wno-return-type -L/usr/local/OB-4.3.4/lib -o  \
   client Hello.o Client.o -lOB -lsocket -lnsl
```

Sur les serveurs SUN avec le compilateur CC, ne pas mettre les directives de gestion de warning et ajouter pour l'édition de lien (`-lJTC -lpthread -lposix4`)

Compilation avec OB 4.3.4 en Java :

- La commande `jidl --tie --package hello Hello.idl` permet de générer les fichiers dans un répertoire `hello`
- Pour compiler le client et les fichiers générés : `javac hello/*.java`
- Attention fixer la variable `CLASSPATH`, elle doit contenir les librairies CORBA pour Java, c'est-à-dire les fichiers `.jar` contenu dans le répertoire `/usr/local/JOB-4.3.4/lib`. Par exemple pour ajouter la librairie `OB.jar`
  `export CLASSPATH=$CLASSPATH:/usr/local/JOB-4.3.4/lib/OB.jar`
- Attention aussi a garder les variables `PATH` et `LD_LIBRARY_PATH` dans l'état précédent c'est-à-dire comme si vous aviez effectué un compilation en C++, ceci afin de bénéficier de la commande `jidl`.
- Déposer tous les fichiers du client et du serveur dans le repertoire `hello` (qui contient les classes du package). Pour lancer le client : `java hello.Client`
- Dans le répertoire `/usr/local` vous trouverez les différentes version des JDK officiel de SUN

Compilation avec OB 4.3.4 C++ serveur eluard Solaris 5.10 :

```
CC -c -I. -I/usr/local/OB-4.3.4/include -m64  Hello.cpp
CC -c -I. -I/usr/local/OB-4.3.4/include -m64  Hello_impl.cpp
```

```
CC -c -I. -I/usr/local/OB-4.3.4/include -m64  Hello_skel.cpp
CC -c -I. -I/usr/local/OB-4.3.4/include -m64  Server.cpp
CC  -L/usr/local/OB-4.3.4/lib  -o server Hello.o Hello_skel.o Hello_impl.o \
 Server.o  -m64 -lnsl -lOB  -lpthread -lJTC -lposix4 -lsocket

CC -c -I. -I/usr/local/OB-4.3.4/include -m64  Client.cpp
CC  -L/usr/local/OB-4.3.4/lib  -o client Hello.o Client.o  -m64 -lnsl -lOB \
   -lpthread -lJTC -lposix4 -lsocket
```

Compilation avec OB 4.3.4 C++ serveur cocteau Debian 6 :

```
g++ -c -I. -I/usr/local/OB-4.3.4/include --no-implicit-templates -Wall \
 -Wno-return-type Hello.cpp
g++ -c -I. -I/usr/local/OB-4.3.4/include --no-implicit-templates -Wall \
 -Wno-return-type Hello_impl.cpp
g++ -c -I. -I/usr/local/OB-4.3.4/include --no-implicit-templates -Wall \
 -Wno-return-type Hello_skel.cpp
g++ -c -I. -I/usr/local/OB-4.3.4/include --no-implicit-templates -Wall \
 -Wno-return-type Server.cpp
g++  -L/usr/local/OB-4.3.4/lib --no-implicit-templates  -o server Hello.o \
 Hello_skel.o Hello_impl.o Server.o -lOB  -lnsl -lJTC -lpthread -ldl

g++ -c -I. -I/usr/local/OB-4.3.4/include --no-implicit-templates -Wall \
 -Wno-return-type Client.cpp
g++  -L/usr/local/OB-4.3.4/lib --no-implicit-templates  -o client Client.o \
 Hello.o -lOB  -lnsl -lJTC -lpthread -ldl
```

3 MOM avec JMS/Joram

Exercice 11. Installation et prise en main de JORAM (*)

JORAM (*Java Open Reliable Asynchronous Messaging*) est un MOM conforme aux spécifications JMS 1.1 dont le développement est piloté par le consortium OW2. Le site officiel de JORAM est le suivant : `http://joram.ow2.org/`.

Le premier exercice vise à acquérir les compétences techniques, le second a pour objectif de mettre en perspective les concepts des MOM avec ceux des autres intergiciels dans une problématique des SI distribués.

1. Télécharger la version 5.9 depuis le site officiel et la décompresser dans votre répertoire personnel.
2. Utiliser le tutorial présent sur le site officiel pour exécuter les exemples de base et valider votre installation [1]. La compilation et l'exécution sont contrôlées par `ant` [2] déjà présent sur votre poste. Les fichiers `build.xml` contiennent les directives à utiliser pour la compilation et l'exécution des programmes. Leur lecture est aisée, identifier les différent composants, le serveur de messagerie, le programme d'administration, déterminer les ports sur lesquels ces services sont accessibles (vous en aurez besoin pour la question 6).

1. `https://joram.ow2.io/doc/tutorials/classic/classic.htm`
2. `http://ant.apache.org/`

3. Créer quelques topics et queues avec le programme `classic_admin` du tutorial. Étudier, modifier et recompiler le code source de ce programme.
4. Tester la console d'administration visualVM du tutorial (`https://visualvm.github.io/`) pour consulter le contenu de l'annuaire. Pensez à installer le plug-in MBeans pour visualiser l'annuaire.
5. Tester une communication point à point avec un programme producteur et un programme consommateur. Identifier les différentes sections de vos programmes (connexion à l'annuaire JNDI, création des `Queue`, objets `Connexion` et `Session`).
6. Produire plusieurs centaines de messages, utiliser plusieurs consommateurs, un à un et en parallèle.
7. Tester vos programmes sur plusieurs machines, deux machines dans un premier temps, puis séparer le serveur de messagerie et utiliser trois machines. Il conviendra : 1) d'adapter le script `ant` (fichier `build.xml`), 2) de bien identifier les ports utilisé par l'annuaire JNDI et par le serveur de messagerie.
8. Mesurer le temps de création de 10 000 messages, leur diffusion et leur consommation.
9. Reprendre les trois question précédentes dans le cadre d'une communication en mode *Publish/Subscribe*.

Exercice 12. Cadre applicatif pour l'utilisation des MOMs

De nombreux éléments de réponse sont donnés dans la documentation officielle de Joram (`http://joram.ow2.org/doc/index.html`).

1. Définir l'architecture logicielle pour une application qui diffuse le cours de matières premières et d'actions dans différents canaux (un canal par action ou hiérarchie) à partir d'une base de données qui enregistre les cotations. Réaliser les programmes.
2. Définir l'architecture logicielle pour un SI d'une entreprise de vente en ligne en supposant que les différents services (validation de commande, préparation, facturation et expédition) possèdent chacun des applications existantes et une base de données commune. Définir le *workflow* du traitement d'une commande, réaliser son implémentation avec JMS.
3. Reprendre l'architecture précédente, avec les hypothèse suivantes :
   — chaque sous-système possède sa propre base de données (client, commandes, stock)
   — les diférents services sont géographiquement répartis
   Définir la nouvelle architecture, implanter une gestion distribuée des message au moyen de plusieurs *brokers* de messages.
4. Discuter des fonctionnalités de *schedule queue* et *clustered topics* dans le cadre applicatif décrit dans la question précédente (expérimentation en option).
5. Discuter des problèmes de concurrence et des transactions dans la cadre de l'exemple.
6. Quelle est la signification du mode transactionnel des MOM ?

4 MOM pour les Big Data avec Apache Kafka

Exercice 13. Installation et de Kafka

À partir des éléments donnés en cours, réaliser l'installation de base d'Apache Kafka (`http://kafka.apache.org/`).

1. Décompresser l'archive. Copier et modifier le fichier des propriétés pour pouvoir lancer 2 brokers (faire attention à bien créer des répertoires différents pour chaque broker). Dans le fichier de configuration, identifier les lignes qui contiennent les paramètres de rétention.
2. Lancer Zookeeper et les deux instances de Kafka (broker) dans trois terminaux différents.
3. Créer un topic en spécifiant le facteur de réplication et le nombre de partitions (2 pour cette expérience).
4. Lancer les consoles producteur et consommateur (dans deux terminaux), expérimenter l'envoi de messages. Observer la trace des logs sur chacun des terminaux.

À l'issue de cette étape, si vous n'avez pas eu de message d'erreur votre installation de Kafka est fonctionnelle.

Exercice 14. Réalisation de programmes clients Java

On se propose d'utiliser Apache Kafka depuis un programme Java et de comparer ses fonctionnalités avec JMS. Vous utiliserez Maven (commande `mvn`) pour compiler les programmes et produire une archive jar autonome. Les exemples sont adaptés de ceux développés par Gwen Shapira, un des *commiteers* de Kafka (`https://github.com/gwenshap/kafka-examples`).

1. Compiler le producteur, le lancer.
2. Tester avec le consommateur en mode console.
3. Mesurer le temps de production de 1000 messages.
4. Compiler le programme consommateur, le tester.
5. Répartir les brokers sur deux machines et les clients sur d'autres.
6. Utiliser plusieurs producteurs afin de produire rapidement 10^6 messages.
7. Consommer cette masse de messages en testant deux stratégies : 1) avec les consommateurs dans un même groupe (tester avec 2 et 3 consommateurs), 2) avec des consommateurs de différents groupes.
8. Comparer les fonctionnalités avec JMS.

Exercice 15. Gestion et quotation d'actions

On se place dans le domaine de la finance.

1. On suppose que certaines actions sont regroupées en packages financiers (fond de pension par exemple). Les packages sont proposés par des banques à leurs clients (qui y souscrivent). Le but final est que les banques calculent en temps réel la valeur des souscriptions de leurs clients et qu'ils leur renvoient périodiquement ces valeurs moyennes (par exemple tous les jours). De plus, si une souscription passe en dessous d'un seuil d'alerte le client doit pouvoir être informé en temps réel. Définir une architecture pour réaliser cette fonctionnalité avec Kafka en utilisant les principes de Kafka et Kakfa streams.
2. Définir et réaliser un programme multi-producteurs pour générer un flux de quotations d'actions simulant par exemple 3 ou 4 places financières (Londres, Paris, Tokyo, New-York).
3. Réaliser les programmes pour les banques et les clients au moyen des éléments proposés par Kafka pour organiser et traiter les flux.

5 Serveur d'application JEE

L'objectif de cette sérié d'exercices est d'identifier les fonctionnalités d'un serveur d'applications et notamment les architectures logicielles, de comprendre les impacts des serveurs d'applications sur les méthodes de développement et d'expérimenter plusieurs types de composants du modèle EJB 2 puis d'étudier plus en détails les EJB 3.

Exercice 16. Étude des fonctionnalités d'un serveur d'application

À partir des documentations fournies sur le site d'ObjectWeb (`http://www.ow2.org/`) et sur Wikipédia en version française et anglaise, répondre aux questions suivantes :

1. Qu'est ce qu'un serveur d'applications (critiquer au passage la définition donnée par le site français de Wikipédia et en proposer une nouvelle) ?
2. Quelles sont les grandes fonctionnalités (ou services) assurées par un serveur d'applications ?
3. Qu'est ce qu'un composant logiciel ? Comparer la notion d'objet à celle de composant.
4. Quels sont les modèles de composants existants (pas nécessairement pour les technologies J2EE mais aussi pour CORBA et les solutions Microsoft) ? Pour chaque modèles donner une définition et présenter ses différences par rapport aux autres.
5. Quel est l'impact du paradigme des objets distribués dans les serveurs d'applications ?
6. La plateforme PHP/Zend, la plateforme ZOPE et Tomcat sont-ils des serveurs d'applications ?
7. Quels sont les liens entre les composants logiciel et les principes du SOA (*Service Oriented Architecture*) et les services Web ?

Exercice 17. Prise en main et premières expérimentations (*)

1. Rappeler les différents types de composants EJB 2, donner un exemple d'utilisation dans le cadre d'une application réelle ?
2. Télécharger et installer Jonas `https://bit.ly/2OLMwaq` Vous l'exécuterez sur votre station et surtout pas sur un des serveurs. Utiliser les documentations en ligne pour configurer et tester Jonas (`http://jonas.ow2.org/JONAS_5_2_2/doc/doc-en/html/index.html`). Exécuter les exemples à partir de l'interface Web.
3. Réaliser un EJB 2.1 simple permettant d'afficher bonjour en mode console en vous appuyant sur celui du cours.
   (a) Déterminer les interfaces, implanter les méthodes.
   (b) Adapter un fichier `build.xml` des exemples de Jonas pour compiler votre EJB.
   (c) Identifier les différentes étapes/classes/interfaces nécessaires à la réalisation d'un composant EJB ?
   (d) Comment se déroule le processus de déploiement d'un EJB depuis sa compilation ?
   (e) Faire plusieurs tests de reproductibilité. Les différents fichiers (source, script ant, descripteurs de déploiement) sont sur le serveur pédagogique à l'adresse `https://bit.ly/2DcvIUH`.

4. Dans le répertoire général de Jonas `jonas-full-5.3.0`, étudier le code source des exemples et plus particulièrement de l'application `javaee5-earsample`. Compiler le code, le déployer, l'exécuter. Étudier l'architecture de l'application. Regarder comment sont nommés les ejb, les différents composants de l'application, leur URL d'exposition.
5. Dupliquer l'exemple et l'utiliser comme squelette pour vos futurs EJB. Pour cela, changer le nom des archives, l'enregistrement des composants dans l'annuaire, recompiler et tester.
6. Remplacer la base H2 par Oracle ou PostgreSQL.

Exercice 18. Autres types d'EJB (*)

Les deux premières questions sont techniques, la troisième est une question de synthèse.

1. Réaliser, dans le cadre des exemples étudiés dans les TP précédents, une mini-application permettant de consulter le solde d'un compte bancaire depuis une application web (page JSP ou un servlet par exemple). On ne s'occupera pas d'authentification. Vous devrez mettre en œuvre un EJB (v2.1 ou v3) et valider ses interactions avec un servlet et/ou un script JSP.
2. Créer un EJB de type message, quel peut être son rôle dans l'application bancaire ? Comment y accéder ?
3. Identifier les différences entre EJB 2 et EJB 3. Les différents types d'EJB sont-ils préservés (d'autres sont ils proposés) ? Quels sont les impacts de ce nouveau modèle sur la qualité du code ?

Exercice 19. EJB et mécanismes de persistance (*)

1. En utilisant un EJB 3 respectant le principe BMP, réaliser une persistance sur Oracle au moyen d'une connexion définie dans un annuaire.
2. Quel est l'intérêt dans ce cas du pattern ObjectFactory ?
3. Comment gérer efficacement les connexions au SGBD (pools, caches etc.) ?
4. Réaliser un EJB 3 respectant le principe CMP. Inspirez vous de la documentation de Jonas.
5. Expliquer la différence entre JPA et Hibernate.
6. Comment réaliser une transaction distribuée entre beans ?

6 Ressources

1. Claude Duvallet met à disposition ses supports de cours sur les serveurs d'applications et le modèle de composant EJB `http://litis.univ-lehavre.fr/~duvallet/enseignements/Cours/JEE/COURS-EJB.pdf`.
2. Documentation officielle de Jonas `http://jonas.ow2.org/doc/` dont les guides du développeur.
3. Pour le développement d'un premier exemple d'EJB 2 `http://julien.coron.chez.com/languages/Java/EJB/`