Master 2 - BDIA
BDED - Bases de Données et Environnements Distribués
Travaux Pratiques

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

Révision : 12 septembre 2023

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.

L'exercice 3 est à rédiger et à rendre par binôme (pour le 28/10), deux autres exercices seront à rendre ensuite, ils constitueront la note de travaux pratiques pour le module BDED.

Table des matières

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/gide` 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/gide`) le nom réel des répertoires sous `/usr/gide` :

```
export PATH=/usr/gide/jdk-X.Y/bin:$PATH
export JAVA_HOME=/usr/gide/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 10	JDK 1.5, 1.6, 1.7, 1.8
aragon	Debian 9	JDK 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=ENSE2023`. Pour JDBC, l'URL de connexion depuis les salles de TP est `eluard:1521:ENSE2023`. 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 versions de JDK.

1 Java RMI

Exercice 1. Rappel du Master 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 `HelloWorld` vu en cours.

1. Tester l'ensemble sur une seule machine.
2. Essayer d'exécuter votre code en générant le stub avec `rmic` et en le mettant à disposition du client, puis sans générer le stub et sans que le client en ait connaissance. Que constatez-vous ? Quelles sont les avantages de chacune des méthodes ? Compléter vos observation en utilisant les options du compilateur `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)
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. Créer une expérience pour mesurer le temps d'invocation d'une méthode à distance.

Patron de conception : `Object Factory` distribué (*)

Réaliser un programme de simulation de gestion de compte bancaires. Vous devez obligatoirement adopter une approche objet et définir une classe `Compte` permettant la création d'un compte, proposant les opérations de dépôt, retrait, affichage du solde et permettant un archivage des opérations effectuées. Le programme s'organisera en plusieurs packages : client, serveur et interface.

1. Lancer plusieurs clients simultanément. Que se passe-t-il ? Comment est (doit-être) gérée la concurrence ?
2. Comment réaliser une application qui gère 200 000 comptes ? Expérimentez le pattern de développement *Object Factory*.
3. Que se passe t-il côté client si le serveur est arrêté puis relancé ? Quelles solutions peut-on envisager pour résoudre ce problème ?

Patron de conception : `Bag Of Tasks` (*)

Réaliser un programme de *bag of tasks* avec le pattern *object factory*. Vous devez obligatoirement définir une classe `Task` contenant une méthode `run` permettant d'exécuter la tâche. Le *bag of tasks* aura un méthode `addResult` qui prendre une tâche en paramètre, afin d'intégrer le résultat d'une tâche.

En guise d'implémentation de la tâche, utiliser le calcul de nombres premiers.

1. Réaliser les programmes, attention les tâches s'exécutent sur les clients.
2. Comparer la durée d'exécution et le speedup avec les autres implémentations réalisées en TD.
3. Cette solution est elle plus facilement scalable ? Qu'en est il de la transparence à la localisation, de l'autonomie des participants ? Comparer avec les autres implémentations réalisées en TD.

Exercice 2. 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 portions de code applicatif, c'est-à-dire par différentes fonctionnalités d'un système informatique de gestion bancaire.

1. Reprendre l'exercice 1 (partie `Object Factory` distribué), stocker les valeurs des attributs d'un compte dans une base de données Oracle au moyen d'une connexion JDBC.
2. Proposer une solution pour éviter de créer trop de connexions JDBC (pool ou factory de connexions).
3. Identifier des problèmes de concurrence et de synchronisation de processus induits par la solution que vous proposez.
4. En conclusion de vos expériences, proposez quelques règles simples de programmation en environnement distribué avec persistance.

Exercice 3. Pattern *bag of tasks* avec base de données

L'objectif est de soumettre, à partir de clients multiples, des requêtes à un système qui va les exécuter en parallèle sur un ou plusieurs SGBD. On commencera avec un seul SGBD. Les clients envoient à un objet distribué des requêtes SQL dont la durée d'exécution peut être longue. L'objet distribué les répartit à des workers qui ensuite font un call-back vers le client pour lui délivrer les résultats.

1. Faire un schéma général avec des étapes (comme en cours et TP)
2. Réaliser un diagramme de classes en spécifiant les interfaces, les objets sérialisables, les objets remote.
3. Faire un diagramme de séquence montrant une exécution.
4. Réaliser le programme du serveur et des clients.
5. Qu'en est-il de l'asynchronisme dans cette approche ? Quelles en sont les limites ?
6. Quels sont les impacts au niveau de la concurrence ?
7. Peut-on gérer tout type (UPDATE, INSERT, SELECT, DELETE) de requêtes en même temps ?

Exercice 4. 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 ?

ENCADRÉ 1 – Java RMI et codebase

Le codebase est utilisé pour charger les classes [a]. Lorsque le client utilise des classes du serveur dont il n'a pas connaissance (par exemple une implémentation de `Task`), cela peut nécessiter une configuration spécifique pour rendre ce codebase disponible. Si le codebase n'est pas accessible, cela peut se manifester par une `UnmarshalException` et une `ClassNotFoundException`. Pour cela, il faut définir côté serveur l'emplacement du codebase (un serveur FTP ou HTTP peut également être utilisé pour rendre le code disponible au client) :

```
System.setProperty("java.rmi.server.codebase", "file:///[chemin_des_classes]");
```

Côté client, il faut définir un `SecurityManager` et indiquer que le codebase à utiliser est celui fourni par le serveur :

```
if (System.getSecurityManager() == null) {
    System.setProperty("java.security.policy", "security.policy");
    System.setProperty("java.rmi.server.useCodebaseOnly", "false");
    System.setSecurityManager(new RMISecurityManager());
}
```

Le fichier `security.policy` peut avoir le contenu suivant dans le cadre des TP :

```
grant {
    permission java.security.AllPermission;
};
```

a. https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/codebase.html

2 CORBA et l'ORB Orbacus

Exercice 5. 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 6. 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 7. 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 8. 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 pour les Big Data avec Apache Kafka

Exercice 9. 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 10. 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. Utiliser les exemples du CM comme point de départ [1].

1. Compiler le producteur, le lancer.
2. Tester avec le consommateur en mode console.

1. Ils sont adaptés de ceux développés par Gwen Shapira, un des *commiteers* de Kafka (https://github.com/gwenshap/kafka-examples)

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 11. Gestion et cotation d'actions

On se place dans le domaine de la finance. 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 packages, que les clients pourront ensuite consulter. De plus, si une souscription passe en dessous d'un seuil d'alerte, le client doit pouvoir être informé en temps réel.

1. 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.

4 Serveur d'application JEE

L'objectif de cette série 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, leurs interactions avec les servlets et les scripts JSP.

Exercice 12. Étude des fonctionnalités d'un serveur d'application

À partir des élements du cours, des documentations fournies sur Wildfly (https://docs.wildfly.org/1) et sur Wikipédia en version française et surtout en version anglaise, vous devez être en mesure de 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 JEE 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 Apache Tomcat sont-ils des serveurs d'applications ?
7. Quels sont les liens entre les composants logiciel et les principes du SOA (*Service Oriented Architecture*), les services Web et les micro-services ?

Exercice 13. Prise en main et premières expérimentations (*)

1. Rappeler les différents types de composants EJB 2, donner pour chacun un exemple d'utilisation dans le cadre d'une application réelle ?
2. Télécharger et installer Wildfly version 19 (`https://www.wildfly.org/downloads/`). Il s'agit de la version libre de JBoss Enterprise Application Platform 7 de RedHat pour lequel cette entreprise propose un support. Reportez vous à la section 5 de ce document pour plus de détails sur l'installation de WildFly. Vous lancerez le serveur d'application en mode autonome (*standalone*) sur votre station travail dans les salles et surtout pas sur un des serveurs, ceci même si vous effectuez le TP à distance.
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) Créer votre fichier `pom.xml` pour Maven afin de compiler votre EJB à partir de l'exemple disponible sur Plubel.
   (c) Réaliser un client indépendant avec son propre fichier `pom.xml` afin d'appeler l'EJB directement depuis le `main()`. Inspirez vous de l'exemple donné sur Plubel.
   (d) Identifier les différentes étapes/classes/interfaces nécessaires à la réalisation d'un composant EJB.
   (e) Comment se déroule le processus de déploiement d'un EJB depuis sa compilation ?
   (f) Faire plusieurs tests de reproductibilité.
4. On veut développer une application pour la gestion d'une collection de livres [2], avec les caractéristiques suivantes (figure 1 :
   — avoir deux interfaces metier, l'une pour effectuer les écriture et l'autre pour des lectures/recherches ;
   — permettre la création de nouveaux livres de manières asynchrone à partir d'une queue JMS ;
   — proposer une interface Web pour visualiser les descriptions des livres et des auteurs avec la possibilité d'ajouter des auteurs directement pour un utilisateur authentifié.

   Le service de persistance sera dans un premier temps réalisé par une base de données H2 en mémoire. Un *driver* JDBC et une `data source` que vous aurez définie sur le serveur d'application permettrons aux EJB de se connecter à la base de données. Vous utiliserez JPA et les annotation pour spécifier vos beans entité.

   Le modèle de données de l'application est simple (figure 2) :

2. Cet exemple est adapté d'un exemple proposé dans la documentation de Jonas et consultable à l'adresse suivante : `https://jonas.ow2.org/doc/JONAS_5_1_7/doc/doc-en/html/getting_started_guide.html`

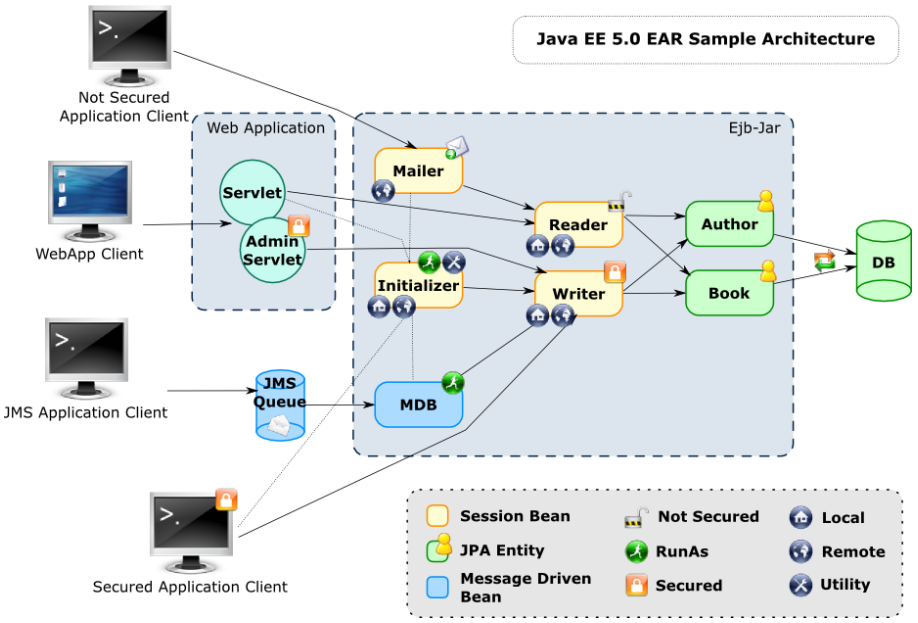

FIGURE 1 – Architecture des beans

— une classe `Author` représenté des auteurs de livres ;
— une classe `Book` représente les livres
— contraintes et hypothèses : on suppose que les livres sont écrits par un seul auteur et qu'un auteur peut avoir écrit plusieurs livres.

Indication pour la réalisation :

— Contruire deux EJB 3 `@Stateless` à partir de deux EJB *entities* `Reader` et `Writer`.
— Le composant `Reader` fournit des methodes de lecture telles que `findAuthor`, `FindBook` qui ne modifient pas les données.
— Le composant `Witer` fournit des méthodes tels que `AddAuthor` et doit être doté d'un accès contrôlé.
— À partir de ces EJB technqiues, construire des EJB plus orientés métier : un Initializer et un Message Driven Bean
  — l'`Initializer` est un bean qui permet si besoin d'initialiser les bean entités. Il est utilisé par les clients pour s'assurer qu'il y a du contenu à afficher aux utilisateurs ;
  — le `MDB` (`Message Driven Bean`) permet de créer un nouveau livre pour chaque message JMS reçu et extrait d'une *queue* spécifique.

Les fonctionnalités supportées par ces beans sont exploitées par des clients soit au travers d'une application Web incluant des pages JSP et des Servlets, soit au moyen d'une application autonome lancée depuis la ligne de commande.

Pour l'application Web, on créra 3 pages :

— une page index avec les instructions pour utiliser l'application ;
— une page publique pour afficher les livres et les auteurs ;
— une page avec authentification pour pouvoir ajouter des auteurs.

Pour les applications clientes et pour poster des messages JMS pour créer des livres, il sera nécessaire d'effectuer une authentification avec JAAS.

5. Remplacer la base H2 de l'exemple par Oracle ou PostgreSQL pour assurer la persistance.

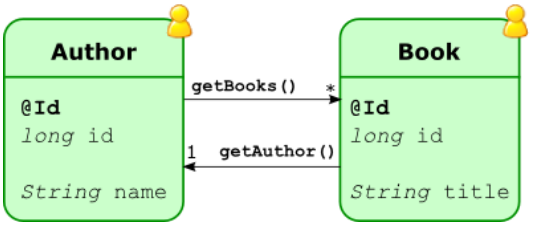

FIGURE 2 – Modèles JPA des données, classes

Exercice 14. 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 15. EJB et mécanismes de persistance (*)

On poursuit le développement de l'application bancaire et sur le modèle de l'exercice 13, on vous demande de développer plusieurs composants.

1. Réaliser un schéma de l'architecture logicielle des composants, et expliquer leur rôle.
2. En utilisant un EJB 3 respectant le principe BMP, réaliser une persistance sur Oracle ou PostgreSQL au moyen d'une connexion définie dans un annuaire.
3. Quel est l'intérêt dans ce cas du pattern ObjectFactory ?
4. Comment gérer efficacement les connexions au SGBD (pools, caches etc.) ?
5. Réaliser un EJB 3 respectant le principe CMP.
6. Expliquer la différence entre JPA et Hibernate.
7. Réaliser un MDB pour l'application bancaire afin d'envoyer les dernières opération effectuées.
8. Comment réaliser une transaction distribuée entre beans ?

5 Ressources

5.1 Liens et bibliographie

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/
4. Exemples de différents types d'EJB pour WildFly : https://github.com/wildfly/quickstart/tree/19.1.x attention, télécharger une archive de la version 19.
5. http://jean-luc.massat.perso.luminy.univ-amu.fr/ens/arch-app/index.html
6. https://olivier-butterlin.developpez.com/tutoriels/javaee/developper-web-jsf-ejb-jta-jpa/

5.2 Configuration d'un serveur d'application Wildfly

Après avoir téléchargé la version 19 du serveur d'applications WildFly (décompresser le fichier `wildfly-19.1.0.0.Final.tar.gz`) il faut fixer le variable `JAVA_HOME` et la variable `PATH` pour désigner le JDK 1.8.

```
1 export JAVA_HOME=/usr/gide/jdk-1.8
2 export PATH=/usr/gide/jdk-1.8/bin:$PATH
3 export JBOSS_HOME=$HOME/TP-EJB-WildFly/wildfly-19.1.0.Final
4 export PATH=$JBOSS_HOME/bin:$PATH
```

Listing 1 – Fichier script `setenv.sh` à adapter et à lancer depuis un terminal avec la commande `source setenv.sh`

Il faut absolument maîtriser l'arborescence pour pouvoir vous repérer lors du déploiement de vos propres applications. Explorez la structure des répertoires du serveur, les fichiers de configuration essentiels, les fichiers journaux, les déploiements des utilisateurs, etc. (tableaux 1, 2).

En mode autonome (*standalone*), chaque instance de serveur WildFly est un processus indépendant (multi-thread), dans ce cas la structure de l'arborescence est décrite à la table 2.

Les fichiers de configuration du serveur *standalone*[3] sont :

— `standalone.xml` (default) : configuration certifiée du profil Web Jakarta[4] ;
— `standalone-ha.xml` : configuration certifiée du profil Web Jakarta avec haute disponibilité ;
— `standalone-full.xml` : configuration certifiée Jakarta Full Platform comprenant toutes les technologies requises pour les EJB, les mappings de données, les web services, etc.
— `standalone-full-ha.xml` : configuration certifiée Jakarta Full Platform avec haute disponibilité ;
— `standalone-microprofile.xml` : configuration orientée vers les micro-services combinée avec JAX-RS (Java API for RESTful Web Services) pour s'intégrer à des services externes ;

3. Le mode domain permet d'avoir un cluster de serveurs d'applications et de les utiliser de manière transparente.

4. Jakarta EE, anciennement Java 2 Enterprise Edition (J2EE), puis Java Platform, Enterprise Edition (Java EE), est une spécification pour la plate-forme Java de SUN puis d'Oracle, destinée aux applications d'entreprise. Cette plateforme étend Java Standard Edition (Java SE) avec différentes API comme le mapping objet-relationnel, les services Web, etc. La plateforme s'appuie sur des composants modulaires exécutés dans un serveur d'applications.

Répertoire	Description
`appclient`	les fichiers de configuration, le contenu du déploiement (zones autorisées en écriture) utilisés par le conteneur du client d'application s'exécutent à partir de cette installation.
`bin`	scripts de démarrage, configuration de démarrage et divers utilitaires en ligne de commande tel que add-user et rapport de diagnostic
`bin/client`	contient les fichiers jar client à utiliser par des clients non basés sur maven
`docs/schema`	fichiers de définition des schémas XML
`docs/examples/configs`	exemples de fichiers de configuration représentant des cas d'utilisation spécifiques
`domain`	les fichiers de configuration, le contenu de déploiement (zones autorisées en écriture) utilisés par les processus en mode domaine s'exécutant à partir de cette installation
`modules`	WildFly est basé sur une architecture modulaire de chargement de classes, les différents modules utilisés dans le serveur sont stockés dans ce répertoire
`standalone`	les fichiers de configuration, le contenu de déploiement (zones autorisées en écritures) utilisés par le serveur autonome unique s'exécutent à partir de cette installation.

TABLE 1 – Structure de l'arboresence de WildFly

— `standalone-microprofile-ha.xml` : configuration orientée micro-services avec haute disponibilité.

Pour lancer le serveur en mode *standalone* utiliser la commandes suivante :
`./standalone.sh --server-config=standalone-full-ha.xml`
Vous pouvez aussi fixer la variable `JBOSS_HOME`, et ajouter `$JBOSS_HOME/bin` dans le `PATH`. Laisser ouvert le terminal dans lequel est lancé le serveur. En ouvrir un autre et fixer les variables d'environnement.

```
1  leclercq@leclercq-Precision-5530:~/TP-EJB-WildFly/wildfly-19.1.0.Final/bin$ ./standalone.sh --server-config=
       standalone-full-ha.xml
2  =========================================================================
3
4    JBoss Bootstrap Environment
5
6    JBOSS_HOME: /home/leclercq/TP-EJB-WildFly/wildfly-19.1.0.Final
7
8    JAVA: /usr/gide/jdk-1.8/bin/java
9
10   JAVA_OPTS:  -server -Xms64m -Xmx512m -XX:MetaspaceSize=96M -XX:MaxMetaspaceSize=256m -Djava.net.
         preferIPv4Stack=true -Djboss.modules.system.pkgs=org.jboss.byteman -Djava.awt.headless=true
11
12 =========================================================================
13
14 10:59:18,130 INFO  [org.jboss.modules] (main) JBoss Modules version 1.10.0.Final
15 10:59:18,470 INFO  [org.jboss.msc] (main) JBoss MSC version 1.4.11.Final
16 10:59:18,479 INFO  [org.jboss.threads] (main) JBoss Threads version 2.3.3.Final
17 10:59:18,588 INFO  [org.jboss.as] (MSC service thread 1-1) WFLYSRV0049: WildFly Full 19.1.0.Final (WildFly Core
       11.1.1.Final) starting
18 ...
```

Listing 2 – Exemple de trace d'exécution correcte

Connectez vous sur la page d'accueil du serveur d'application avec notre navigateur : http://localhost:8080/ puis créer un utilisateur administrateur avec la commande `add-user.sh`. Vous pouvez ensuite accéder à l'interface d'administration avec

Répertoire	Description
`configuration`	fichiers de configuration pour le serveur autonome qui s'exécute à partir de cette base, toutes les informations de configuration du serveur en cours d'exécution se trouvent ici et constituent le seul emplacement pour les modifications de configuration du serveur autonome
`data`	informations persistantes écrites par le serveur pour continuer suite à un redémarrage
`deployments`	le contenu de déploiement (applications) accessibles à l'utilisateur final peut être placé dans ce répertoire pour une détection automatique. L'API de gestion du serveur est recommandée pour l'installation et le déploiement des application. Les capacités d'analyse de déploiement automatique à partir du système de fichiers sont là pour la commodité des développeurs.
`lib/ext`	emplacement des fichiers JAR des bibliothèques installées et utilisées par les applications à l'aide du mécanisme Extension-List
`log`	fichiers journaux du serveur autonome
`tmp`	emplacement des fichiers temporaires écrits par le serveur
`tmp/auth`	emplacement spécial utilisé pour échanger des jetons d'authentification avec les clients locaux afin qu'ils puissent confirmer qu'ils sont locaux au processus AS en cours d'exécution.

TABLE 2 – Structure de la sous-arborescence de WildFly en mode autonome

l'url `http://localhost:9990/console/index.html`. Pour administer le SA vous disposez aussi d'une interface en ligne de commande `jbosscli.sh`, il faut ensuite entrer des commandes comme `connect`, `shutdown` ou bien directement passer les paramètres : `./jboss-cli.sh --connect command=:shutdown`.