

Applications Web dynamiques les Entreprise Java Beans



Daniel Hagimont

**IRIT/ENSEEIH
2 rue Charles Camichel - BP 7122
31071 TOULOUSE CEDEX 7**

**Daniel.Hagimont@enseeiht.fr
<http://hagimont.perso.enseeiht.fr>**

1

L'objectif a été de fournir des technologies logicielles permettant de faciliter le développement des applications Web dynamiques. Nous avons vu précédemment la structuration suivant le modèle MVC, avec le Contrôleur (des servlets), la Vue (des JSP) et le Modèle (pour l'instant des objets Java).

Dans la suite, nous nous intéressons à structurer la partie Modèle (métier).

Préambules



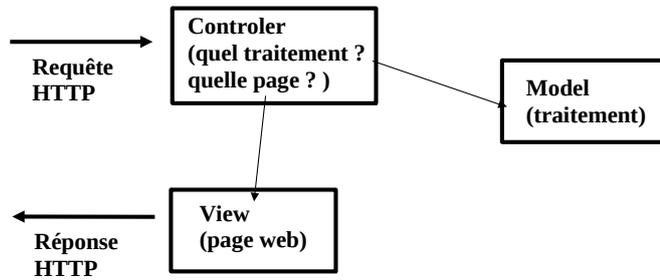
- Les éléments techniques de ce cours reposent sur
 - ◆ Java jdk1.8.0_221
 - ◆ JBoss EAP-7.0.0
 - ◆ Eclipse Java EE IDE for Web Developers

2

Les éléments et exemples présentés dans ce cours reposent (et peuvent être testés) sur les suites logicielles ci-dessus.

Modèle MVC

- Model View Contrôler
- Séparation entre
 - ◆ Le Contrôleur : servlet qui aiguille les requêtes
 - ◆ La Vue : pages JSP pour l'affichage à l'écran
 - ◆ Le Modèle : les classes (beans) qui traitent les données

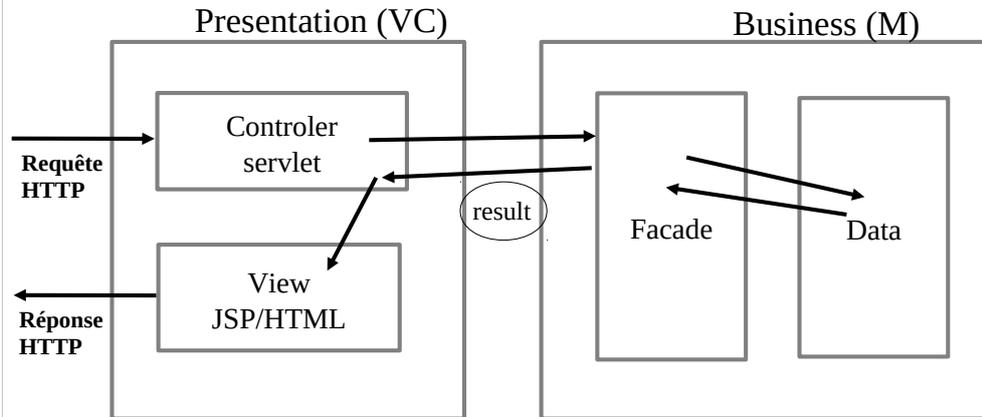


3

Souvenez vous de ce schéma illustrant le modèle MVC.

Le Contrôleur reçoit la requête HTTP, appelle le Modèle pour faire le traitement correspondant à la requête (et récupère un résultat), puis appelle la Vue pour générer la page Web présentant les résultats.

Architecture souhaitée

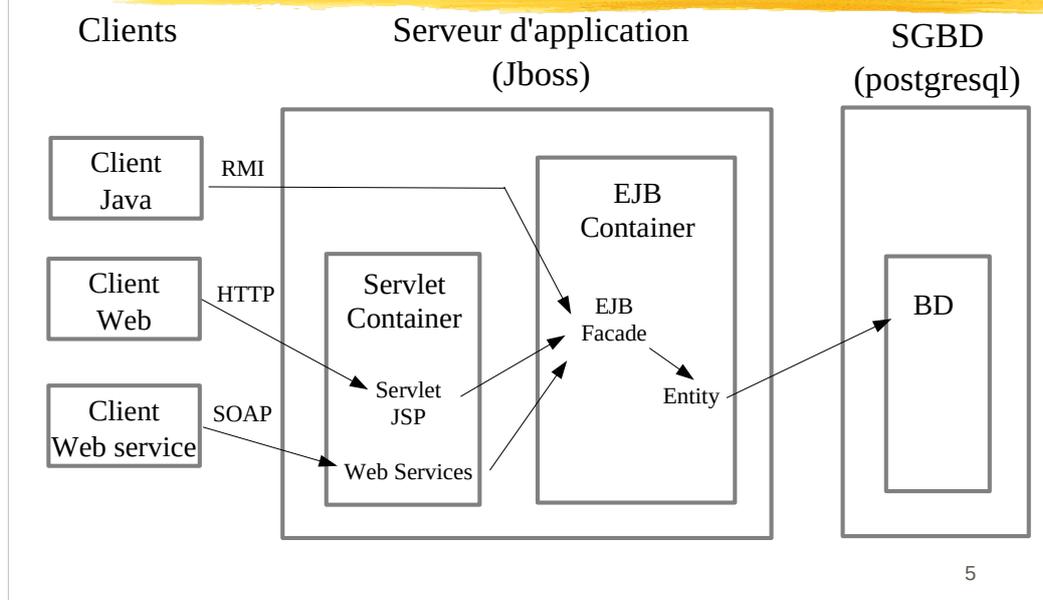


4

On avait également vu ce schéma, dans lequel la partie frontale à gauche (qu'on appelle aussi souvent front-end) correspond au servlet container qui héberge les servlets du Contrôleur et de la Vue (les JSP étant compilées en servlet).

La partie à droite (souvent appelée back-end) correspond au Modèle aussi appelé code métier, car cela implante la logique de l'application indépendamment du fait que ce soit une application Web ou pas. Cette partie back-end est divisée entre Facade et Data. La Facade correspond au code de l'application, qui fournit une interface utilisée par le Contrôleur. La partie Data correspond à la gestion des données que l'on conserve. Cette partie Data repose en général sur une base de données.

Architecture souhaitée



Quand on instancie cette architecture dans un environnement tel que celui proposé par Jboss (mais on retrouve les mêmes principes dans d'autres environnements), on obtient l'architecture ci-dessus.

On voit qu'on a toujours les servlets et JSP dans le servlet container, par contre la Facade est implantée sous la forme d'un EJB (Entreprise Java Beans) dans un autre container (un EJB container). Ces containers peuvent s'exécuter sur la même machine (dans la même JVM) ou sur des machines différentes.

L'intérêt d'implanter la Facade avec des EJB est de bénéficier d'un tas de services gratuitement (sans effort d'implémentation) : des transactions, de la sécurité, mais aussi le fait que votre application métier n'est pas seulement accessible à travers l'application Web (un navigateur), mais également directement via RMI depuis un client Java, ou encore elle est visible comme un Web Service (avec une API WSDL ou REST).

Enfin, la gestion des données repose principalement sur une base de données (ici Postgresql). Mais pour faciliter l'utilisation de la base de données depuis la Facade, des EJB Entity (dans l'EJB container) simplifient énormément le boulot : on ne manipule plus la BD avec des requêtes SQL, mais on voit des objets Java stockés dans la BD.

Nous allons voir tous ces aspects un peu plus en détail. La présentation courante se concentre sur la Facade et la présentation suivante se concentrera sur la gestion des données (avec des EJB Entity).

Les EJB



- Caneva (framework) logiciel pour gérer
 - ◆ Des Entity beans
 - ▶ Représentation des données stockées en base de donnée
 - ▶ Évite d'avoir à implanter l'accès à la BD avec JDBC
 - ◆ Des Session beans
 - ▶ Pour implanter la facade
 - ▶ Inclut le code de l'application qui utilise les données (Entity)
 - ◆ Des Message Driven beans
 - ▶ Pour des traitements asynchrones

6

Les EJB (Entreprise Java Beans) permettent de gérer trois types d'objets que l'on appelle des beans : des entity beans, des session beans et des message driven beans.

Les entity beans sont des objets qui représentent les données de l'application. Leur grand avantage est qu'on les manipule assez naturellement (comme des objets Java standards) et qu'ils sont automatiquement sauvegardés dans la BD. Le va et vient des objets entre la BD et la mémoire est géré automatiquement et vous n'avez (presque) plus à programmer en SQL.

Les session beans correspondent au code de l'application, c'est à dire la Facade. On peut en avoir plusieurs en fonction des besoins. Dans le code de la Facade, on manipule des données, donc des entity beans.

Les messages driven beans sont des objets qui peuvent recevoir des messages JMS (et aussi en envoyer) et modifier l'état de l'application en fonction de ces messages (asynchrones). Les message driven beans ne sont pas traités dans ce cours.

Prise en compte de services techniques

- Services techniques appelé propriétés non-fonctionnelles
 - ◆ La distribution
 - ◆ Les transactions
 - ◆ La persistance
 - ◆ Le cycle de vie
 - ◆ La montée en charge
 - ◆ Le concurrence
 - ◆ Le sécurité
 - ◆ La sérialisation
 - ◆ ...
- Implantation par le container, configuration

7

Un des grands avantages de EJB est de bénéficier d'un tas de services gratuitement (sans effort d'implémentation) :

- les servlets et les EJB peuvent être répartis sur différentes machines, cela ne change pas vos programmes
- les transactions vous garantissent qu'en cas de crash, vous restez dans un état cohérent (état des données avant ou après la transaction courante, mais pas un état intermédiaire incohérent).
- la persistance des données dans la BD est gérée automatiquement
- la gestion du cycle de vie. On entend ici le chargement/déchargement des données en mémoire
- le nombre de serveurs (et aussi de session beans pour la Facade qui fait les calculs) peut être augmenté pour faire face à une montée de la charge, cela ne change pas vos programmes
- la gestion de la concurrence, de la sécurité sont simplifiées

Les EJB

- Entity beans
 - ◆ Représentent les données manipulées par l'application
 - ◆ Chaque Entity est associé à une table au niveau de la base de données
- Session beans
 - ◆ Accessibles à distance (via RMI et IIOP) ou en local, et depuis la servlet
 - ◆ Implémentent le code métier
 - ◆ Stateless session beans : sans état
 - ▶ Une instance pour plusieurs connexions clientes (allouée à partir d'un pool)
 - ▶ Ne conserve aucune donnée dans son état
 - ◆ Statefull session bean : avec état
 - ▶ Création d'une instance pour chaque connexion cliente
 - ▶ Conserve des données entre les échanges avec le client
 - ◆ Singleton: Instance Unique
 - ▶ Création d'une instance unique quelque soit le nombre de connexion.
- Message Driven Beans : Beans de messages
 - ◆ Un listener exécute des traitements à réception d'un message JMS

8

Les EJB (entity, session ou message-driven) sont des classes Java avec des annotations.

Les entity beans représentent les données de l'application. Chaque entity correspond à une table dans la base de données.

Les session beans permettent d'implanter la Facade. Ils peuvent être accédés en local (depuis la même JVM, un seul serveur) ou à distance avec différents protocoles, notamment RMI ou SOAP (Web Service), depuis un client lourd (une application) ou une servlet.

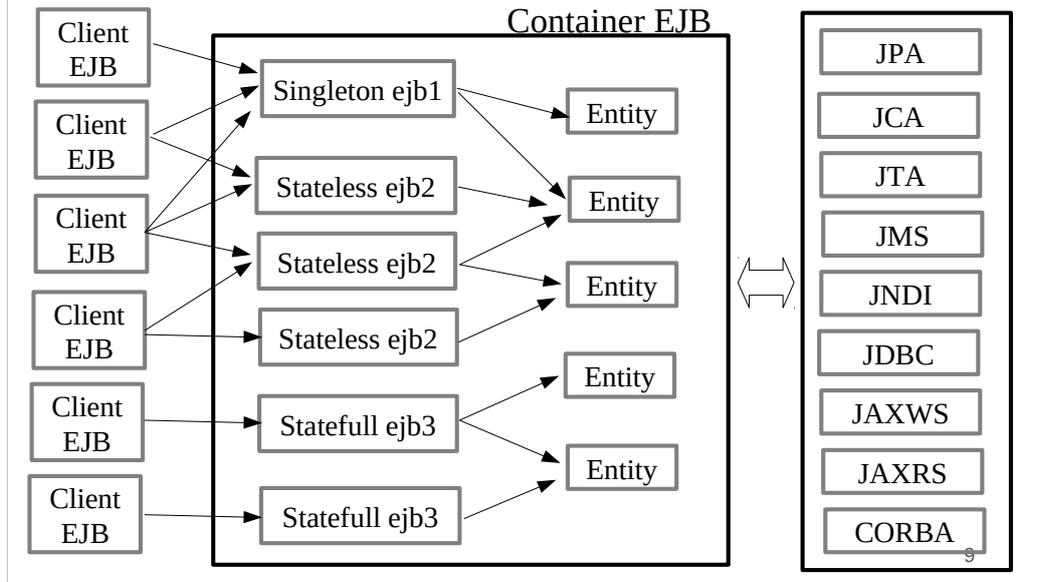
On distingue différents types de session beans :

- les stateless. Quand on déclare un session bean comme étant stateless, cela signifie qu'il n'est pas sensé avoir un état (que du code dans la classe). Cela permet au container de répliquer les beans et de répartir la charge sur ces instances.

- statefull bean. Quand on déclare un session bean comme statefull, on peut gérer un état (des champs dans la classe). Le container crée une instance par client et l'état est privé pour le client, ce qui est similaire à la notion de session dans les servlets (ça fait double emploi). Ce type de bean n'est pas très utilisé, car on gère plutôt les sessions dans les servlets.

- singleton. Quand on déclare un session bean comme singleton, le container crée une instance unique utilisée pour tous les appels. On peut gérer un état, partagé par tous les clients.

Beans et Container



Ce schéma montre l'organisation des EJB dans un container.

Le container peut gérer plusieurs instances de stateless, une seule instance de singleton et une instance de statefull par client.

Le container peut être paramétré pour utiliser (sans effort de programmation) des services comme les transactions (JTA), la persistance (JPA), ...

EJB session

- Interface Remote (annotation @Remote)
 - ◆ Accessible à distance
- Interface Local (annotation @Local)
 - ◆ Accessible en local uniquement (dans le serveur d'application)
 - ◆ Si on ne définit pas d'interface, Local par défaut
- Classe du bean (annotations @Singleton, @Statefull, @Stateless)
 - ◆ Peut implanter les 2 interfaces Local et Remote
- JNDI (service de nommage comme rmiregistry)
 - ◆ Les interfaces définies sont exportées dans JNDI
 - ◆ Nom de la classe utilisée, ou nom passé en attribut du tag
 - ▶ @Singleton(name="monBean")
 - ◆ Une recherche retourne une référence au bean (unique ou à une copie)

10

Quand on programme un session bean, on commence par définir des interfaces Java. Pour un session bean, on peut définir une interface locale et/ou une interface remote, signifiant que le bean peut être accessible en local et/ou à distance. Il suffit de déclarer une interface Java et de l'annoter avec @Local et/ou @Remote. Si on ne déclare pas d'interface (on ne programme que la classe), c'est local par défaut (et on utilise la classe à la place de l'interface).

Ensuite, on définit la classe en l'annotant avec @Singleton, @Statefull ou @Stateless.

Au déploiement, les instances de session bean (une ou plusieurs en fonction du type de bean) sont créées et leurs interfaces (ou des stubs dans le cas remote) sont enregistrés dans un service de nommage JNDI (Java Naming and Directory Interface), l'équivalent du rmiregistry que vous connaissez. Par défaut le nom de la classe est utilisé, mais vous pouvez spécifier le nom à utiliser dans un paramètre de l'annotation de la classe (voir l'exemple avec @Singleton).

Lorsqu'un client (un programme, une servlet ...) recherche dans JNDI, il reçoit une référence (locale ou stub) vers le bean, soit vers une instance unique dans le cas d'un singleton, soit vers une nouvelle copie d'instance pour un statefull ou vers une instance choisie dans un pool pour un stateless (répartition de la charge).

Un exemple : application bancaire

- Gestion de compte
 - ◆ Données
 - ▶ numero, nom, solde
 - ◆ Méthodes
 - ▶ Ajouter un compte
 - ▶ Consulter un compte
 - ▶ Consulter tous les comptes
 - ▶ Créditer sur un compte
 - ▶ Débiter d'un compte
- Depuis un client lourd distant
- Depuis une application Web
- On peut le faire avec un WS

11

Reprenons l'exemple de la gestion de comptes bancaires vu dans le cours précédent.

Nous allons voir comment la Facade peut être programmée comme un session bean, puis nous verrons que ce session bean (correspondant à la partie métier de l'application) peut être appelé depuis un client lourd (avec RMI), depuis une application Web (une servlet) ou sous la forme d'un Web Service (exporté avec un fichier WSDL).

Les données manipulées

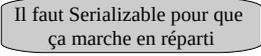
```
public class Compte implements Serializable {
    private int num;
    private String nom;
    private int solde;

    public Compte() {}

    public Compte(int num, String nom, int solde) {
        this.num = num; this.nom = nom; this.solde = solde;
    }

    public String toString() {
        return "Compte [num="+num+", nom="+nom+", solde="+solde+"]";
    }

    // setters and getters
}
```



Il faut Serializable pour que ça marche en réparti

12

On a ici la même classe `Compte` que dans le cours précédent, mais ici, on ajoute `Serializable` pour que les instances de comptes puissent être copiées si on accède à la Facade depuis un client distant.

Interface Remote / interface Local



@Remote

```
public interface IBankLocal {  
    public void addCompte(Compte c);  
    public Collection<Compte> consulterComptes();  
    public Compte consulterCompte(int num)  
        throws RuntimeException;  
    public void debit(int num, int montant)  
        throws RuntimeException;  
    public void credit(int num, int montant);  
}
```

@Local

```
public interface IBankRemote {  
    public void addCompte(Compte c);  
    public Collection<Compte> consulterComptes();  
    public Compte consulterCompte(int num)  
        throws RuntimeException;  
    public void debit(int num, int montant)  
        throws RuntimeException;  
    public void credit(int num, int montant);  
}
```

13

On crée ici deux interfaces, une locale et une distante, avec les 2 annotations correspondantes.

Implantation d'un EJB session

```
@Singleton
public class BankImpl implements IBankLocal, IBankRemote {
    private Map<Integer, Compte> comptes = new Hashtable<Integer, Compte>();

    public void addCompte(Compte c) {
        comptes.put(c.getNum(), c);
    }

    public Collection<Compte> consulterComptes() {
        return comptes.values();
    }

    public Compte consulterCompte(int num) {
        Compte c = comptes.get(num);
        if (c == null) throw new RuntimeException("Compte introuvable");
        return c;
    }
}
```

14

On implante ici la Facade, qui est renommée ici BankImpl.

Notez l'annotation @Singleton.

Notez aussi que la classe implante les deux interfaces (local et remote), impliquant que le session bean est accessible en local et à distance.

L'implantation est la même que dans le cours précédent.

Implantation d'un EJB session

```
public void debit(int num, int montant) {
    Compte c = consulterCompte(num);
    if (c.getSolde() < montant) throw new RuntimeException("Solde insuffisant");
    c.setSolde(c.getSolde() - montant);
}

public void credit(int num, int montant) {
    Compte c = consulterCompte(num);
    c.setSolde(c.getSolde() + montant);
}

@PostConstruct
public void initialisation() {
    addCompte(new Compte(1, "dan", 2000));
    addCompte(new Compte(2, "alain", 4000));
    addCompte(new Compte(3, "luc", 6000));
}
}
```

15

Une petite particularité : la création des 3 comptes était faite dans le constructeur dans le cours précédent.

Ici la création est faite dans une méthode `initialisation()` qui est annotée par `@PostConstruct`.

La raison est que Jboss finalise la création des instances des EJB sessions après l'exécution du constructeur et qu'on ne peut faire des choses dans ce bean avant cette finalisation.

Déploiement dans JBoss

- On déploie l'archive du projet (war) qui inclut
 - ◆ Les 2 interfaces : IBankLocal et IBankRemote
 - ◆ Les 2 classes : Compte et BankImpl
 - ◆ <project-ejb> Export → WAR File
- Dans JBoss
 - ◆ On a lancé JBoss (bin/standalone.sh)
 - ◆ On exporte le war dans standalone/deployments
- Le bean est exporté dans le JNDI du serveur

16

Jboss est un serveur qui fonctionne comme Tomcat que l'on a vu avant.

L'application est packagée dans une archive WAR comme précédemment, qui inclut les pages HTML, JSP et les classes des servlets et EJB.

Le serveur Jboss est lancé avec le script standalone.sh qui est dans le répertoire bin de Jboss.

Cette archive est copiée dans le répertoire standalone/deployments. Jboss détecte l'apparition de l'archive et installe tout, incluant la création des session beans qui sont enregistrés dans le service de nommage (JNDI).

Les clients (client lourd, ou servlets) peuvent récupérer une référence à un session bean auprès de ce service de nommage.

Déploiement dans JBoss

```
hagimont@pc-hagimont: ~/install/EAP-7.0.0/bin
Fichier Édition Affichage Rechercher Terminal Aide
thread 1-8) HV000001: Hibernate Validator 5.2.4.Final-redhat-1
06:44:26,317 INFO [org.jboss.as.ejb3.deployment] (MSC service thread 1-8) WFLYE
JB0473: Les liaisons JNDI du session bean nommées 'BankImpl' dans l'unité de dé
veloppement 'deployment "jbbk.war"' sont les suivantes :
    java:global/jbbk/BankImpl!bk.IBankRemote
    java:app/jbbk/BankImpl!bk.IBankRemote
    java:module/BankImpl!bk.IBankRemote
    java:jboss/exported/jbbk/BankImpl!bk.IBankRemote
    java:global/jbbk/BankImpl!bk.IBankLocal
    java:app/jbbk/BankImpl!bk.IBankLocal
    java:module/BankImpl!bk.IBankLocal
06:44:26,452 INFO [org.jboss.weld.deployer] (MSC service thread 1-4) WFLYWELD00
06: Démarrage des services pour le déploiement CDI : jbbk.war
06:44:26,481 INFO [org.jboss.weld.Version] (MSC service thread 1-4) WELD-000900
: 2.3.3 (redhat)
06:44:26,510 INFO [org.jboss.weld.deployer] (MSC service thread 1-8) WFLYWELD00
09: Lancement du service Weld pour le déploiement jbbk.war
06:44:27,085 INFO [org.wildfly.extension.undertow] (ServerService Thread Pool -
- 61) WFLYUT0021: Contexte web enregistré : /jbbk
06:44:27,118 INFO [org.jboss.as.server] (DeploymentScanner-threads - 1) WFLYSRV
0010: Déploiement de "jbbk.war" (runtime-name: "jbbk.war")
```

17

Quand on a copié le WAR, on observe que les session beans (pour les interfaces local et remote) ont été enregistrés dans JNDI.

Client lourd

- Un projet Java séparé
- Une librairie jboss-client.jar doit être importée dans le client
- Elle gère un JNDI client
- Elle génère des stubs pour les références récupérées
- Le nom du bean dans le JNDI client est
 - ◆ `String jndiName = "ejb:"+appName+"/"+"moduleName+"/"+"distinctName+"!"+viewClassName;`
 - ▶ `appName=""`
 - ▶ `moduleName="jbbk"` (projet eclipse)
 - ▶ `distinctName="BankImpl"` (classe)
 - ▶ `viewClassName="bk.IBankRemote"` (interface)

18

Pour implanter un client lourd qui fait appel (à distance) à notre session bean, on crée un projet Java dans eclipse.

Il faut importer dans ce projet la librairie jboss-client.jar

Le nom sous lequel le session bean (son interface remote) est enregistré dans JNDI est initialisé avec le code ci-dessus.

Conformément au slide précédent ce sera :

`ejb:/jbbk/BankImpl!bk.IBankRemote`

Un client lourd

- Utilise RMI pour accéder à l'interface Remote

```
public class ClientEjb {
    public static void main(String args[]) {
        try {
            String appName="";
            String moduleName="jbbk";
            String distinctName="BankImpl";
            String viewClassName=IBankRemote.class.getName();
            Context ctx = new InitialContext();
            String jndiName = "ejb:"+appName+"/"+moduleName+"/"+distinctName
                +"!"+viewClassName;

            System.out.println(jndiName);
            IBankRemote stub = (IBankRemote)ctx.lookup(jndiName);
            System.out.println("affichage de tous les comptes");
            Collection<Compte> cptes = stub.consulterComptes();
            for(Compte cp:cptes) System.out.println(cp);
        } catch (Exception ex) {ex.printStackTrace();}
    }
}
```

19

Voici le code du client.

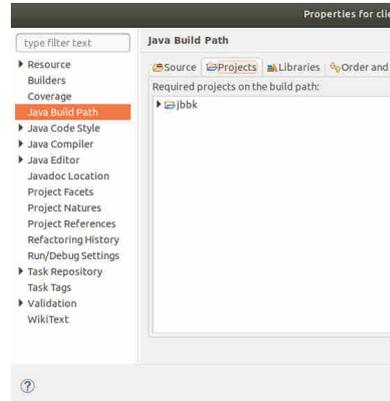
Pour faire une recherche dans JNDI, on crée une instance de InitialContext. Cette instance est initialisée avec des fichiers de configuration qui sont donnés dans les slides suivants.

Cette instance est ensuite utilisée pour rechercher dans JNDI (comme on le fait avec le rmiregistry).

On récupère dans la variable stub une référence à distance au session bean et on peut alors appeler les méthodes du session bean.

Dépendances du client

- <project-client> Build Path → Configure Build Path
- Ajouter <project-ejb> dans Projects
 - ◆ Pour avoir accès aux interfaces du bean
- Ajouter jboss-client.jar dans Libraries
 - ◆ Un stub générique
 - ◆ Se trouve dans <jboss-home>/bin/client



20

Dans le projet eclipse, on ajoute dans le build-path :

- le projet du serveur pour que les classes comme Compte soit connues dans le projet (car les objets Compte sont sérialisés et retournés au client)
- l'archive jboss-client.jar

Configuration JNDI pour le client

à placer dans le classpath du client, par exemple dans src du projet eclipse

jboss-ejb-client.properties

```
endpoint.name=client-endpoint
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=false

remote.connections=default

remote.connection.default.host=localhost
remote.connection.default.port=8080
```

jndi.properties

```
java.naming.factory.url.pkgs=org.jboss.ejb.client.naming
```

21

On ajoute ces 2 fichiers de configuration dans le client pour faire fonctionner le service JNDI.

Exécution client lourd

```
<terminated> Client [Java Application] /home/hagimont/install/jdk1.8.0_221/bin/java (4 mars 2021 à 06:47:42)
ejb:/jbbk/BankImplIbk.IBankRemote
mars 04, 2021 6:47:43 AM org.jboss.ejb.client.EJBClient <clinit>
INFO: JBoss EJB Client version 2.1.4.Final-redhat-1
affichage de tous les comptes
mars 04, 2021 6:47:43 AM org.xnio.Xnio <clinit>
INFO: XNIO version 3.3.6.Final-redhat-1
mars 04, 2021 6:47:43 AM org.xnio.nio.NioXnio <clinit>
INFO: XNIO NIO Implementation Version 3.3.6.Final-redhat-1
mars 04, 2021 6:47:43 AM org.jboss.remoting3.EndpointImpl <clinit>
INFO: JBoss Remoting version 4.0.18.Final-redhat-1
mars 04, 2021 6:47:43 AM org.jboss.ejb.client.remoting.VersionReceiver handleMessage
INFO: EJBCLIENT000017: Received server version 2 and marshalling strategies [river]
mars 04, 2021 6:47:43 AM org.jboss.ejb.client.remoting.RemotingConnectionEJBReceiver associate
INFO: EJBCLIENT000013: Successful version handshake completed for receiver context EJBReceiverContext{clie
Compte [num=3, nom=luc, solde=6000]
Compte [num=2, nom=alain, solde=4000]
Compte [num=1, nom=dan, solde=2000]
```

22

On peut alors exécuter le client et vérifier que tout fonctionne.

Un client web (servlet)

- Soit la servlet est dans une JVM séparée
 - ◆ Initialisation comme un client lourd
- Soit la servlet est dans le serveur d'application (avec les EJB)
 - ◆ Injection de dépendance

```
@EJB  
private IBankLocal facade;
```

- ◆ Au déploiement, variable initialisée avec une instance
 - ▶ de stateless, statefull ou singleton

23

Dans le cadre d'une application Web, c'est la servlet (le Contrôleur) qui doit utiliser le session bean.

Si la servlet s'exécute dans une JVM différente que celle exécutant le session bean, elle doit s'initialiser comme un client lourd.

Si la servlet est dans la même JVM que le session bean, on peut utiliser ce qu'on appelle l'injection de dépendance. Il suffit de déclarer une référence au session bean (son interface locale) dans la servlet et de l'annoter avec @EJB. La variable (facade) sera initialisée automatiquement avec la référence au session bean.

Un client web (servlet)

```
@WebServlet("/Controller")
public class Controller extends HttpServlet {
    private static final long serialVersionUID = 1L;
    @EJB
    private IBankLocal facade;
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        try {
            String action=request.getParameter("action");
            if (action.equals("consulter")) {
                int num=Integer.parseInt(request.getParameter("num"));
                request.setAttribute("num", num);
                request.setAttribute("compte", facade.consulterCompte(num));
            }
        }

        // et la suite de l'application qu'on avait vue avant (servlet + JSP)
    }
}
```

24

Voici le code de la servlet de notre exemple.

On voit bien que l'on n'instancie plus la Facade, mais on utilise une injection de dépendance (@EJB).

Un client Web Service

- Il faut exporter le singleton comme un WS
 - ◆ @WebService, @WebMethod

```
@Remote
@WebService
public interface IBankRemote {
    public void addCompte(Compte c);
    public Collection<Compte> consulterComptes();
    public Compte consulterCompte(int num);
    @WebMethod
    public void debit(int num, int montant);
    @WebMethod
    public void credit(int num, int montant);
}

@Singleton
@WebService(endpointInterface="bk.IBankRemote", serviceName="BankWS")
public class BankImpl implements IBankLocal, IBankRemote {
    ...
}
```

25

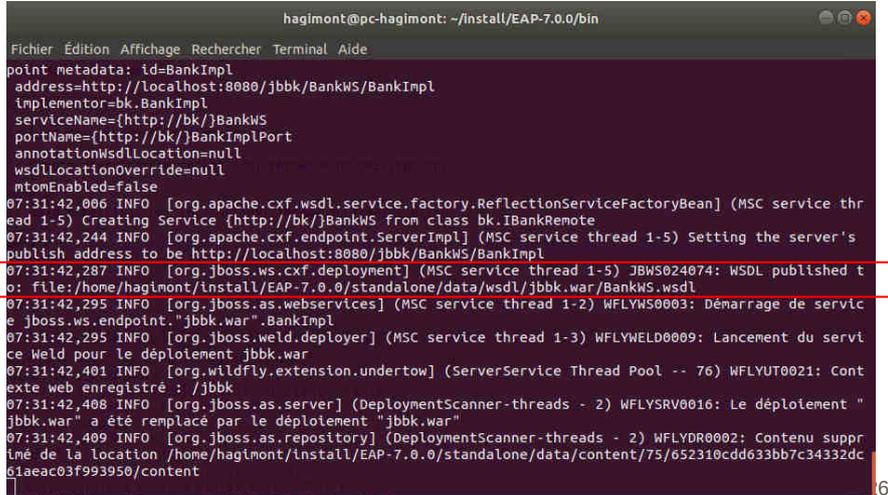
Les EJB permettent également d'exporter le session bean (la Facade) comme un WebService.

Il suffit d'annoter l'interface remote de notre session bean avec @WebService et d'annoter les méthodes accessibles dans cette interface avec @WebMethod.

Il faut également annoter la classe du session bean avec @WebService en spécifiant l'interface du session bean.

Un client Web Service

- Et on récupère le fichier WSDL généré



```
hagimont@pc-hagimont: ~/install/EAP-7.0.0/bin
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
point metadata: id=BankImpl
address=http://localhost:8080/jbbk/BankWS/BankImpl
implementor=bk.BankImpl
serviceName={http://bk/}BankWS
portName={http://bk/}BankImplPort
annotationWsdLocation=null
wsdlLocationOverride=null
ntomEnabled=false
07:31:42,006 INFO [org.apache.cxf.wsdl.service.factory.ReflectionServiceFactoryBean] (MSC service thr
ead 1-5) Creating Service {http://bk/}BankWS from class bk.IBankRemote
07:31:42,244 INFO [org.apache.cxf.endpoint.ServerImpl] (MSC service thread 1-5) Setting the server's
publish address to be http://localhost:8080/jbbk/BankWS/BankImpl
07:31:42,287 INFO [org.jboss.ws.cxf.deployment] (MSC service thread 1-5) JBWS024074: WSDL published t
o: file:/home/hagimont/install/EAP-7.0.0/standalone/data/wsd/jbbk.war/BankWS.wsdl
07:31:42,295 INFO [org.jboss.as.webservices] (MSC service thread 1-2) WFLYWS0003: Démarrage de servic
e jboss.ws.endpoint."jbbk.war".BankImpl
07:31:42,295 INFO [org.jboss.weld.deployer] (MSC service thread 1-3) WFLYWELD0009: Lancement du servi
ce Weld pour le déploiement jbbk.war
07:31:42,401 INFO [org.wildfly.extension.undertow] (ServerService Thread Pool -- 76) WFLYUT0021: Cont
exte web enregistré : /jbbk
07:31:42,408 INFO [org.jboss.as.server] (DeploymentScanner-threads - 2) WFLYSRV0016: Le déploiement "
jbbk.war" a été remplacé par le déploiement "jbbk.war"
07:31:42,409 INFO [org.jboss.as.repository] (DeploymentScanner-threads - 2) WFLYDR0002: Contenu suppr
imé de la location /home/hagimont/install/EAP-7.0.0/standalone/data/content/75/652310cdd633bb7c34332dc
61aeac03f993950/content
```

Lors du déploiement, on voit que le Web Service est créé et un fichier WSDL a été généré et peut être diffusé aux clients.

Notons que la génération du Web Service s'est traduit par la génération d'une servlet qui peut recevoir les appels au Web Service (requêtes HTTP) et transforme ces appels en appels au session bean.

Un client WS

- La chaîne d'outil génère les stubs à partir du fichier WSDL

```
public class DebitDan {  
  
    public static void main(String args[]) {  
        try {  
            IBankRemoteProxy p = new IBankRemoteProxy();  
            IBankRemote b = p.getIBankRemote();  
            b.debit(1, 20);  
        } catch (Exception ex) {  
            ex.printStackTrace();  
        }  
    }  
}
```

27

Comme on l'a vu dans le cours Intergiciels dans la partie sur les Web Services, le fichier WSDL peut être utilisé dans un projet eclipse pour générer les stubs permettant de réaliser des appels au Web Service.

Nous avons ici le code d'un client qui appelle le Web Service.

Bilan EJB

- **Implantation de la Facade avec un EJB session**
 - ◆ Peut être singleton, stateless ou statefull
 - ◆ Facilite la gestion en cluster
 - ◆ Permet de gérer pleins de propriétés non-fonctionnelle (pas détaillé ici)
 - ◆ Peut être appelé depuis un client lourd, une servlet, ou comme un WS
- **Gestion des données**
 - ◆ Pour l'instant des objets Java dans le EJB session (pas persistant)
 - ◆ On voudrait gérer ces objets dans une base de données

28

En résumé, les EJB permettent d'implanter la Facade sous la forme d'un session bean. Ce session bean peut être unique (Singleton), avec état (Statefull, comme une session, mais c'est très peu utilisé) ou sans état (Stateless, pour répartir la charge sur plusieurs exemplaires).

Ces session beans permettent de faciliter la gestion dans un cluster (ensemble de serveurs répartis pour avoir une puissance de calcul).

Ces session beans apportent la gestion de propriétés non-fonctionnelles concernant les transactions, la persistance, la sécurité ... pleins d'aspects que je ne peux détailler dans ce cours.

Ces session beans peuvent être appelés depuis une servlet, un client lourd ou sous la forme d'un Web Service.

Le deuxième aspect des EJB est la gestion des données. Pour l'instant, dans l'exemple précédent des comptes bancaires, les données sont gérées dans une table dans le session bean. Ces données ne sont donc pas persistantes, car si on arrête Jboss (ou si il crashe), on perd toutes les données.

Le cours suivant montre comment on peut gérer les données dans une base de données, et de façon simple d'un point de vue programmation (sans avoir à utiliser JDBC qui peut s'avérer complexe quand le schéma de données est complexe).