Les descripteurs de déploiement

Il y a deux descripteurs de déploiement. Le descripteur de déploiement standard le fichier *web.xml*, et le descripteur de déploiement spécifique, le fichier *jboss-web.xml*. Le descripteur de déploiement standard est portable d'un serveur d'application à l'autre, le descripteur de déploiement spécifique est propre au serveur JBoss.

Le descripteur standard contient la configuration de l'application web, des références aux ressources. Le descripteur de déploiement spécifique contient la liaison entre les ressources référencées dans le fichier web.xml et leur déploiement dans le serveur d'application.

Les descripteurs de déploiement sont situés dans le répertoire WEB-INF de l'application Web.

1. Descripteur de déploiement standard web.xml

L'élément racine `<web-app>` du fichier *web.xml* comprend les éléments fils suivants :

- `<icon>`, `<display-name>`, `<description>` qui sont des éléments utilisés par certains outils de gestion de déploiement, pour afficher une icône, un nom et une description associés à l'application Web.
- `<distribuable>` permet de marquer l'application comme distribuable sur plusieurs serveurs, ce qui permet d'exploiter la répartition de charge contrôlée par le serveur.
- `<context-param>` qui contient la valeur des paramètres utilisables dans toute l'application. Ces paramètres, définis dans le fichier *web.xml*, peuvent être utilisés dans les servlets et les pages JSP. Les paramètres sont déclarés dans les balises `<param-name>` et `<param-value>`.
- `<filter>` et `<filter-mapping>` qui permettent de déclarer des filtres sur des URL. Les filtres vont permettre de faire des traitements avant que la requête n'arrive à la servlet ou à la JSP, ou après que la réponse soit générée. Ce type de classe doit implémenter l'interface `javax.servletFilter`.
- `<listener>` qui permet de déclarer des écouteurs sur des événements : entre autres, démarrage et retrait de l'application web, début et fin de session.
- `<servlet>` et `<servlet-mapping>` qui permettent de déclarer les servlets et de les associer avec des URL.
- `<session-config>` qui contient l'élément `<session-timeout>` permettant de définir le temps maximal d'une session, en minutes. Le temps par défaut de 30 minutes est défini dans le fichier *server\default\deploy\jboss-web.deployer\conf\web.xml*. Vous pouvez régler ce temps, application par application.
- `<mime-mapping>` qui reprend les types MIME, ces types sont configurés dans le web.xml par défaut.
- `<welcome-file-list>` qui déclare la liste des noms des pages d'accueils par défaut, si aucune ressource n'est demandée dans l'URL.
- `<error-page>` définit les pages d'erreur à renvoyer en cas de code erreur HTTP, ou d'exception Java.
    - `<error-code>` contient le code erreur HTTP, par exemple 404 ;
    - `<exception-type>` contient le nom de la classe d'exception ;
    - `<location>` contient le chemin de la page d'erreur à afficher.
- `<security-constraint>`, `<security-role>`, `<login-config>` permettent de mettre en place des sections du site qui sont soumises à des autorisations d'accès. Nous aurons l'occasion de revenir plus en détail sur l'utilisation de ces balises, dans le chapitre Gestion de la sécurité consacré à la sécurité.
- `<resource-ref>`, `<ejb-ref>` déclarent des références vers des ressources, ou des EJB, qui sont déployées sur le serveur. Ces déclarations seront associées aux ressources réelles dans le fichier de déploiement spécifique *jboss-web.xml*.

2. Descripteur de déploiement spécifique jboss-web.xml

Les ressources déclarées dans le fichier web.xml par l'intermédiaire des éléments <resource-ref>, font référence au nommage JNDI de l'application, dans le contexte `java:com/env`. Les ressources réelles sont nommées dans contexte JNDI plus global. Il serait dommage que les servlets et JSP, si elles utilisent des ressources, soient obligées de connaître les noms JNDI de plus haut niveau, cela nuirait à la portabilité. Le fichier jboss-web.xml va permettre d'associer les contextes JNDI.

L'élément racine de ce fichier est `<jboss-web>`. Nous allons découvrir ce fichier en l'illustrant par quelques exemples.

a. Nom de l'application

```
<jboss-web>
    <context-root>chap5</context-root>
</jboss-web>
```

L'élément `<context-root>` permet de préciser le nom de l'application web. Si ce nom n'est pas précisé, l'application sera nommée avec le nom du fichier war.

Si notre archive s'appelle "Chap 5 - Web.war", le nom de l'application déployée sera "Chap 5 - Web" qui correspondra à l'URL http://localhost:8080/Chap 5 - Web/.

Si une valeur est donnée à l'élément `<context-root>`, le nom de l'application correspondra à cette valeur, dans notre cas : http://localhost:8080/chap5/.

b. Association de ressources

Association d'une source de données

Dans le fichier *web.xml*, la ressource est déclarée de la manière suivante :

```
<web-app>

...

  <resource-ref>
    <res-ref-name>jdbc/Bovoyage</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>

...

</web-app>
```

Dans la servlet, la récupération de la ressource est effectuée par un code ressemblant à cela :

```
InitialContext context = new InitialContext();
DataSource ds =
  (DataSource)context.lookup("java:comp/env/jdbc/Bovoyage");
```

Dans le fichier *jboss-web.xml*, l'association est effectuée de la manière suivante :

```
<jboss-web>

...

    <resource-ref>
        <res-ref-name>jdbc/Bovoyage</res-ref-name>
        <jndi-name>java:jdbc/BovoyageDS</jndi-name>
    </resource-ref>


...
```

```
</jboss-web>
```

Si nous interrogeons la liste des noms JNDI, nous pouvons voir que l'association a été effectuée :

Contexte JNDI `java:` :

```
java: Namespace

  +- jaas (class: javax.naming.Context)
  |  +- HsqlDbRealm (class: org.jboss.security.plugins.SecurityDomainContext)
  |  +- jbossmq (class: org.jboss.security.plugins.SecurityDomainContext)
  |  +- JmsXARealm (class: org.jboss.security.plugins.SecurityDomainContext)
  +- TransactionPropagationContextImporter (class: com.arjuna.ats.internal.
jbossatx.jta.PropagationContextManager)
  +- comp.ejb3 (class: javax.naming.Context)
  |  NonContext: null
  +- JmsXA (class: org.jboss.resource.adapter.jms.JmsConnectionFactoryImpl)
  +- DefaultDS (class: org.jboss.resource.adapter.jdbc.WrapperDataSource)
  +- StdJMSPool (class: org.jboss.jms.asf.StdServerSessionPoolFactory)
  +- TransactionManager (class: com.arjuna.ats.jbossatx.jta.
TransactionManagerDelegate)
  +- test_seamDatasource (class: org.jboss.resource.adapter.jdbc.
WrapperDataSource)
  +- TransactionPropagationContextExporter (class: com.arjuna.ats.internal.
jbossatx.jta.PropagationContextManager)
  +- ConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)
  +- jdbc (class: org.jnp.interfaces.NamingContext)
  |  +- BovoyageDS (class: org.jboss.resource.adapter.jdbc.WrapperDataSource)
```

Contexte JNDI de l'application Web :

```
java:comp namespace of the Chap 5 - Web.war application:

  +- UserTransaction[link -> UserTransaction] (class: javax.naming.LinkRef)
  +- env (class: org.jnp.interfaces.NamingContext)
  |  +- unEntier (class: java.lang.Integer)
  |  +- jdbc (class: org.jnp.interfaces.NamingContext)
  |  |   +- Bovoyage[link -> java:jdbc/BovoyageDS] (class: javax.naming.
LinkRef)
```

Nous constatons donc que les contextes JNDI ont bien été effectués. Il est important de travailler selon ce procédé, plutôt que d'aller, dans l'application Web, chercher dans les contextes JNDI de haut niveau. En effet cela permet d'assurer une meilleure portabilité entre les serveurs d'applications.

Association avec un EJB

Dans ce type d'association, nous pouvons utiliser des EJB situés :

- sur le même serveur, donc les interfaces locales seront utilisées ;
- sur un autre serveur, et alors les interfaces distantes sont utilisées.

Dans le fichier web.xml, les références locales sont mises en place par l'élément `<ejb-local-ref>` et les références distantes le sont par `<ejb-ref>`.

```
<web-app>

...

  <ejb-ref>
    <ejb-ref-name>ejb/calc</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>
      fr.eni.editions.jboss.ejb2.calculette.CalculetteHome
    </home>
    <remote>
      fr.eni.editions.jboss.ejb2.calculette.Calculette
    </remote>
  </ejb-ref>
```

```
  <ejb-local-ref>
    <ejb-ref-name>ejb/calcH</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local-home>    fr.eni.editions.jboss.ejb2.calculette.CalculetteLocalHome
    </local-home>
    <local>
        fr.eni.editions.jboss.ejb2.calculette.CalculetteLocal
    </local>
  </ejb-local-ref>
</web-app>
```

`<ejb-ref-name>` contient la valeur utilisée dans la recherche dans le contexte JNDI.

Cette recherche produirait un code ressemblant à celui-ci, pour l'EJB local :

```
InitialContext context = new InitialContext();
CalculetteLocalHome home =
(CalculetteLocalHome)context.lookup("java:comp/env/ejb/calcH");
CalculetteLocal bean = home.create();
```

Ou produirait un code ressemblant à celui-ci, pour l'EJB distant :

```
InitialContext context = new InitialContext();
CalculetteHome home =
(CalculetteHome)context.lookup(("java:comp/env/ejb/calc");
Calculette bean = home.create();
```

Le fichier *jboss-web.xml* met en place les associations de contexte :

```
<jboss-web>

...

    <ejb-ref>
        <ejb-ref-name>ejb/calc</ejb-ref-name>
        <jndi-name>ejb2/calculette/remote</jndi-name>
    </ejb-ref>

    <ejb-local-ref>
        <ejb-ref-name>ejb/calcH</ejb-ref-name>
        <local-jndi-name>
            ejb2/calculette/local
        </local-jndi-name>
    </ejb-local-ref>


</jboss-web>
```

Si nous interrogeons la liste des noms JNDI, nous pouvons voir que l'association a été effectuée :

Dans le contexte JNDI global :

```
  +- ejb2 (class: org.jnp.interfaces.NamingContext)
  |  +- calculette (class: org.jnp.interfaces.NamingContext)
  |  |  +- local (proxy: $Proxy60 implements interface fr.eni.editions.jboss.
ejb2.calculette.CalculetteLocalHome)
  |  |  +- remote (proxy: $Proxy62 implements interface fr.eni.editions.
jboss.ejb2.calculette.CalculetteHome,interface javax.ejb.Handle)
```

Dans le contexte JNDI de l'application Web :

```
java:comp namespace of the Chap 5 - Web.war application:

  +- UserTransaction[link -> UserTransaction] (class: javax.naming.LinkRef)
  +- env (class: org.jnp.interfaces.NamingContext)
  |  +- unEntier (class: java.lang.Integer)
  |  +- jdbc (class: org.jnp.interfaces.NamingContext)
  |  |   +- Bovoyage[link -> java:jdbc/BovoyageDS] (class: javax.naming.
LinkRef)
```

```
  |  +- ejb (class: org.jnp.interfaces.NamingContext)
  |  |   +- calc[link -> ejb2/calculette/remote] (class: javax.naming.
LinkRef)
  |  |   +- calcH[link -> ejb2/calculette/local] (class: javax.naming.
LinkRef)
```

Accès à l'environnement.

Des valeurs peuvent être mises en contexte JNDI par l'intermédiaire de l'élément `<env-entry>` dans le fichier *web.xml*.

```
<web-app>

...

  <env-entry>
    <env-entry-name>unEntier</env-entry-name>
    <env-entry-type>java.lang.Integer</env-entry-type>
    <env-entry-value>25</env-entry-value>
  </env-entry>
</web-app>
```

- `<env-entry-name>` est le nom JNDI dans le contexte de l'application ;
- `<env-entry-type>` est le type de l'entrée. Cela peut être `java.lang.Boolean`, `java.lang.Integer`, `java.lang.Double`, `java.lang.Float` ou `java.lang.String` ;
- `<env-entry-value>` est la valeur de l'entrée.

Cela se traduit par la mise en place dans le contexte JNDI de l'entrée :

```
java:comp namespace of the Chap 5 - Web.war application:

  +- UserTransaction[link -> UserTransaction] (class: javax.naming.LinkRef)
  +- env (class: org.jnp.interfaces.NamingContext)
  |  +- unEntier (class: java.lang.Integer)
```

Et la recherche de l'entrée pourrait être un code ressemblant à celui-ci :

```
Integer entier = null;
try
{
    InitialContext ctx = new InitialContext();
    entier = (Integer)ctx.lookup("java:comp/env/unEntier");
    System.out.println("<<< valeur de l'entier : "+entier);
}
catch(Exception e)
{
    System.out.println("<<< erreur : "+e);
}
```

Les recherches dans un contexte JNDI peuvent être remplacées par de l'injection de ressources avec Tomcat 6. La version de Tomcat utilisée ici par JBoss est la 5.5.

Les hôtes virtuels

Les hôtes virtuels permettent d'associer plusieurs noms de domaine avec une application Web. Le nom de domaine doit être enregistré auprès d'une autorité DNS. Mais vous pouvez aussi forcer un nom de domaine en modifiant le fichier hosts. Ce fichier est situé :

- sous Ubuntu, dans */etc/hosts* ;
- sous Windows, dans *C:\WINDOWS\system32\drivers\hosts*.

```
127.0.0.1       localhost
127.0.0.1       www.toto.net
127.0.0.1       www.titi.net
```

Des domaines et des adresses IP sont associés dans ce fichier. Ici, il y deux nouveaux domaines associés à la machine. Lors de l'invocation des domaines *www.titi.net* et *www.toto.net*, l'adresse 127.0.0.1 sera réellement invoquée.

Le fichier server.xml, situé sous *<répertoire jboss>/server/default/conf*, a été modifié pour ajouter les lignes suivantes dans la balise `<Engine>` :

```
<Engine>

...

<Host name="toto"
  autoDeploy="false" deployOnStartup="false" deployXML="false"
  configClass="org.jboss.web.tomcat.security.config.JBossContextConfig">

  <Alias>www.toto.net</Alias>
  <Alias>www.titi.net</Alias>
  <Valve className="org.jboss.web.tomcat.service.jca.CachedConnectionValve"
cachedConnectionManagerObjectName="jboss.jca:service=CachedConnectionManager"
      transactionManagerObjectName="jboss:service=TransactionManager" />
</Host>

<Engine>
```

La balise `<Host>` représente un hôte virtuel. Elle possède les attributs suivants :

- `name` : le nom de l'hôte virtuel, habituellement un nom réseau ;
- `appBase` : chemin relatif ou absolu où se trouvent les applications Web, par défaut le répertoire de déploiement ;
- `autoDeploy` : auto déploiement des applications, ici à false car un service JBoss déploie les applications et non pas Tomcat ;
- `deployOnStartup` : déploiement automatique au démarrage ;
- `deployXML` : désactivation des fichiers XML de contexte ;
- `configClass` : classe de gestion de la configuration.

Les balises `<Alias>` permettent d'ajouter des alias au nom réseau défini dans l'attribut name de `<Host>`.

La balise `<Valve>` permet de définir des éléments qui viendront s'insérer dans le traitement des requêtes.

Dans le fichier *jboss-web.xml*, nous pouvons associer un hôte virtuel défini dans *server.xml* avec notre application Web, et un nom d'application.

```
<jboss-web>
    <context-root>/</context-root>
    <virtual-host>toto</virtual-host>
```

```
</jboss-web>
```

L'élément `<context-root>` contient "/" qui est l'application par défaut. Par ce biais, nous avons associé notre application aux URL :

- http://toto:8080/
- http://www.toto.net:8080/
- http://www.titi.net:8080/

Mais vous pouvez vérifier que l'URL http://localhost:8080/ pointe toujours vers la page d'accueil de JBoss.

Configuration du conteneur d'EJB

Comme nous avons pu le voir lors des rappels sur les EJB au chapitre Introduction, le client d'un EJB ne crée jamais directement d'instance d'EJB, et n'invoque jamais directement une méthode sur l'EJB. C'est le conteneur d'EJB du serveur JBoss qui est chargé d'effectuer ces appels.

Pour un EJB2 par exemple, il est de la responsabilité du serveur de fournir les `javax.ejb.EJBHome` et `javax.ejb.EJBObject` pour un EJB, le client référencera `EJBHome` pour tout ce qui a trait au cycle de vie de l'EJB et `EJBObject` pour les invocations des méthodes métier.

Côté client

Côté client, les phases de déploiement d'un EJB2 et la fabrication des proxies par le serveur peuvent être résumées par les étapes suivantes :

1. Le moteur de déploiement d'EJB (`org.jboss.ejb.EJBDeployer`) déploie l'archive JAR de l'EJB. Un module EJB (`org.jboss.ejb.EJBModule`) est alors créé.

2. Lors de la phase de création du module EJB, une fabrique de proxy (`org.jboss.ejb.EJBFactory`) gère la création des proxies des interfaces Home et Object de l'EJB.

3. La fabrique de proxy crée un proxy logique composé d'un proxy dynamique (`java.lang.reflect.Proxy`), des interfaces Home de l'EJB, d'une implémentation d'un ProxyHandler (`java.lang.reflect.InvocationHandler`) et des classes d'interception. Les interfaces homes de l'EJB sont liées au contexte JNDI.

4. Lorsque le client recherche dans un contexte JNDI l'interface Home de l'EJB, le proxy associé est transporté dans la machine virtuelle du client.

Lorsque le client invoque une méthode de l'EJB, il y a création d'une instance de la classe `org.jboss.invocation.Invocation` qui encapsule la demande du client. Cette invocation passe au travers d'une chaîne de classes d'interception. L'ensemble des classes associées à la fabrique des proxies et liées à la chaîne d'interception des invocations sont décrites dans le fichier *<répertoire du serveur>/conf/standardjboss.xml* dont voici un extrait :

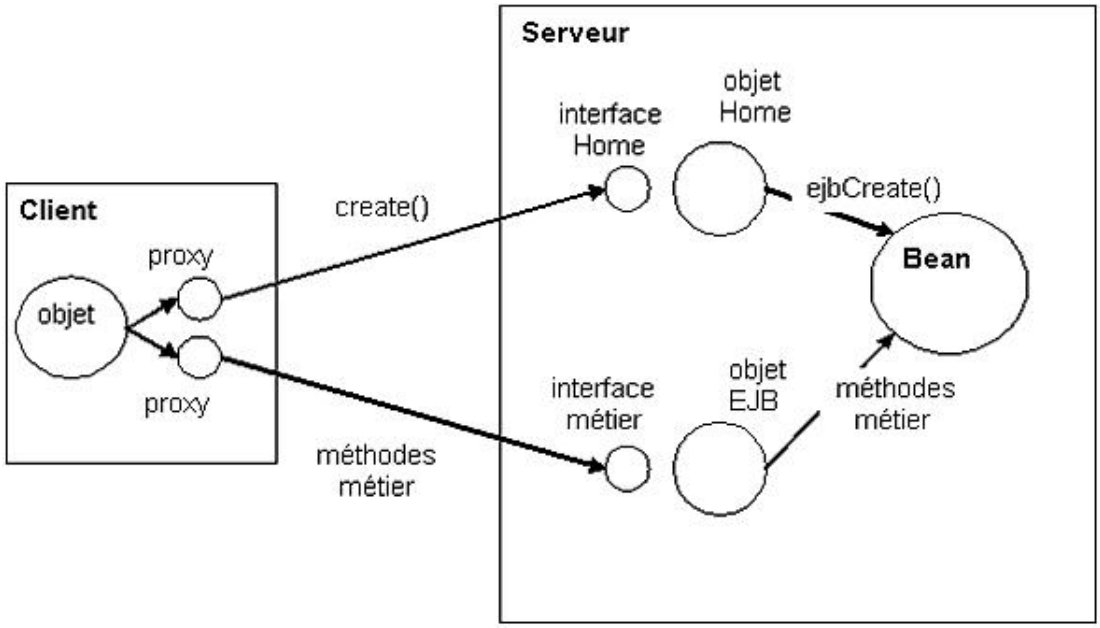
```
<invoker-proxy-binding>
  <name>stateless-unified-invoker</name>
  <invoker-mbean>jboss:service=invoker,type=unified</invoker-mbean>
  <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-factory>
  <proxy-factory-config>
    <client-interceptors>
      <home>
        <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
        <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
        <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
        <interceptor call-by-value=
"false">org.jboss.invocation.InvokerInterceptor</interceptor>
        <interceptor call-by-value=
"true">org.jboss.invocation.MarshallingInvokerInterceptor</interceptor>
      </home>
      <bean>
        <interceptor>org.jboss.proxy.ejb.StatelessSessionInterceptor
</interceptor>
        <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
        <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
        <interceptor call-by-value="false">org.jboss.invocation.
InvokerInterceptor</interceptor>
        <interceptor call-by-value="true">org.jboss.invocation.
MarshallingInvokerInterceptor</interceptor>
      </bean>
    </client-interceptors>
  </proxy-factory-config>
</invoker-proxy-binding>
```

Le nom de la classe de fabrique des proxies se retrouve dans la balise `<proxy-factory>`. Puis la description de la chaîne d'interception pour les interfaces de gestion du cycle de vie et d'exposition des méthodes métier de l'EJB se retrouve dans les balises `<home>` et `<bean>`.

Nous voyons que nous avons des intercepteurs pour :

- les interfaces : `org.jboss.proxy.ejb.HomeInterceptor` et `org.jboss.proxy.ejb.StatelessSessionInterceptor` ;

- le contexte sécurité : `org.jboss.proxy.SecurityInterceptor` ;
- les transactions : `org.jboss.proxy.TransactionInterceptor` ;
- le mode de transport de la requête du client : .
  - par référence, c'est-à-dire dans la même machine virtuelle, avec `org.jboss.invocation.InvokerInterceptor` ;
  - par valeur, c'est-à-dire par RMI, avec `org.jboss.invocation.MarshallingInvokerInterceptor`.



Côté serveur

L'invocation d'un EJB se termine côté serveur. L'invocation d'une méthode de l'EJB va être transportée vers la machine virtuelle de JBoss, via le bus JMX.

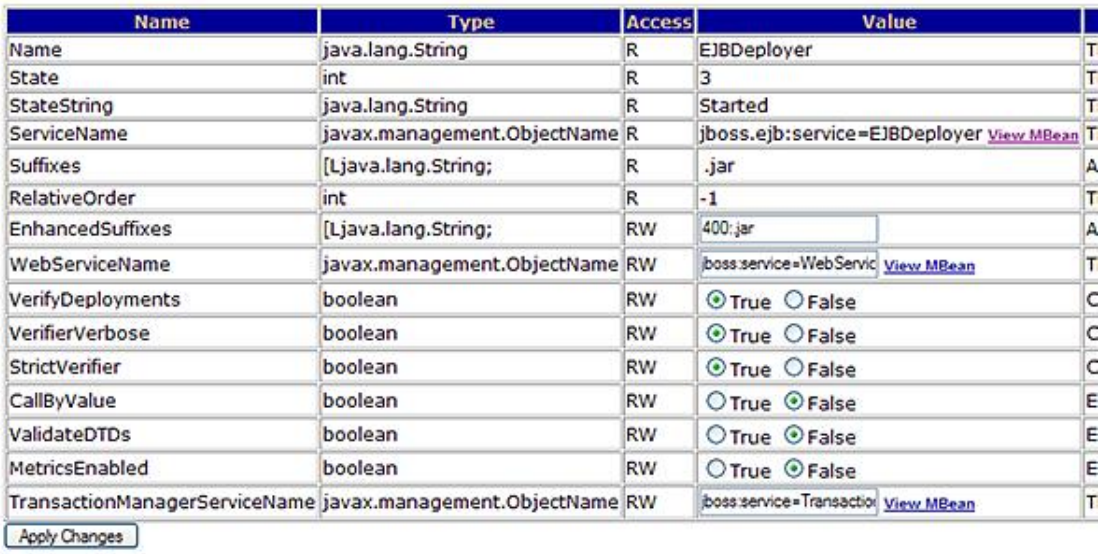
Chaque proxy est associé à une classe d'invocation et à un protocole de transport.

Le service `EJBDeployer` est responsable de la création des conteneurs EJB.

Nous pouvons retrouver ce service dans la console de visualisation des services, à l'adresse : http://localhost:8080/jmx-console/ puis, en recherchant le lien service=EJBDeployer sous la rubrique jboss.ejb.

Vous pouvez alors voir les attributs du MBean sur l'écran suivant :

List of MBean attributes:



Name	Type	Access	Value
Name	java.lang.String	R	EJBDeployer
State	int	R	3
StateString	java.lang.String	R	Started
ServiceName	javax.management.ObjectName	R	jboss.ejb:service=EJBDeployer View MBean
Suffixes	[Ljava.lang.String;	R	.jar
RelativeOrder	int	R	-1
EnhancedSuffixes	[Ljava.lang.String;	RW	400:.jar
WebServiceName	javax.management.ObjectName	RW	jboss:service=WebServic View MBean
VerifyDeployments	boolean	RW	True False
VerifierVerbose	boolean	RW	True False
StrictVerifier	boolean	RW	True False
CallByValue	boolean	RW	True False
ValidateDTDs	boolean	RW	True False
MetricsEnabled	boolean	RW	True False
TransactionManagerServiceName	javax.management.ObjectName	RW	jboss:service=Transactio View MBean

Apply Changes

Parmi les attributs modifiables :

- **`VerifyDeployments`** : vérifie que l'EJB est conforme aux spécifications EJB2.1. Il est très utile pour s'assurer de la validité du déploiement. Le résultat de la vérification n'affecte pas le déploiement lui-même, voir `StrictVerifier`.
- **`VerifierVerbose`** : passe la vérification du déploiement en mode "bavard".
- **`StrictVerifier`** : si cet attribut est positionné à "true", le déploiement n'est pas effectué si la vérification a échoué.
- **`CallByValue`** : mise en place par défaut, des invocations avec passage des paramètres par valeur.
- **`ValidateDTDs`** : vérifie que les fichiers XML ejb-jar.xml et jboss.xml sont valides par rapport à leur DTD.

Déploiement des EJB 2

Le composant EJB2 est packagé dans un module EJB (archive en .jar). Ce module peut être déployé directement, ou être lui-même inclus dans une application d'entreprise (archive en .ear).

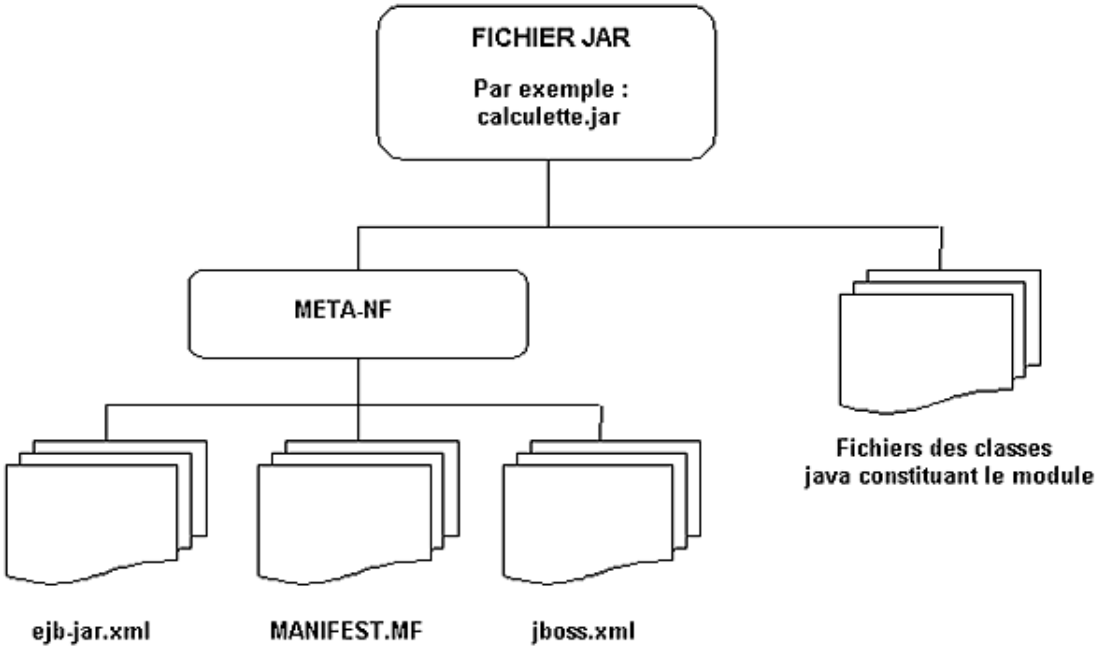
1. Packaging

a. Packaging en module EJB

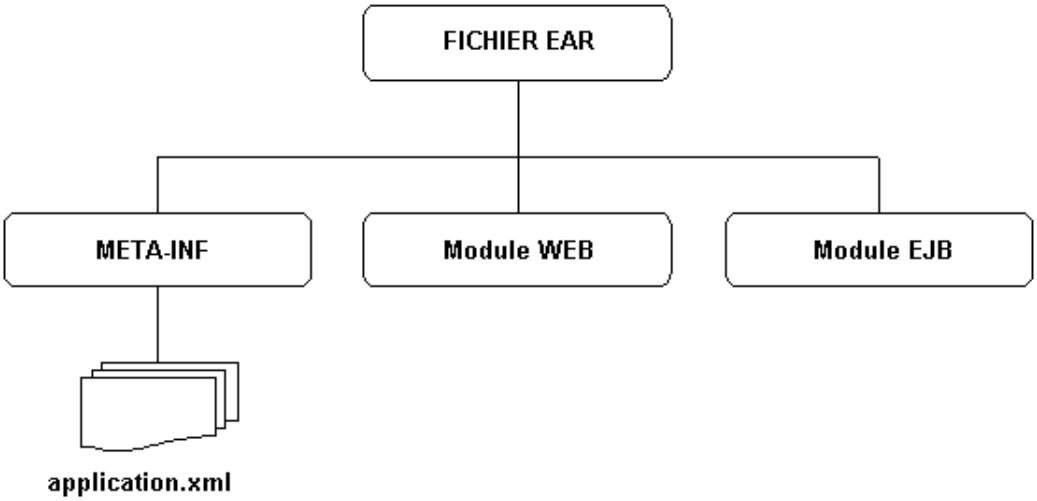
Le répertoire META-INF comporte :

- le fichier de déploiement standard de l'EJB : *ejb-jar.xml* ;
- le fichier de déploiement spécifique pour JBoss : *jboss.xml*.

Le fichier de déploiement standard sera portable d'un serveur à l'autre, tandis que le fichier de déploiement spécifique est propre à JBoss. Ce fichier spécifique sera donc à réécrire en cas de déploiement sous un autre serveur d'application. Heureusement, ces fichiers ont une syntaxe très proche les uns des autres.



b. Packaging dans une application d'entreprise



Le module application d'entreprise (fichier ear) peut regrouper :

- des modules Web (fichiers war) ;
- des modules EJB (fichiers jar) ;
- des librairies java (fichiers jar) ;
- un répertoire *META-INF* contenant le fichier *application.xml* qui permet de déclarer les modules et de surcharger les déploiements par défaut.

c. Déploiement

Le déploiement est effectué très simplement en plaçant le module EJB, ou l'application d'entreprise, dans le répertoire *deploy* du serveur. Par exemple, si vous travaillez sur le serveur par défaut, le répertoire de déploiement sera : *<répertoire installation JBoss>/server/default/deploy*.

Ce répertoire est normalement le répertoire de déploiement à chaud par défaut. Il est géré par le MBean `org.jboss.deployment.scanner.URLDeploymentScanner`. Nous avons vu au chapitre Architecture de JBoss comment nous pouvions modifier le répertoire de déploiement.

2. Descripteurs de déploiements

Les descripteurs de déploiement sont au format XML.

Le répertoire *<répertoire installation JBoss>/docs/dtd* contient l'ensemble des fichiers DTD *(Document Type Definition)* utilisés par JBoss.

Il y a deux types de descripteurs de déploiement :

- les standards qui sont conformes aux spécifications de Sun ;
- les spécifiques qui sont propres au serveur d'application.

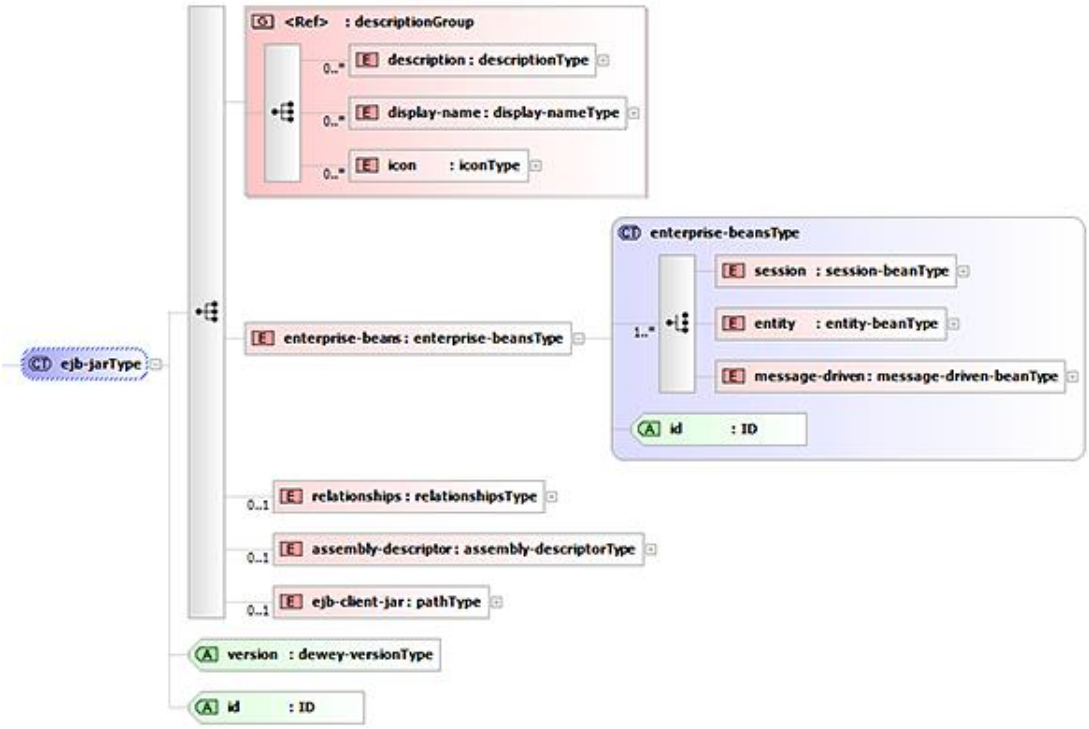
Par exemple, les spécifications J2EE expliquent comment faire référence et utiliser une source de données, une authentification dans un module, mais ne spécifient pas comment les mettre en place et comment les lier avec le module. Cette mise en place de la source de données et la liaison avec le module sont propres à chaque serveur.

a. Descripteur de déploiement standard ejb-jar.xml

Vous pouvez télécharger le schéma XML correspondant au fichier *ejb-jar.xml* à l'URL suivante : http://java.sun.com/xml/ns/j2ee/#resources

Ce fichier XML est portable d'un serveur à l'autre. Il permet de constituer le composant, en y décrivant :

- les classes interface ;
- la classe d'implémentation ;
- le type d'EJB ;
- les configurations de gestion de l'EJB ;
- la déclaration des champs persistants pour les EJB entités.



Nous ne reprendrons ici que les éléments principaux. Les exemples de déploiement qui suivent dans ce chapitre vont permettre d'illustrer ce fichier *ejb-jar.xml*.

- **<ejb-jar>** : racine du document XML ;
- **<enterprise-beans>** : contient les éléments fils `<session>`, `<entity>`, `<message-driven>` correspondant aux différents EJB du module ;
- **<session>** : déclaration d'un EJB de type session ;
- **<entity>** : déclaration d'un EJB de type entité, pour la persistance ;
- **<message-driven>** : déclaration d'un EJB de type orienté message ;
- **<ejb-name>** : nom de l'EJB, c'est ce nom qui sera utilisé dans le fichier jboss.xml ;
- **<home>** : nom de l'interface qui étend l'interface `javax.ejb.EJBHome`, celle-ci présentant les méthodes du cycle de vie utilisables à distance par RMI ;
- **<local-home>** : nom de l'interface qui étend l'interface `javax.ejb.EJBLocalHome`, présentant les méthodes du cycle de vie utilisables en local, c'est-à-dire dans la même machine virtuelle ;
- **<remote>** : nom de l'interface qui étend l'interface `javax.ejb.EJBObject`, présentant les méthodes métier utilisables à distance par RMI ;

- **<local>** : nom de l'interface qui étend l'interface `javax.ejb.EJBLocalObject`, présentant les méthodes métier utilisables en local, c'est-à-dire dans la même machine virtuelle ;

- **<ejb-class>** : nom de la classe d'implémentation de l'EJB, qui étend une des classes `javax.ejb.SessionBean`, `javax.ejb.EntityBean` ou `javax.ejb.MessageDrivenBean` en fonction du type d'EJB ;

- **<relationships>** : décrit les relations entre les EJB entités.

b. Descripteur de déploiement spécifique jboss.xml

Le fichier *jboss.xml* est le fichier de déploiement spécifique à JBoss. En fonction de la version de JBoss que vous utilisez, vous pouvez référencer des DTD différentes.

```
<!DOCTYPE jboss
    PUBLIC "-//JBoss//DTD JBOSS 4.0//EN"
    "http://www.jboss.org/j2ee/dtd/jboss_4_0.dtd">

<jboss>
    ...
</jboss>
```

Vous retrouverez l'ensemble des DTDs disponibles dans le répertoire *<répertoire installation JBoss>/docs/dtd*.

Chaque fichier *jboss.dtd* commence par un commentaire présentant la structure du fichier *jboss.xml*.

Pour le fichier *jboss.dtd*, nous avons la structure suivante :

```
<jboss>

  <secure />
  <security-domain />

  <enterprise-beans>

    <entity>
      <ejb-name />
      <jndi-name />
      <resource-ref>
        <res-ref-name />
        <resource-name />
      </resource-ref>
    </entity>

    <session>
      <ejb-name />
      <jndi-name />
      <resource-ref>
        <res-ref-name />
        <resource-name />
      </resource-ref>
    </session>

  </enterprise-beans>

  <resource-managers>

    <resource-manager>
      <res-name />
      <res-jndi-name />
    </resource-manager>

    <resource-manager>
      <res-name />
      <res-url />
    </resource-manager>
```

```
  </resource-managers>

  <container-configurations>

    <container-configuration>
      <container-name />
      <container-invoker />
      <container-interceptors />
      <instance-pool />
      <instance-cache />
      <persistence-manager />
      <transaction-manager />
      <container-invoker-conf />
      <container-cache-conf />
      <container-pool-conf />
      <commit-option />
      <role-mapping-manager/>
      <authentication-module/>
    </container-configuration>

  </container-configurations>

</jboss>
```

Cette structure de base s'est enrichie au fur et à mesure de l'évolution des versions de JBoss.

Les exemples de déploiement exposés dans ce chapitre permettront d'illustrer l'utilisation de ce fichier.

Dans le fichier *jboss.xml*, nous allons pouvoir, entre autres, y préciser :

- les nommages JNDI des EJB ;
- les références à des ressources, comme les sources de données...

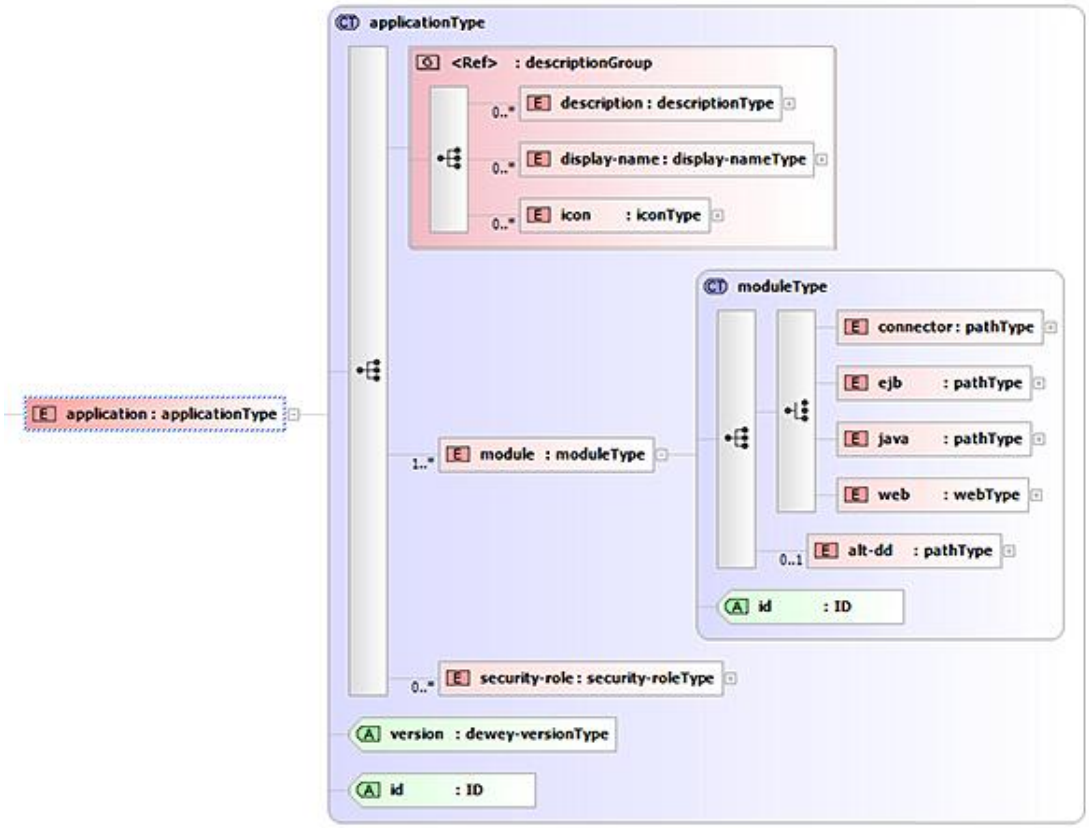
Seuls les éléments les plus utilisés seront ici décrits. Certains éléments, comme l'élément `<secure>` ont disparu avec les versions supérieures des DTD. Il convient donc toujours de bien vérifier les versions qui sont utilisées.

- **`<enterprise-beans>`** encapsule les balises `<session>`, `<entity>` et `<message-driven>`. L'ensemble de ces balises correspond aux balises de même nom dans le fichier *ejb-jar.xml* en y ajoutant des paramètres supplémentaires spécifiques à JBoss. Nous avons donc en général, autant d'entrées d'EJB de type session entité ou orienté message dans les deux fichiers *ejb-jar.xml* et *jboss.xml*.
- **`<ejb-name>`** est un élément fils des éléments `<session>`, `<entity>` et `<message-driven>`. Il contient le nom de l'EJB qui doit être le même que dans le fichier *ejb-jar.xml*.
- **`<jndi-name>`** est un élément fils des éléments `<session>`, `<entity>`. Il contient le nom de JNDI de l'interface Home distante de l'EJB.
- **`<local-jndi-name>`** est un élément fils des éléments `<session>`, `<entity>`. Il contient le nom de JNDI de l'interface Home locale de l'EJB.
- **`<ejb-ref>`** est un élément fils des éléments `<session>`, `<entity>` et `<message-driven>`. Il contient la référence vers un autre EJB distant.

c. Descripteur de déploiement application.xml

Le plus simple pour déployer un ensemble de composants sur un même serveur est de regrouper l'ensemble de ces composants dans une même archive. Cette archive est un fichier dont l'extension est *ear*, pour enterprise archive.

Cette archive comporte dans son répertoire META-INF, un descripteur de déploiement, le fichier application.xml, dont la racine est <application>.



Le fichier XML suivant présente un exemple typique de ce fichier. Ce fichier est extrait du projet *BovoyageCMP_EAR* que nous retrouverons plus loin dans ce chapitre.

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:application="http://java.sun.com/xml/ns/javaee/application_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/application_5.xsd
id="Application_ID" version="5">

  <module>
    <ejb>Bovoyage_CMP.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>BovoyageWeb_CMP.war</web-uri>
      <context-root>bovoyage-cmp</context-root>
    </web>
  </module>
</application>
```

Nous y retrouvons la déclaration d'un module EJB :

```
  <module>
    <ejb>Bovoyage_CMP.jar</ejb>
  </module>
```

et d'un module Web auquel nous avons défini un nom d'application :

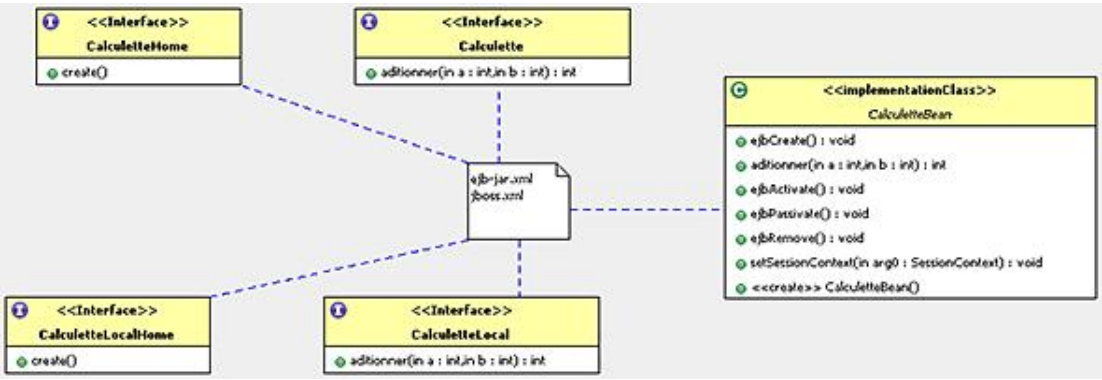
```
  <module>
    <web>
      <web-uri>BovoyageWeb_CMP.war</web-uri>
      <context-root>bovoyage-cmp</context-root>
    </web>
  </module>
```

3. Outils d'aide au codage des EJB 2

Les composants EJB 2 peuvent vite devenir complexes à écrire.

Ainsi, pour créer une EJB2 de type session, le développeur doit créer les 5 classes nécessaires à notre EJB, et les fichiers XML :

- l'interface qui étend `javax.ejb.EJBHome` ;
- l'interface qui étend `javax.ejb.EJBlocalHome` ;
- l'interface qui étend `javax.ejb.EJBObject` ;
- l'interface qui étend `javax.ejb.EJBLocalObject` ;
- la classe d'implémentation qui étend `javax.ejb.SessionBean` ;
- le fichier *ejb-jar.xml* ;
- le fichier *jboss.xml*.



XDoclet est une bibliothèque open-source qui via, des balises de documentation spécifiques, générera les fichiers java et xml adaptés à la cible choisie. Dans notre cas, cette cible sera de l'EJB 2 avec comme serveur de déploiement JBoss.

Vous trouverez dans le chapitre Produits supplémentaires une présentation succincte de l'utilisation de XDoclet.

XDoclet allège le processus de création. Le développeur n'a qu'à créer la classe d'implémentation et paramétrer les balises XDoclet, l'outil générera l'ensemble des fichiers nécessaires, ainsi que des classes d'aide (helper) :

- une classe `xxxUtil`, dans notre cas `CalculetteUtil`, qui possède un certain nombre de méthodes permettant d'encapsuler la recherche des interfaces Home et LocalHome via JNDI ;
- une classe `xxxSession` , dans notre cas `CalculetteSession`, qui hérite de notre classe d'implémentation et de `javax.ejb.SessionBean`. C'est cette classe qui sera référencée dans le fichier *ejb-jar.xml*. Pour les autres types d'EJB, le nom de la classe est différent : xxxCMP pour les entités CMP par exemple ;
- une classe `xxxData` de type POJO *(Plain Old Java Object)*, pour les EJB de type entité, qui pourra être utilisée avec des objets de transfert.

4. EJB2 de session

Les exemples qui illustrent les EJB2 sont disponibles dans l'archive téléchargeable sur le site ENI. Ces exemples sont une vue simplifiée d'une application d'entreprise. Entre autres, un certain nombre de modèles de conception, que nous utilisons dans des réalisations réelles, ne sont pas présents ici pour ne pas obscurcir la lisibilité du code : pas d'objets de transferts (TO pour *Transfer Object*), une couche DAO simplifiée, des façades simplifiées, absence de classe de localisation (Locator).

De même, la partie Web n'utilise pas de framework comme Struts ou autre, mais en garde l'esprit en utilisant un contrôleur.

Cet exemple simule le choix d'un voyage sur une agence en ligne.

Les EJB de cet exemple ont été générés avec XDoclet. Seules les classes EJB se terminant par Bean (`CaddieBean`, `DestinationFacadeBean`...) ont été écrites. Les interfaces et classes helpers, ainsi que les fichiers de déploiement `ejb-jar.xml` et `jboss.xml`, ont été générés par un script Ant utilisant XDoclet. L'utilisation de XDoclet dans Eclipse est automatisée, le chapitre Produits supplémentaires abordera ce sujet.

Cet ouvrage n'est pas centré sur les EJB 2 ou EJB 3, il n'y aura donc pas une présentation en profondeur de ces frameworks. Les EJB 2 sont présentés avec les fichiers xml minimum pour que le lecteur qui ne connaît pas cette technologie, ne soit pas perdu s'il la rencontre. Les EJB 3 ont supplanté les EJB 2 dans les nouveaux projets. Mais il n'est pas rare qu'un développeur, administrateur ou chef de projet soit amené à intervenir ou à déployer des EJB 2 sur des projets plus anciens, c'est la raison pour laquelle ils sont présentés dans cet ouvrage.

Les EJB de session sont des composants qui vont être utilisés par une application cliente, distante ou non, mais dont les états ne seront pas persistés en base de données. Tandis que par définition, les EJB entités sont persistants, et donc leur état est sauvegardé en base de données.

Les EJB de session sont utilisés pour effectuer des traitements liés à des requêtes du client, ou gérer des objets de session.

Les EJB de session peuvent être sans état (`stateless`), ou avec état (`statefull`).

L'EJB sans état ne conserve pas son état, d'un appel à l'autre d'un client, alors que l'EJB avec état, le conserve. Une instance d'EJB sans état peut être utilisée par plusieurs clients, alors que l'instance d'une EJB avec état est utilisée par un seul client.

Il faut noter que ce n'est pas parce que le client demande la création d'un EJB par la méthode `create(...)` sur son interface `Home`, qu'il y a instanciation d'un EJB. Le serveur peut gérer des pools d'instances et recycler les instances qui ne sont pas utilisées. De même, le serveur peut être amené à sérialiser un EJB de session, et donc à le désérialiser. Attention, ce n'est pas parce qu'il y a sérialisation, qu'il y a persistance. La sérialisation est un mécanisme de gestion des instances interne au serveur. La persistance, quant à elle, permet de sauvegarder l'état d'un objet en base de données.

Les méthodes du cycle de vie de l'EJB, qui sont exposées par le biais des interfaces `Home`, sont appelées par le serveur.

De même les méthodes métier ne sont jamais directement invoquées par le client, elles le sont par le serveur si toutes les conditions de transactions, d'autorisations sont satisfaites.

Quelques exemples d'EJB stateless :

- un calcul d'intérêt ;
- la récupération du code postal d'une ville.

Quelques exemples d'EJB statefull :

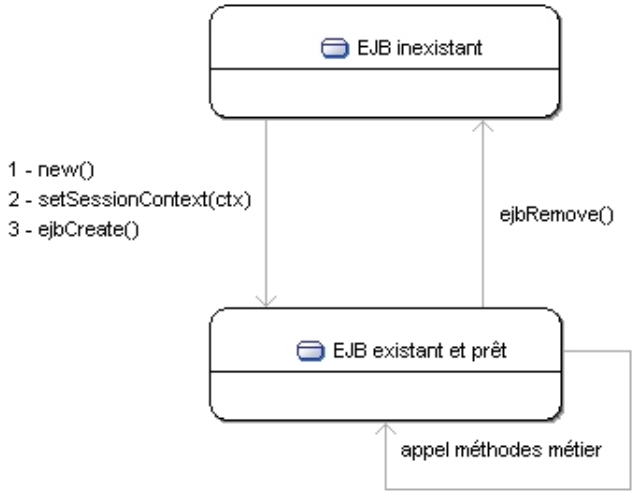
- un panier de commande ;
- un suivi de paiement.

a. stateless

Comme vu précédemment, un EJB sans état n'est pas associé à un client donné. Une même instance d'EJB stateless peut être utilisée par plusieurs requêtes de clients. Une seule requête est exécutée à la fois, les EJB n'étant pas multithreadés.

Cycle de vie

Le cycle de vie d'un EJB sans état est très simple. Son instance est existante ou pas.



Comme illustration d'un EJB sans état, nous prendrons l'EJB `DestinationFacadeBean` du projet `Bovoyage_CMP`. Cet EJB permet à une application cliente de récupérer un certain nombre de données sur les destinations. Cet EJB rend des services à l'application cliente, mais ne conserve aucun état entre deux requêtes du client. Il s'agit donc bien d'un EJB sans état.

Méthodes du cycle de vie

La méthode `ejbCreate()` est codée dans la classe héritant de `javax.ejb.SessionBean`.

Remarquez que cette méthode renvoie `void`.

```
public abstract class DestinationFacadeBean implements javax.ejb.SessionBean
{
...
    public void ejbCreate(){}
...
}
```

La méthode `create()` est exposée dans la classe de type `javax.ejb.EJBHome`. Cette méthode renvoie un proxy sur l'interface distante `javax.ejb.EJBObject`, qui implémente les méthodes métier accessibles via RMI.

```
public interface DestinationFacadeHome
   extends javax.ejb.EJBHome
{
   public org.bovoyage.ejb2.DestinationFacade create()
      throws javax.ejb.CreateException,java.rmi.RemoteException;

}
```

La méthode `create()` est aussi exposée dans la classe de type `javax.ejb.EJBLocalHome`. Cette méthode renvoie un proxy sur l'interface locale `javax.ejb.EJBLocalObject`, qui implémente les méthodes métier accessibles au sein de la même machine virtuelle.

```
public interface DestinationFacadeLocalHome
   extends javax.ejb.EJBLocalHome
{
   public DestinationFacadeLocal create()
      throws javax.ejb.CreateException;

}
```

Notez que les exceptions susceptibles d'être levées sont différentes : l'exception `java.rmi.RemoteException` n'est levée que s'il s'agit de méthodes distantes.

Méthodes métier

Les méthodes métier sont exposées par les interfaces de type `javax.ejb.EJBObject` pour les méthodes accessibles via RMI, et `javax.ejb.EJBLocalObject` pour les méthodes accessibles localement, au sein de la même machine virtuelle. Ces méthodes sont implémentées dans la classe héritant de `javax.ejb.SessionBean`.

Extraits des classes :

```
public interface DestinationFacade
   extends javax.ejb.EJBObject
{
   public java.util.List getDestinations(  )
      throws java.rmi.RemoteException;

   public java.util.List getSejours(String idDestination )
      throws java.rmi.RemoteException;

...

}
```

```
public interface DestinationFacadeLocal
   extends javax.ejb.EJBLocalObject
{
   public java.util.List getDestinations(  ) ;

   public java.util.List getSejours(String idDestination ) ;

...

}
```

```
public abstract class DestinationFacadeBean implements javax.ejb.SessionBean
{

...
public List getDestinations()
{
ArrayList destinations = new ArrayList();
try
{
    DestinationLocalHome home = DestinationUtil.getLocalHome();
    Collection liste = home.findAll();
    Iterator it = liste.iterator();
    while (it.hasNext())
    {
        DestinationLocal local = (DestinationLocal) it.next();
        DestinationData data = new DestinationData(local.getId(),
                                        local.getPays(),
                                        local.getDescription());
        destinations.add(data);
    }
}
catch (Exception e)
{
        // TODO Auto-generated catch block
    e.printStackTrace();
}

return destinations;
}

...

}
```

Descripteur ejb-jar.xml

Pour cet EJB stateless, le descripteur de déploiement standard `ejb-jar.xml` contient les lignes suivantes :

```
<session id="Session_DestinationFacade">
    <description><![CDATA[An EJB named DestinationFacade]]></description>
    <display-name>DestinationFacade</display-name>
```

```
    <ejb-name>DestinationFacade</ejb-name>

    <home>org.bovoyage.ejb2.DestinationFacadeHome</home>
    <remote>org.bovoyage.ejb2.DestinationFacade</remote>
    <local-home>
      org.bovoyage.ejb2.DestinationFacadeLocalHome
    </local-home>
    <local>org.bovoyage.ejb2.DestinationFacadeLocal</local>
    <ejb-class>
      org.bovoyage.ejb2.DestinationFacadeSession
    </ejb-class>

    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>

</session>
```

L'élément `<session>` qui indique que l'EJB décrit est de type session contient ici les éléments suivants :

- `<description>` et `<display-name>` sont utilisés par certains outils de gestion d'EJB au sein d'un serveur. Ils peuvent être omis.
- `<ejb-name>` contient le nom de l'EJB. C'est ce nom qui sera repris dans le fichier *jboss.xml*.
- `<home>` contient le nom de l'interface qui étend `javax.ejb.EJBHome`, exposant les méthodes distantes du cycle de vie.
- `<remote>` contient le nom de l'interface qui étend `javax.ejb.EJBObject`, exposant les méthodes métier distantes.
- `<local-home>` contient le nom de l'interface qui étend `javax.ejb.EJBLocalHome`, exposant les méthodes locales du cycle de vie.
- `<local>` contient le nom de l'interface qui étend `javax.ejb.EJBLocalObject`, exposant les méthodes métier locales.
- `<ejb-class>` contient le nom de la classe d'implémentation des méthodes du cycle de vie et des méthodes métier. Cette classe implémente l'interface `javax.ejb.SessionBean`.
- `<session-type>` indique le type d'EJB de session, ici un EJB sans état.
- `<transaction-type>` indique ici que les transactions sont gérées par le conteneur.

Descripteur jboss.xml

Pour cet EJB, le descripteur de déploiement spécifique *jboss.xml* contient les lignes suivantes :

```
<session>
   <ejb-name>DestinationFacade</ejb-name>
   <jndi-name>ejb2/DestinationFacade</jndi-name>
   <local-jndi-name>ejb2/local/DestinationFacade</local-jndi-name>
</session>
```

L'élément `<session>` du fichier *jboss.xml* est le pendant du même élément du fichier *ejb-jar.xml*. Il permet de préciser le déploiement dans le serveur JBoss. Nous y trouvons ici :

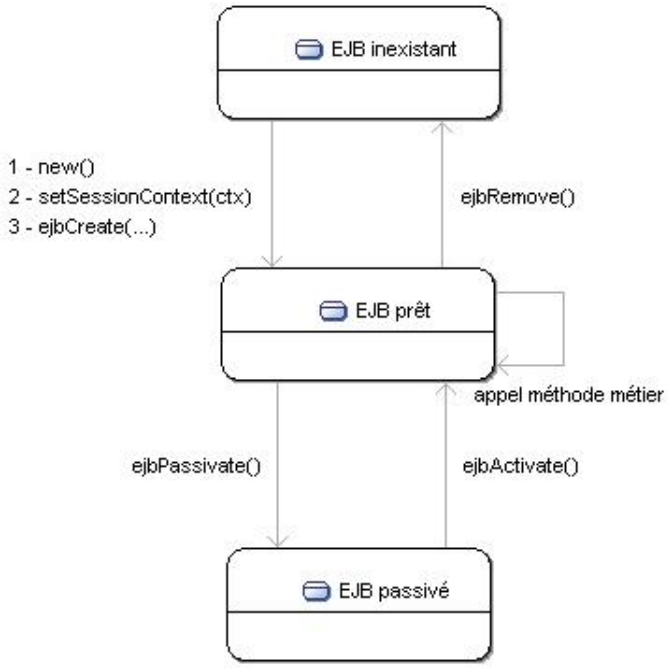
- `<ejb-name>` qui contient le nom de l'EJB tel qu'il a été défini dans la balise `<ejb-name>` du fichier *ejb-jar.xml*.
- `<jndi-name>` contient le nom distant de l'EJB, c'est ce nom qui sera utilisé par les clients distants, via RMI. L'EJB est mis dans le contexte JNDI le plus élevé du serveur pour pouvoir être accessible en dehors du contexte serveur.

- `<local-jndi-name>` contient le nom local de l'EJB. C'est ce nom qui sera utilisé par les clients qui sont dans la même machine virtuelle que l'EJB, donc au sein du même serveur. Ce nom fait aussi parti du contexte JNDI le plus élevé, mais il ne peut pas être utilisé par un client distant.

b. statefull

Cycle de vie

L'instance de l'EJB statefull est liée au client qui l'utilise, l'état de cet EJB peut donc être sérialisé. Ainsi, le cycle de vie de l'EJB est un peu plus complexe, puisque l'instance peut exister mais être "passiver".



Comme illustration d'un EJB avec état, nous prendrons l'EJB `CaddieBean` du projet `Bovoyage_EJB2`. Cet EJB reflète le contenu du panier d'achat d'un utilisateur, c'est donc bien un EJB statefull.

Méthodes du cycle de vie

Ce type d'EJB peut posséder plusieurs méthodes de création. Il faut veiller à ce qu'il y ait autant de méthodes `createXxxx(...)` et `create(...)` exposées par les interface `Home` et `LocalHome`, que de méthodes `ejbCreateXxxx(...)` et `ejbCreate(...)` dans la classe d'implémentation.

```
public abstract class CaddieBean implements javax.ejb.SessionBean
{
    private SessionContext ctx;
    private List sejours = new ArrayList();
...

    public void ejbCreateAvecSejour(SejourData sejour)
    {    this.add(sejour);}

    public void ejbCreate(SejourData sejour)
    {    this.add(sejour);}

    public void ejbCreate(){}

    public void setSessionContext(SessionContext arg0) throws
EJBException, RemoteException
    {
        this.ctx = arg0;
    }

...
}
```

Notez que la classe d'implémentation met à jour sa propriété `ctx` de type `javax.ejb.SessionContext`, lors de l'invocation par le conteneur de la méthode `setSessionContext(...)`.

Voici les signatures dans l'interface `javax.ejb.EJBLocalHome`, notez qu'elles ne sont pas susceptibles de générer une exception de type `java.rmi.RemoteException`, puisque les valeurs sont transmises par référence.

```
public interface CaddieLocalHome
   extends javax.ejb.EJBLocalHome
{

   public CaddieLocal createAvecSejour(SejourData sejour)
      throws javax.ejb.CreateException;

   public CaddieLocal create(SejourData sejour)
      throws javax.ejb.CreateException;

   public CaddieLocal create()
      throws javax.ejb.CreateException;

}
```

Voici les signatures dans l'interface `javax.ejb.EJBHome` :

```
public interface CaddieHome
   extends javax.ejb.EJBHome
{
   public Caddie createAvecSejour(SejourData sejour)
      throws javax.ejb.CreateException,java.rmi.RemoteException;

   public Caddie create(SejourData sejour)
      throws javax.ejb.CreateException,java.rmi.RemoteException;

   public Caddie create()
      throws javax.ejb.CreateException,java.rmi.RemoteException;

}
```

Méthodes métier

Les méthodes métier sont exposées par les interfaces de type `javax.ejb.EJBObject` pour les méthodes accessibles via RMI, et `javax.ejb.EJBLocalObject` pour les méthodes accessibles localement, au sein de la même machine virtuelle. Ces méthodes sont implémentées dans la classe héritant de `javax.ejb.SessionBean`.

Extraits des classes :

```
public interface Caddie
   extends javax.ejb.EJBObject
{
   public void add(SejourData sejour )
      throws java.rmi.RemoteException;

   public void remove(SejourData sejour )
      throws java.rmi.RemoteException;

   public double getPrix(  )
      throws java.rmi.RemoteException;

...

}
```

```
public interface CaddieLocal
   extends javax.ejb.EJBLocalObject
{
   public void add(SejourData sejour ) ;
```

```
   public void remove(SejourData sejour ) ;

   public double getPrix(  ) ;

...

}
```

```
public abstract class CaddieBean implements javax.ejb.SessionBean
{
    private SessionContext ctx;
    private List sejours = new ArrayList();


    public void add(SejourData sejour)
    {
        if(!sejours.contains(sejour))
            sejours.add(sejour);
    }



    public void remove(SejourData sejour)
    {
        if(sejours.contains(sejour))
            sejours.remove(sejour);
    }


    public double getPrix()
    {
        double prix=0;
        Iterator it = sejours.iterator();
        while(it.hasNext())
       {
            SejourData s = (SejourData)it.next();
            prix += s.getPrix().doubleValue();
        }
        return prix;
    }

...
}
```

Descripteur ejb-jar.xml

Pour cet EJB statefull, le descripteur de déploiement standard `ejb-jar.xml` contient les lignes suivantes :

```
<session id="Session_Caddie">
   <description><![CDATA[An EJB named Caddie]]></description>
   <display-name>Caddie</display-name>

   <ejb-name>Caddie</ejb-name>

   <home>org.bovoyage.ejb2.CaddieHome</home>
   <remote>org.bovoyage.ejb2.Caddie</remote>
   <local-home>org.bovoyage.ejb2.CaddieLocalHome</local-home>
   <local>org.bovoyage.ejb2.CaddieLocal</local>
   <ejb-class>org.bovoyage.ejb2.CaddieSession</ejb-class>
   <session-type>Stateful</session-type>
   <transaction-type>Container</transaction-type>

</session>
```

L'élément `<session>` qui indique que l'EJB décrit est de type session contient ici les éléments suivants :

- `<description>` et `<display-name>` sont utilisés par certains outils de gestion de EJB au sein d'un serveur. Ils

peuvent être omis.

- <ejb-name> contient le nom de l'EJB. C'est ce nom qui sera repris dans le fichier *jboss.xml*.
- <home> contient le nom de l'interface qui étend `javax.ejb.EJBHome`, exposant les méthodes distantes du cycle de vie.
- <remote> contient le nom de l'interface qui étend `javax.ejb.EJBObject`, exposant les méthodes métier distantes.
- <local-home> contient le nom de l'interface qui étend `javax.ejb.EJBLocalHome`, exposant les méthodes locales du cycle de vie.
- <local> contient le nom de l'interface qui étend `javax.ejb.EJBLocalObject`, exposant les méthodes métier locales.
- <ejb-class> contient le nom de la classe d'implémentation des méthodes du cycle de vie et des méthodes métier. Cette classe implémente l'interface `javax.ejb.SessionBean`.
- <session-type> indique le type d'EJB de session, ici un EJB avec état. Attention à l'orthographe de Stateful qui ne prend ici qu'une lettre "l" finale.
- <transaction-type> indique ici que les transactions sont gérées par le conteneur.

Descripteur jboss.xml

Pour cet EJB, le descripteur de déploiement spécifique *jboss.xml* contient les lignes suivantes :

```
<session>
   <ejb-name>Caddie</ejb-name>
   <jndi-name>ejb2/Caddie</jndi-name>
   <local-jndi-name>ejb2/local/Caddie</local-jndi-name>
</session>
```

L'élément <session> du fichier *jboss.xml* est le pendant du même élément du fichier *ejb-jar.xml*. Il permet de préciser le déploiement dans le serveur JBoss. Nous y trouvons ici :

- <ejb-name> qui contient le nom de l'EJB tel qu'il a été défini dans la balise <ejb-name> du fichier *ejb-jar.xml*.
- <jndi-name> contient le nom distant de l'EJB, c'est ce nom qui sera utilisé par les clients distant, via RMI. L'EJB est mis dans le contexte JNDI le plus élevé du serveur pour pouvoir être accessible en dehors du contexte serveur.
- <local-jndi-name> contient le nom local de l'EJB. C'est ce nom qui sera utilisé par les clients qui sont dans la même machine virtuelle que l'EJB, donc au sein du même serveur. Ce nom fait aussi partie du contexte JNDI le plus élevé, mais il ne peut pas être utilisé par un client distant.

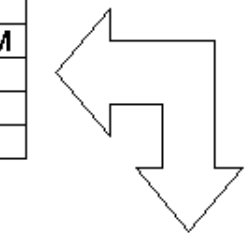
5. EJB2 entité

L'EJB entité représente un objet persistant dont l'état sera donc synchronisé avec un enregistrement en base de données. C'est un objet qui possède en plus des méthodes, des données qui correspondent à une réalité du métier. Cette réalité existe en dehors de tout traitement, par exemple un compte client, l'identité d'une personne, un voyage, etc.

Le conteneur d'EJB est chargé de retrouver un EJB entité en utilisant les données stockées en base. À chaque enregistrement en base correspond une instance d'EJB entité différente. Lorsque l'état de l'EJB entité change, le conteneur synchronise le contenu de la base de données.

Le conteneur doit connaître les associations entre l'entité et la base de données. Pour cela, le mapping décrit comment les propriétés de l'entité sont enregistrées dans une colonne d'une table de la base de données.

CONTACTS			
ID	CIVILITE	NOM	PRENOM
1	M	LEBLANC	Juste
42	Mme	BLANSEC	Adèle
89	Mlle	O'HARA	Scarlette



cl : Contact

id = 1
civilite = "M"
nom = "LEBLANC"
prenom = "JUSTE"

Il existe deux types d'EJB entité :

- les CMP pour *Container Managed Persistence*. Ce type d'EJB est complètement pris en charge par le conteneur d'EJB. C'est le conteneur qui assure la persistance des données, grâce aux descripteurs de déploiement.

- les BMP pour *Bean Managed Persistence*. C'est le bean qui assure la persistance des données. Le développeur écrit le code de persistance.

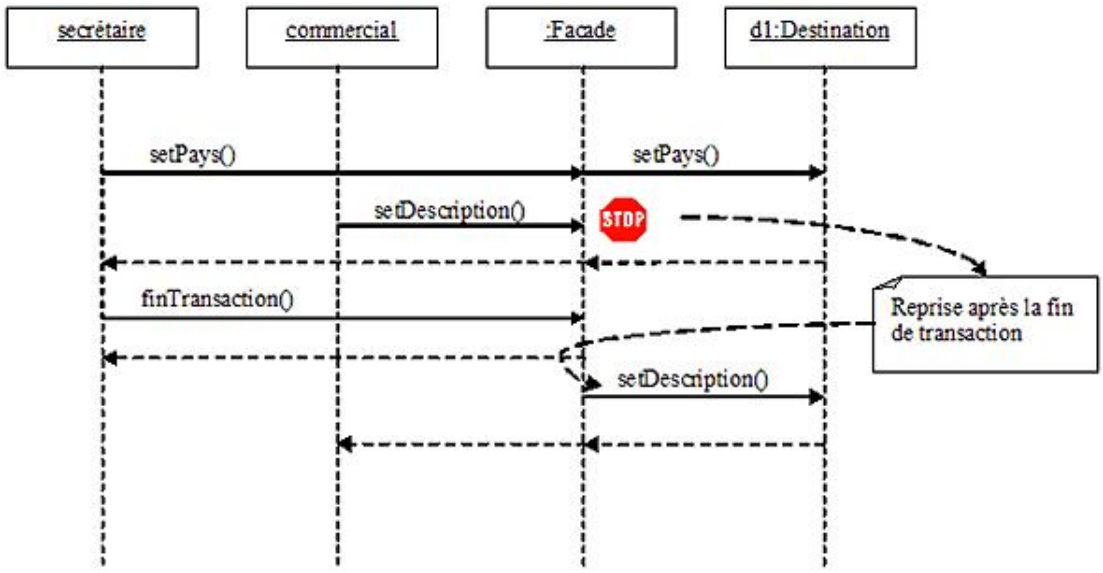
L'unicité d'un EJB entité est liée à la clé primaire qui est, par définition, unique. Cette clé est toujours la même dans tout le cycle de vie de l'EJB. Cette notion de clé primaire est primordiale car le conteneur ne connait que cette propriété (qui est immuable pour lui) pour identifier, de manière sûre, un EJB entité.

Pour que les états d'un EJB entité soient persistés, il faut que les propriétés liées aux états soient sérialisables.

Un EJB entité représentant un ensemble de données provenant du système d'information, il peut être sollicité simultanément par plusieurs clients. Dans notre exemple, une entité qui représente une destination précise peut être consultée par plusieurs utilisateurs. Les EJB entité sont donc soumis à une concurrence. C'est le conteneur qui gère cette concurrence.

Concurrence

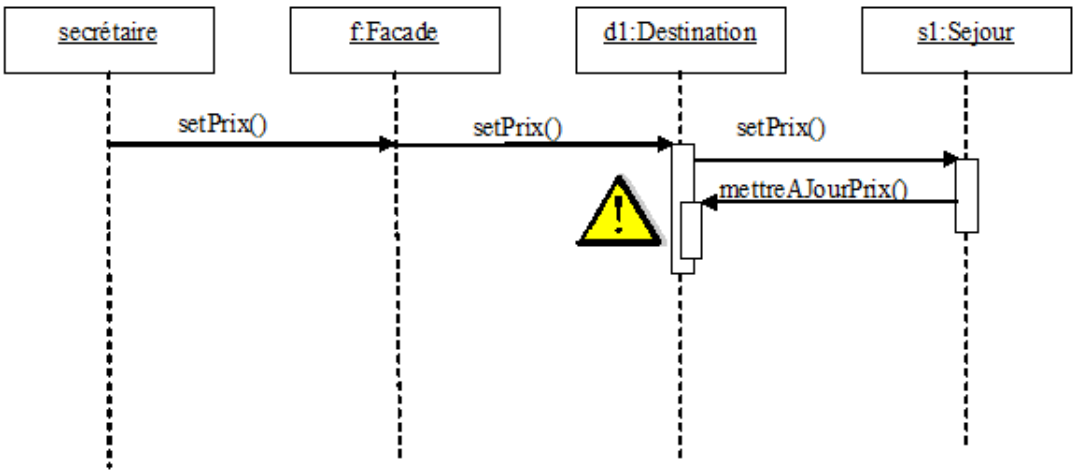
Cette gestion de la concurrence est importante lorsque deux utilisateurs veulent modifier le même EJB entité. Imaginons que sur une destination d1, la secrétaire veuille modifier le champ pays pour corriger une faute de frappe, et qu'au même instant, le responsable commercial modifie la description de la destination.



La concurrence apparaît si plusieurs utilisateurs utilisent une même entité simultanément. Elle n'apparaît pas si les utilisateurs utilisent le bean entité de manière successive et ceci même si les utilisateurs référencient le bean de manière continue.

Réentrance

La réentrance est le fait que, lors de l'exécution d'un code sollicitant des EJB entité ou session, il est possible d'y retrouver un même EJB deux fois. Le bean est sollicité une première fois par un appel de méthode, puis du fait des appels successifs de méthodes, il sera sollicité une seconde fois. Si vous entrez dans le bean une seconde fois, il y a réentrance.



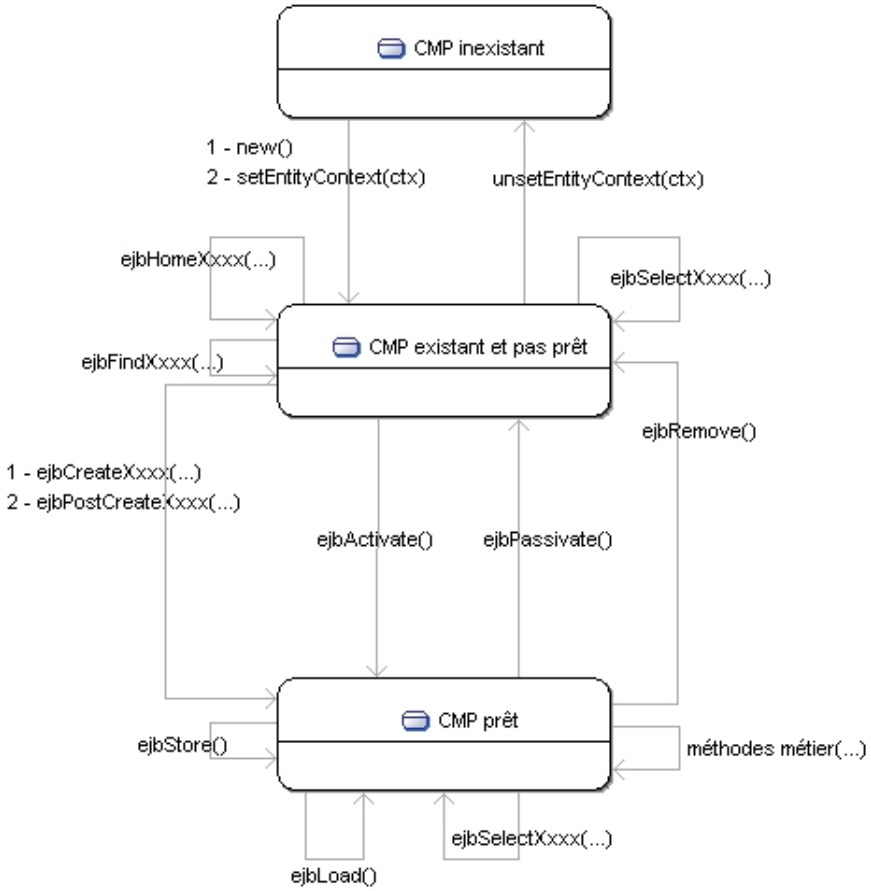
Ici, la méthode `setPrix()` sur `s1` invoque la méthode `mettreAJourPrix()` sur `d1`, alors que la méthode `setPrix()` de `d1` n'est pas terminée. Il n'y a pas réentrance si les méthodes sont appelées successivement.

Il n'est pas possible par défaut de faire de la réentrance sur les JEB. L'élément `<reentrant>` du fichier *ejb-jar.xml* permet de modifier cette politique.

a. CMP

Cycle de vie

Le cycle de vie d'un EJB entité est plus complexe que celui d'un EJB session. En effet, en plus des méthodes de type `create()`, il y a toutes les méthodes de recherche d'un EJB en base de données, en fonction de sa clé primaire. C'est le conteneur qui gère ces recherches, qui vont déboucher sur la création d'une instance. Il s'agit donc de méthodes de création d'objet. Ces méthodes de recherches sont du type `findXxxx(...)`, elles sont appelées méthodes "finder". Ces méthodes sont déclarées dans les interfaces de type `javax.ejb.EJBLocalHome` et `javax.ejb.EJBHome`. Il faut noter que pour respecter l'encapsulation de la couche DAO par rapport aux applications clientes, l'accès aux finders est souvent effectué par des EJB de session. Il en résulte que souvent, seules les interfaces de type `Local` sont utilisées.



Le conteneur appelle les différentes méthodes du cycle de vie. En l'occurrence, avant l'appel d'une méthode métier, le conteneur va synchroniser le CMP avec la base de données. Ensuite, il va appeler la méthode `ejbLoad()`, le lien entre l'EJB et l'enregistrement étant la clé primaire.

Si certaines propriétés du CMP ne sont pas persistantes, comme des valeurs calculées, le développeur utilisera la méthode `ejbLoad()` pour mettre à jour ces propriétés.

Il faut noter que le développeur ne fournit que des classes abstraites. Le conteneur créera les classes concrètes, qui étendront les classes du développeur. Le développeur ne fournit pas non plus les variables d'instances qui stockent l'état du CMP.

Méthodes du cycle de vie

Les méthodes du cycle de vie sont exposées via l'interface `Home`, ici `javax.ejb.LocalHome`. Notez que les méthodes de recherche font partie des méthodes du cycle de vie.

```
public interface DestinationLocalHome
   extends javax.ejb.EJBLocalHome
{
   public DestinationLocal create()
      throws javax.ejb.CreateException;

   public Collection findAll()
      throws javax.ejb.FinderException;

   public DestinationLocal findByPays(String nom)
      throws javax.ejb.FinderException;

   public DestinationLocal findByPrimaryKey(Integer pk)
      throws javax.ejb.FinderException;

}
```

La classe d'implémentation qui implémente `javax.ejb.EntityBean` ne fournit pas d'implémentation pour les méthodes de recherche. Ces méthodes "finders" seront décrites dans le fichier ejb-jar.xml. Le conteneur se chargera de la création des méthodes. Nous avons les implémentations classiques des méthodes du cycle de vie `create()`, `load()`, etc.

Extraits de la classe d'implémentation :

```
public abstract class DestinationBean implements javax.ejb.EntityBean {

    private EntityContext ctx = null;

    public java.lang.Integer ejbCreate()
        throws javax.ejb.CreateException
        {
            return null;
        }

    public void ejbPostCreate() throws javax.ejb.CreateException
    {}

    public void setEntityContext(EntityContext arg0)
        throws EJBException,RemoteException
    {this.ctx = arg0;}

    public void unsetEntityContext()
        throws EJBException, RemoteException
        {
            this.ctx=null;
        }
```

Méthodes métier

En général, les méthodes métier sont principalement composées des getters et setters. Les méthodes sont ici exposées par l'interface `DestinationLocal` qui étend `javax.ejb.EJBLocalObject`.

```
public interface DestinationLocal
   extends javax.ejb.EJBLocalObject
{
   public java.lang.Integer getId(  ) ;

   public void setId(Integer id ) ;

   public java.lang.String getPays(  ) ;

   public void setPays(String pays ) ;

   public java.lang.String getDescription(  ) ;

   public void setDescription(String description ) ;

   public java.util.Collection getSejours(  ) ;

   public void setSejours(Collection sejours ) ;

}
```

L'implémentation consiste uniquement à fournir des méthodes abstraites.

```
public class DestinationBean
   implements javax.ejb.EntityBean
{
   public abstract java.lang.Integer getId(  ) ;

   public abstract void setId(Integer id ) ;

   public abstract java.lang.String getPays(  ) ;

   public abstract void setPays(String pays ) ;

   public abstract java.lang.String getDescription(  ) ;
```

```
   public abstract void setDescription(String description ) ;

   public abstract java.util.Collection getSejours(  ) ;

   public abstract void setSejours(Collection sejours ) ;

}
```

Descripteur ejb-jar.xml

Visualisons d'abord l'extrait du fichier *ejb-jar.xml* correspondant à la déclaration des EJB entités.

```
<entity id="ContainerManagedEntity_Sejour">

  <ejb-name>Sejour</ejb-name>
  <local-home>org.bovoyage.ejb2.cmp.SejourLocalHome</local-home>
  <local>org.bovoyage.ejb2.cmp.SejourLocal</local>

  <ejb-class>org.bovoyage.ejb2.cmp.SejourCMP</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>java.lang.Integer</prim-key-class>
  <reentrant>false</reentrant>
  <cmp-version>2.x</cmp-version>
  <abstract-schema-name>Sejour</abstract-schema-name>
  <cmp-field id="CMPAttribute_1">
    <field-name>id</field-name>
  </cmp-field>
  <cmp-field Id="CMPAttribute_2">
    <field-name>depart</field-name>
  </cmp-field>
  <cmp-field id="CMPAttribute_3">
    <field-name>retour</field-name>
  </cmp-field>
  <cmp-field id="CMPAttribute_4">
    <field-name>prix</field-name>
  </cmp-field>
  <primkey-field>id</primkey-field>
  <query>
    <query-method>
      <method-name>findAll</method-name>
        <method-params>        </method-params>
    </query-method>
    <ejb-ql><![CDATA[SELECT OBJECT(a) FROM Sejour as a]]></ejb-ql>
  </query>
</entity>

<entity id="ContainerManagedEntity_Destination">
  <ejb-name>Destination</ejb-name>
  <local-home>org.bovoyage.ejb2.cmp.DestinationLocalHome</local-home>
  <local>org.bovoyage.ejb2.cmp.DestinationLocal</local>
  <ejb-class>org.bovoyage.ejb2.cmp.DestinationCMP</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>java.lang.Integer</prim-key-class>
  <reentrant>false</reentrant>
  <cmp-version>2.x</cmp-version>
  <abstract-schema-name>Destination</abstract-schema-name>
  <cmp-field id="CMPAttribute_5">
    <field-name>id</field-name>
  </cmp-field>
  <cmp-field id="CMPAttribute_6">
    <field-name>pays</field-name>
  </cmp-field>
  <cmp-field id="CMPAttribute_7">
    <field-name>description</field-name>
  </cmp-field>
  <primkey-field>id</primkey-field>
  <query>
    <query-method>
      <method-name>findAll</method-name>
```

```
      <method-params></method-params>
    </query-method>
    <ejb-ql><![CDATA[SELECT OBJECT(a) FROM Destination as a]]>
....</ejb-ql>
  </query>
</entity>
```

L'élément <session> qui indique que les EJB décrits sont de type entité, contient ici, les éléments suivants :

- <description> et <display-name> sont utilisés par certains outils de gestion de EJB au sein d'un serveur. Ils peuvent être omis.
- <ejb-name> contient le nom de l'EJB. C'est ce nom qui sera repris dans le fichier *jboss.xml*.
- <home> contient le nom de l'interface qui étend javax.ejb.EJBHome, exposant les méthodes distantes du cycle de vie. Dans l'exemple, il n'y a pas d'interface Home distante.
- <remote> contient le nom de l'interface qui étend javax.ejb.EJBObject, exposant les méthodes métier distantes. Dans l'exemple, il n'y a pas d'interface Object distante.
- <local-home> contient le nom de l'interface qui étend javax.ejb.EJBLocalHome, exposant les méthodes locales du cycle de vie.
- <local> contient le nom de l'interface qui étend javax.ejb.EJBLocalObject, exposant les méthodes métier locales.
- <ejb-class> contient le nom de la classe d'implémentation des méthodes du cycle de vie et des méthodes métier. Cette classe implémente l'interface javax.ejb.EntityBean.
- <persistence-type> indique comment le conteneur gère la persistance. Il s'agit ici de CMP, donc la valeur de l'élément est Container, la valeur Bean est réservée pour les entités de type BMP.
- <reentrant> indique si la réentrace dans l'EJB est permise ou pas. False indique ici que la réentrance n'est pas permise.
- <cmp-version> indique la version du CMP.
- <prim-key-class> indique le type Java de la clé primaire.
- <abstract-schema-name> : contient le nom du schéma abstrait, qui sera, entre autres, repris dans les requêtes en EJB-QL.

Ensuite, les différentes propriétés à persister sont décrites dans un élément <cmp-field> qui contient l'élément <description> utilisé par certains outils, mais surtout, l'élément <fiel-name> contenant le nom du champ dans le schéma abstrait. Ce nom correspond aux accesseurs de la classe abstraite de l'entité.

Par exemple, le champ prix correspond aux méthodes abstraites setPrix(...) et getPrix().

Nous trouvons ensuite la déclaration des requêtes. Les requêtes correspondent aux méthodes de recherche d'entités, dites "méthodes finders". Elles sont déclarées dans l'interface Home, puisqu'elles concourent à la création d'une instance et donc, font partie du cycle de vie de l'EJB entité.

L'élément <query> comporte les éléments fils :

- <query-method> qui décrit le côté Java de la méthode finder.
- <method-name> qui est un élément de <query-method> et contient le nom du finder, ici findAll. Ce nom correspond à la méthode findAll() de l'interface Home.
- <method-params> qui est un élément de <query-method> et qui décrit les éventuels paramètres à passer à la méthode dans un élément <method-param>. Celui-ci contient le type Java du paramètre et correspondra à un paramètre de requête en EJB-QL symbolisé par ?x, où x est le rang du paramètre dans la requête.

- `<ejb-ql>` : contient la requête écrite en EJB-QL. En général, cette requête est contenue dans un CDATA pour éviter l'interprétation des signes >, en effet > peut se trouver dans une clause `WHERE` de la requête. Par exemple :

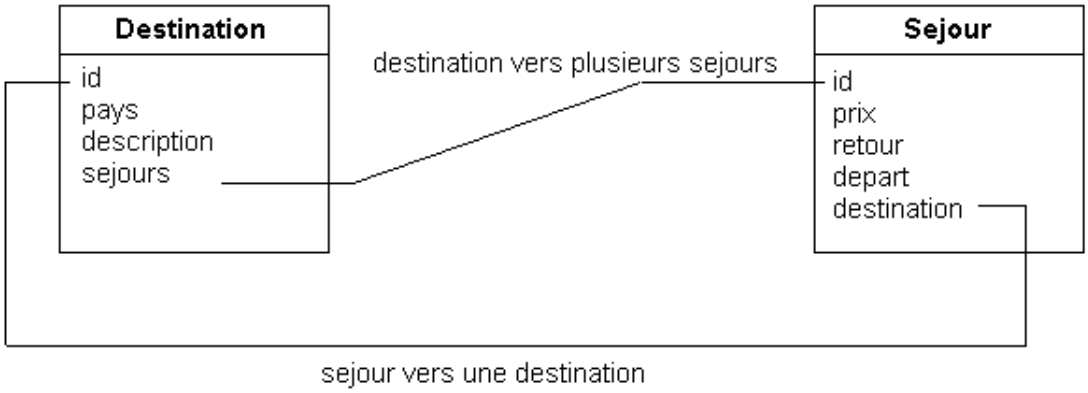
```
<query>
  <query-method>
    <method-name>findByPrixInferieurA</method-name>
    <method-params>
      <method-param>java.lang.Double</method-param>
    </method-params>
  </query-method>
  <ejb-ql>
    <![CDATA[SELECT OBJECT(d) FROM Destination AS d WHERE d.prix <?1]]>
  </ejb-ql>
</query>
```

Voici maintenant l'extrait du fichier *ejb-jar.xml* correspondant à la description de la relation existant entre les deux EJB entité.

```
<relationships id="Relationships_1">
  <ejb-relation id="EJBRelation_1">
  <ejb-relation-name>
    Destination-to-Sejours
  </ejb-relation-name>
  <ejb-relationship-role id="EJBRelationshipRole_1">
    <ejb-relationship-role-name>
       sejour-vers-une-destination
    </ejb-relationship-role-name>
    <multiplicity>Many</multiplicity>
    <relationship-role-source id="RoleSource_1">
      <ejb-name>Sejour</ejb-name>
      </relationship-role-source>
      <cmr-field id="CMRField_1">
        <cmr-field-name>destination</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>

    <ejb-relationship-role id="EJBRelationshipRole_2">
      <ejb-relationship-role-name>
        destination-vers-plusieurs-sejours
      </ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source id="RoleSource_2">
        <ejb-name>Destination</ejb-name>
        </relationship-role-source>
          <cmr-field id="CMRField_2">
            <cmr-field-name>sejours</cmr-field-name>
            <cmr-field-type>
              java.util.Collection
            </cmr-field-type>
          </cmr-field>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
```

La balise `<relationships>` contient la définition des relations entre entités. Chaque relation est définie par un élément `<ejb-relation>`. Avant de décrire les éléments de la relation, regardons comment les relations sont exprimées dans le schéma abstrait.



Il s'agit d'une relation bidirectionnelle du type "un vers plusieurs". Il existe une collection de Sejours dans Destination et une Destination dans Sejour. Le conteneur doit pouvoir retrouver les données qui correspondent aux deux entités. La description logique d'une relation est effectuée dans l'élément `<ejb-relation>`. Comme le nom de l'élément l'indique, il s'agit bien de la description de la relation entre deux EJB, et non pas en base de données, qui sera explicitée dans le fichier *jbosscmp-jdbc.xml*.

Descripteur jboss.xml

La partie déclarant les entités dans le descripteur jboss.xml est assez simple. Nous y retrouvons un élément `<entity>` qui est dans l'élément père `<enterprise-beans>`.

```
...
<entity>
    <ejb-name>Sejour</ejb-name>
    <local-jndi-name>ejb2/local/Sejour</local-jndi-name>
    <method-attribute></method-attribute>
</entity>
<entity>
    <ejb-name>Destination</ejb-name>
    <local-jndi-name>ejb2/local/Destination
      </local-jndi-name>
</entity>
...
```

Nous y retrouvons de manière classique maintenant :

- `<ejb-name>` contient le nom de l'EJB tel qu'il a été défini dans la balise `<ejb-name>` du fichier *ejb-jar.xml*.
- `<jndi-name>` contient le nom distant de l'EJB, c'est ce nom qui sera utilisé par les clients distants, via RMI. L'EJB est mis dans le contexte JNDI le plus élevé du serveur pour pouvoir être accessible en dehors du contexte serveur. Cet élément n'est pas utilisé ici.
- `<local-jndi-name>` contient le nom local de l'EJB. C'est ce nom qui sera utilisé par les clients qui sont dans la même machine virtuelle que l'EJB, donc au sein du même serveur.

L'élément `<method-attribute>` n'est pas utilisé ici, il permet d'optimiser le comportement du conteneur vis-à-vis de certaines méthodes et de leur gestion dans les transactions. Cet élément peut contenir des éléments fils `<method>` qui contiennent à leur tour :

- `<method-name>` contient le nom d'une méthode, ou un modèle de nom de méthode, comme get*, qui correspond à toutes les getters de l'entité.
- `<read-only>` contient la valeur true qui permettra à JBoss d'éviter les accès concurrents sur la même instance d'entité, lors de l'appel des méthodes marquées en lecture seule.
- `<transaction-timeout>` peut contenir une valeur de limite temporelle sur une transaction, exprimée en secondes.

Descripteur jbosscmp-jdbc.xml

Ce descripteur permet à JBoss de faire les liaisons entre les EJB entités décrits dans le fichier *ejb-jar.xml* et les

champs en base de données. Ce fichier vient compléter les paramétrages généraux qui sont définis dans le fichier de configuration *standardjbosscmp-jbdc.xml*, celui-ci étant dans le répertoire *conf* de la configuration serveur.

```
<jbosscmp-jdbc>
   <defaults>
     <datasource>java:/BovoyageDS</datasource>
     <datasource-mapping>mySQL</datasource-mapping>
     <create-table>false</create-table>
       <alter-table>false</alter-table>
     <remove-table>false</remove-table>
     <preferred-relation-mapping>
        foreign-key
     </preferred-relation-mapping>
   </defaults>

   <enterprise-beans>

      <entity>
        ...
      </entity>

      <entity>
        ...
      </entity>

   </enterprise-beans>

  <relationships>
    <ejb-relation>
      ...
    </ejb-relation>
  </relationships>

</jbosscmp-jdbc>
```

La racine `<jbosscmp-jdbc>` contient principalement les éléments suivants :

- `<defaults>` contient les valeurs par défaut, utilisées par les EJB entités et les relations. Le fichier *standardjbosscmp-jbdc.xml* contient une source de données par défaut, qui est `DefaultDS`. Ici, une nouvelle source de données est mise en place par défaut, pour nos entités et relations. N'oubliez pas que pour que la source de données existe, il faut qu'elle ait été déployée par l'intermédiaire d'un fichier *xxx-ds.xml*.

- `<enterprise-beans>` contient les EJB entités qui seront à associer à la base de données

- `<relationships>` contient la description des relations entre les entités par rapport à la base de données.

- `<dependent-value-classes>` contient la description de types complexes, autres classes, qui pourraient correspondre à des descriptions de champ contenus dans des éléments `<cmp-field>`. L'élément `<cmp-field>` sera étudié plus loin.

- `<type-mappings>` contient le mapping entre les types java, JDBC et SQL. Ce mapping dépend aussi de la base de données. Une liste existe par défaut dans le fichier *standardjbosscmp-jbdc.xml*.

- `<reserved-words>` contient des éléments `<word>` permettant de créer une liste de mots réservés. Ces mots réservés ne peuvent pas être utilisés pour nommer des tables ou des champs. Une liste existe par défaut dans le fichier *standardjbosscmp-jbdc.xml*.

- `<user-type-mappings>` contient les associations de type personnalisées.

- `<entity-commands>` permet de définir des commandes qui pourront être utilisées au sein d'un élement `<entity>`. Une liste d'`entity-command` existe dans le fichier *standardjbosscmp-jbdc.xml*.

Exemple d'`entity-command`, extrait du fichier *standardjbosscmp-jbdc.xml*.

```
<!-- retrieves generated key of the record inserted into hsql db -->
```

```
<entity-command name="hsqldb-fetch-key"
class="org.jboss.ejb.plugins.cmp.jdbc.keygen.JDBCHsqldbCreateCommand">

<attribute name="pk-sql">CALL IDENTITY()</attribute>

</entity-command>
```

Exemple d'utilisation de cette `entity-command`.

```
<jbosscmp-jdbc>
    <enterprise-beans>
        <entity>
            <ejb-name>LocationEJB</ejb-name>
            ...
            <entity-command name="hsqldb-fetch-key"/>
        </entity>
    </enterprise-beans>
</jbosscmp-jdbc>
```

Nous allons voir maintenant comment est effectué le mapping entre l'EJB entité et la base de données.

```
...
<entity>
    <ejb-name>Sejour</ejb-name>
    <datasource>java:jdbc/BovoyageDS</datasource>
    <datasource-mapping>mySQL</datasource-mapping>
    <create-table>false</create-table>
    <remove-table>false</remove-table>
    <table-name>sejours</table-name>

    <cmp-field>
        <field-name>id</field-name>
        <read-only>false</read-only>
        <column-name>kp_sejour</column-name>

        <jdbc-type>INTEGER</jdbc-type>
        <sql-type>INTEGER</sql-type>

    </cmp-field>
    <cmp-field>
        <field-name>depart</field-name>
        <read-only>false</read-only>
        <column-name>depart</column-name>

        <jdbc-type>DATE</jdbc-type>
        <sql-type>DATETIME</sql-type>

    </cmp-field>
...
</entity>
...
```

L'élément `<entity>`, contenu dans l'élément `<enterprise-beans>`, contient principalement :

- `<ejb-name>` contient le nom de l'EJB tel qu'il a été défini dans les fichiers *ejb-jar.xml* et *jboss.xml*.
- `<datasource>` contient le nom de la source de données utilisée par cette entité, sinon la source de données par défaut est utilisée.
- `<datasource-mapping>` contient le mapping de type utilisé. Attention de bien respecter l'orthographe qui est déclaré dans le fichier *standardjbosscmp-jbdc.xml*.
- `<create-table>` contient true ou false qui permet la création de la table si celle-ci n'existe pas.
- `<remove-table>` contient true ou false qui permet la suppression de la table lors du repli de l'application.

- <table-name> contient le nom de la table utilisée pour le mapping avec l'EJB entité.
- <cmp-field> contient la description des associations entre les propriétés de l'entité et les champs de la base de données.

Dans l'élément <cmp-field>, nous trouverons en général :

- <field-name> qui contient le nom du champ dans le CMP.
- <read-only> qui précise si le champ est en lecture seule ou non.
- <column-name> qui contient le nom de la colonne de la table.
- <jdbc-type> qui correspond au type JDBC.
- <sql-type> qui correspond au type SQL.

Visualiser les appels à la base de données

Le conteneur d'EJB gère et génère les requêtes SQL. Pour pouvoir les visualiser sur la console, vous pouvez ajouter une configuration pour le framework de suivi des logs : Log4j. L'ajout d'un appender dans le fichier *jboss-log4j.xml* du répertoire *conf* de la configuration serveur, permettra de suivre, sur la console, où vers un fichier en modifiant la configuration, les opérations effectuées en base de données.

Dans la liste des appenders présents, vous pouvez ajouter les lignes suivantes :

```
<appender name="CMP" class="org.apache.log4j.ConsoleAppender">
  <errorHandler class="org.jboss.logging.util.OnlyOnceErrorHandler"/>
  <param name="Target" value="System.out"/>

  <layout class="org.apache.log4j.PatternLayout">
     <param name="ConversionPattern" value="%d{ABSOLUTE} %-5p [%c{1}] %m%n"/>
  </layout>
</appender>
```

Ainsi, vous pourrez voir dans la console d'exécution de JBoss, la prise en charge des CMP et les requêtes générées.

```
13:49:42,218 DEBUG [Sejour] Initializing CMP plugin for Sejour
13:49:42,234 DEBUG [Sejour] Loading standardjbosscmp-jdbc.xml : file:/C:/
SERVEURS/jboss-4.2.2.GA/server/default/conf/standardjbosscmp-jdbc.xml
13:49:42,250 DEBUG [Sejour] jar:file:/C:/SERVEURS/jboss-4.2.2.GA/server/
default/tmp/deploy/tmp1741BovoyageEJB2_EAR.ear-contents/Bovoyage_EJB2.jar!/
META-INF/jbosscmp-jdbc.xml found. Overriding defaults
13:49:42,265 DEBUG [Destination] Initializing CMP plugin for Destination
13:49:42,281 DEBUG [Sejour] Insert Entity SQL: INSERT INTO sejours
(kp_sejour, depart, retour, prix, ke_destination) VALUES (?, ?, ?, ?, ?)
13:49:42,281 DEBUG [Sejour] Entity Exists SQL: SELECT COUNT(*) FROM sejours
WHERE kp_sejour=?
13:49:42,281 DEBUG [Sejour] entity-command: [commandName=default,commandClass
=class org.jboss.ejb.plugins.cmp.jdbc.JDBCCreateEntityCommand,attributes={}]
13:49:42,281 DEBUG [Sejour] Remove SQL: DELETE FROM sejours WHERE kp_sejour=?
13:49:42,281 DEBUG [Sejour] Table not create as requested: sejours
13:49:42,281 DEBUG [Sejour#findByPrimaryKey] SQL: SELECT t0_Sejour.kp_sejour
FROM sejours t0_Sejour WHERE t0_Sejour.kp_sejour=?
13:49:42,281 DEBUG [Sejour] Added findByPrimaryKey query command for local
home interface
13:49:42,281 DEBUG [Sejour#findAll] EJB-QL: SELECT OBJECT(a) FROM Sejour as a
13:49:42,281 DEBUG [Sejour#findAll] SQL: SELECT t0_a.kp_sejour FROM sejours
t0_a
13:49:42,281 DEBUG [Destination] Insert Entity SQL: INSERT INTO destinations
(kp_destination, pays, description) VALUES (?, ?, ?)
13:49:42,281 DEBUG [Destination] Entity Exists SQL: SELECT COUNT(*) FROM
destinations WHERE kp_destination=?
13:49:42,281 DEBUG [Destination] entity-command: [commandName=default,
commandClass=class org.jboss.ejb.plugins.cmp.jdbc.JDBCCreateEntityCommand,
```

```
attributes={}]
13:49:42,281 DEBUG [Destination] Remove SQL: DELETE FROM destinations WHERE
kp_destination=?
13:49:42,281 DEBUG [Destination] Table not create as requested: destinations
13:49:42,281 DEBUG [Destination#findByPrimaryKey] SQL: SELECT t0_Destination.
kp_destination FROM destinations t0_Destination WHERE t0_Destination.kp_
destination=?
13:49:42,281 DEBUG [Destination] Added findByPrimaryKey query command for
local home interface
13:49:42,281 DEBUG [Destination#findAll] EJB-QL: SELECT OBJECT(a) FROM
Destination as a
13:49:42,281 DEBUG [Destination#findAll] SQL: SELECT t0_a.kp_destination
FROM destinations t0_a
13:49:42,281 DEBUG [Destination#findByPays] EJB-QL: SELECT OBJECT(a) FROM
Destination AS a WHERE a.pays = ?1
13:49:42,296 DEBUG [Destination#findByPays] SQL: SELECT t0_a.kp_destination
FROM destinations t0_a WHERE (t0_a.pays = ?)
13:49:42,296 DEBUG [Sejour] Foreign key constraint not added as requested:
relationshipRolename=sejour-vers-une-destination
```

Vous pouvez y voir la prise en charge des CMP, puis la création des requêtes SQL à partir des requêtes en EJB QL.

b. BMP

Les EJB entité de type BMP *(Bean Managed Persistence)* demandent un peu plus de travail de la part du développeur. Ils peuvent être utiles lorsque plus de souplesse est demandée à l'entité, et lorsque le langage EJB QL ne suffit pas. La prise en charge par le conteneur, d'un BMP par rapport à un CMP, diffère par le fait que les accès à la base de données sont codés par le développeur. Bien que le développeur code les méthodes, elles seront toujours appelées par le conteneur d'EJB.

Cycle de vie

Le cycle de vie d'un BMP est très proche du cycle de vie d'un CMP. La grande différence est que le développeur devra fournir le code pour les méthodes de recherches du type `findXxxx(...)`. Ces méthodes sont déclarées classiquement dans les interfaces de type `javax.ejb.EJBLocalHome` et `javax.ejb.EJBHome`. Comme pour les CMP, afin de respecter l'encapsulation de la couche DAO par rapport aux applications clientes, l'accès aux finders est souvent effectué par des EJB de session. Il en résulte que souvent, seules les interfaces de type `Local` sont utilisées.

Méthodes du cycle de vie

L'interface de type `javax.ejb.LocalHome` expose les méthodes du cycle de vie. Nous y retrouvons les méthodes de création et de recherche.

```
public interface ParticipantLocalHome extends EJBLocalHome
{
    public ParticipantLocal create(String civilite,String
nom,String prenom,String email, String passeport,Integer sejourKey)
throws CreateException;

    public ParticipantLocal create(ParticipantData participant)
throws CreateException;

    public ParticipantLocal findByPrimaryKey(Integer id) throws
FinderException;

    public ParticipantLocal findByEmail(String email) throws
FinderException;

    public Collection findBySejourKey(Integer sejourKey) throws
FinderException;
}
```

Les implémentations des méthodes métier sont plus complexes que pour le CMP, puisque le développeur doit lui-même gérer les requêtes en base de données.

Exemple d'implémentation d'une méthode `create(...)` :

```
public class ParticipantBean implements EntityBean
{
...
    public Integer ejbCreate(ParticipantData participant) throws
CreateException
    {
```

```
        Integer kp = null;

        this.civilite = participant.getCivilite();
        this.nom = participant.getNom();
        this.prenom = participant.getPrenom();
        this.email = participant.getEmail();
        this.passeport = participant.getPasseport();
        this.sejourKey = participant.getSejourKey();

        try
        {
            kp = bmpCreate();

        }
        catch (SQLException e)
        {
            throw new CreateException("Erreur creation participant \n"+e);
        }
        return kp;
    }

    public void ejbPostCreate(ParticipantData participant)
    {}


    public Integer bmpCreate() throws SQLException
    {
        String sql = "INSERT INTO participants SET
civilite=?,nom=?,prenom=?,email=?,passeport=?,ke_sejour=?";
        Integer kp = null;
        Connection con = ds.getConnection();
        PreparedStatement ps = con.prepareStatement(sql);
        ps.setString(1, this.civilite);
        ps.setString(2, this.nom);
        ps.setString(3, this.prenom);
        ps.setString(4, this.email);
        ps.setString(5, this.passeport);
        ps.setInt(6, this.sejourKey.intValue());
        ps.executeUpdate();
        ResultSet rs = ps.getGeneratedKeys();
        if (rs.next())
        {
            kp = new Integer(rs.getInt(1));
        }
        ps.close();
        con.close();
        return kp;
    }

...

}
```

Exemple d'implémentation d'un finder :

```
public class ParticipantBean implements EntityBean
{
...
    public Integer ejbFindByPrimaryKey(Integer kp) throws FinderException
    {
        String sql = "SELECT * FROM participants WHERE kp_participant=?";
        Connection con = null;
        boolean trouve=false;
        try
        {
            con = ds.getConnection();
            PreparedStatement ps = con.prepareStatement(sql);
```

```
            ps.setInt(1, kp.intValue());
            ResultSet rs = ps.executeQuery();
            trouve = rs.next();
            con.close();
        }
        catch (SQLException e)
        {
            throw new EJBException("Erreur recherche primaryKey");
        }
        if(trouve)
        {
            return this.primaryKey;
        }
        else
        {
            throw new ObjectNotFoundException("Pas de primaryKey en "+kp);
        }
    }

...
}
```

Méthodes métier

Classiquement, les accesseurs sont exposés dans les interfaces `javax.ejb.EJBOject` et `javax.ejb.EJBLocalObjetc`.

```
public interface ParticipantLocal extends EJBLocalObject
{

    public String getCivilite();
    public void setCivilite(String civilite);

    public String getNom();
    public void setNom(String nom);

    public String getPrenom();
    public void setPrenom(String prenom);

    public String getEmail();
    public void setEmail(String email);

    public String getPasseport();
    public void setPasseport(String passeport);
}
```

La classe d'implémentation de type `javax.ejb.EntityBean` implémente les accesseurs et méthodes métier. Notez que la classe d'implémentation d'un CMP doit utiliser des propriétés d'instance.

```
public class ParticipantBean implements EntityBean
{
    ...

    private Integer primaryKey;
    private Integer sejourKey;
    private String civilite;
    private String nom;
    private String prenom;
    private String email;
    private String passeport;

...

    public String getCivilite()
    {
        return civilite;
    }
```

```
    public void setCivilite(String civilite)
    {
        this.civilite = civilite;
    }
...
}
```

Descripteur ejb-jar.xml

Nous retrouvons de manière classique la définition de l'EJB.

```
...<entity>
    <ejb-name>Participant</ejb-name>

    <local-home>org.bovoyage.ejb2.bmp.ParticipantLocalHome</local-home>
    <local>org.bovoyage.ejb2.bmp.ParticipantLocal</local>
    <ejb-class>org.bovoyage.ejb2.bmp.ParticipantBean</ejb-class>
    <persistence-type>Bean</persistence-type>
    <prim-key-class>java.lang.Integer</prim-key-class>
    <reentrant>false</reentrant>

...

</entity>
...
```

La recherche de la source de données de type `javax.sql.DataSource` est effectuée lors de l'appel de la méthode `setEntityContext(...)`. Il y a recherche dans le contexte JNDI par le nom JNDI, mis en place par le serveur via le fichier de configuration de la source de données.

```
public class ParticipantBean implements EntityBean
{
...

    private final String dsName = "java:jdbc/BovoyageDS";
    private DataSource ds;

    public void setEntityContext(EntityContext arg0) throws
EJBException, RemoteException
    {
        this.ctx = arg0;
        try
        {
            InitialContext ct = new InitialContext();
            ds = (DataSource) ct.lookup(this.dsName);
        }
        catch (NamingException e)
        {
            throw new EJBException(">>> Erreur recherche DataSource", e);
        }
    }
...
}
```

Il faut alors ajouter dans le fichier *ejb-jar.xml* une référence vers une ressource. Ceci est effectué avec l'élément `<resource-ref>`. Le fichier *ejb-jar.xml* devient alors le suivant :

```
<entity>
...
    <ejb-name>Participant</ejb-name>
    <local-home>org.bovoyage.ejb2.bmp.ParticipantLocalHome</local-home>
    <local>org.bovoyage.ejb2.bmp.ParticipantLocal</local>
    <ejb-class>org.bovoyage.ejb2.bmp.ParticipantBean</ejb-class>
    <persistence-type>Bean</persistence-type>
    <prim-key-class>java.lang.Integer</prim-key-class>
    <reentrant>false</reentrant>
    <resource-ref>
      <res-ref-name>jdbc/toto</res-ref-name>
```

```
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
    </resource-ref>
...
</entity>
```

L'élément <resource-ref> accepte les éléments fils suivants :

- <res-ref-name> qui correspond au nom de la ressource dans le contexte java:comp/env de l'EJB ;
- <res-type> qui contient le nom de la classe correspondant à la ressource, ici javax.sql.DataSource ;
- <res-auth> qui demande l'authentification auprès de la ressource par l'intermédiaire du conteneur. Si l'EJB gère l'authentification par programmation, cet élément doit contenir Application. Il faut faire attention dans ce cas aux problèmes de portabilité de l'EJB.

Avec cette configuration, la ressource peut être accessible par le nom JNDI java:comp/env/jdbc/toto au lieu de java:jdbc/BovoyageDS.

Pour que ceci fonctionne, il faut aussi mettre en place, dans le fichier jboss.xml, l'association entre les deux nommages JNDI. C'est ce que nous allons voir maintenant.

Descripteur jboss.xml

Dans le descripteur *jboss.xml*, la déclaration du BMP dans l'élément <entity> vous est maintenant familière. Vous y retrouvez le nom de l'EJB tel qu'il est défini dans le fichier ejb-jar.xml, et ses associations JNDI. Ici, seule l'interface LocalHome est présente, il n'y a donc pas de nom JNDI pour l'interface de création distante.

```
<jboss>

   <enterprise-beans>

  <entity>
    <ejb-name>Participant</ejb-name>
    <local-jndi-name>ejb2/local/Participant</local-jndi-name>

  </entity>
...
  <resource-managers>
    <resource-manager>
        <res-name>jdbc/toto</res-name>
        <res-jndi-name>java:jdbc/BovoyageDS</res-jndi-name>
    </resource-manager>
  </resource-managers>

</jboss>
```

Pour assurer l'association entre le nom JNDI utilisé dans l'EJB, soit dans l'exemple précédent java:comp/env/jdbc/toto, et le nom JNDI de la ressource telle qu'elle a été enregistrée par JBoss, soit java:jdbc/BovoyageDS, il nous faut ajouter les lignes suivantes au fichier *jboss.xml*.

```
<jboss>
...
<resource-managers>
    <resource-manager>
        <res-name>jdbc/toto</res-name>
        <res-jndi-name>java:jdbc/BovoyageDS</res-jndi-name>
    </resource-manager>
</resource-managers>
...
</jboss>
```

Ces lignes sont situées en fin de fichier. Les différents alias sont décrits au sein des balises <resource-manager> qui ont, comme élément parent, <resource-managers>.

- <res-name> contient le nom de la ressource dans le fichier *ejb-jar*. Dans le fichier *ejb-jar.xml*, ce nom est contenu dans la balise <res-ref-name> ;

- <res-jndi-name> contient le nom JNDI de la ressource tel qu'il a été monté dans le contexte JNDI par JBoss.

Vérifiez que JBoss est bien en cours d'exécution et que l'archive BovoyageEJB2_EAR est déployée, ouvrez un navigateur et tapez l'URL de la console JMX : http://localhost:8080/jmx-console.

Retrouvez le service d'affichage de l'annuaire de nommage JNDI, c'est-à-dire le lien service=JNDIView dans la rubrique jboss.

Cliquez sur le lien.

Cliquez sur le bouton **Invoke** de l'opération `list()` du service.

Vous devez retrouvez le contexte de nommage pour le BMP `Participant`.

```
java:comp namespace of the Participant bean:

  +- env (class: org.jnp.interfaces.NamingContext)
  |  +- jdbc (class: org.jnp.interfaces.NamingContext)
  |  |   +- toto[link -> java:jdbc/BovoyageDS] (class: javax.naming.LinkRef)
```

Nous voyons que le contexte `java:comp/env/jdbc/toto` est lié au contexte `java:jdbc/BovoyageDS`.

Retrouvez le contexte de nommage `java:` :

```
java: Namespace

  +- XAConnectionFactory (class: org.jboss.mq.SpyXAConnectionFactory)
  +- DefaultDS (class: org.jboss.resource.adapter.jdbc.WrapperDataSource)
  +- SecurityProxyFactory (class:
org.jboss.security.SubjectSecurityProxyFactory)
  +- DefaultJMSProvider (class: org.jboss.jms.jndi.JNDIProviderAdapter)
  +- comp (class: javax.naming.Context)
  +- jdbc (class: org.jnp.interfaces.NamingContext)
  |  +- BovoyageDS (class: org.jboss.resource.adapter.jdbc.WrapperDataSource)
  +- JmsXA (class: org.jboss.resource.adapter.jms.JmsConnectionFactoryImpl)
  +- ConnectionFactory (class: org.jboss.mq.SpyConnectionFactory)
```

Le lien est effectif. Lorsque l'EJB recherchera dans son contexte de nommage la ressource référencée par `java:comp/env/jdbc/toto`, il recevra la ressource qui y est liée : `java:jdbc/BovoyageDS`.

L'implémentation de l'EJB est devenu indépendante des nommages des ressources.

6. EJB2 orienté message

Si vous n'êtes pas familier avec les services de messagerie JMS, reportez-vous au chapitre Architecture de JBoss.

Les EJB orientés message, MDB pour *Message Driven Bean*, sont simples.

- Ils n'ont pas d'interface d'exposition de méthodes métier ou de création, donc par de Home, LocalHome, Object, LocalObject.

- Ils disposent d'une méthode onMessage().

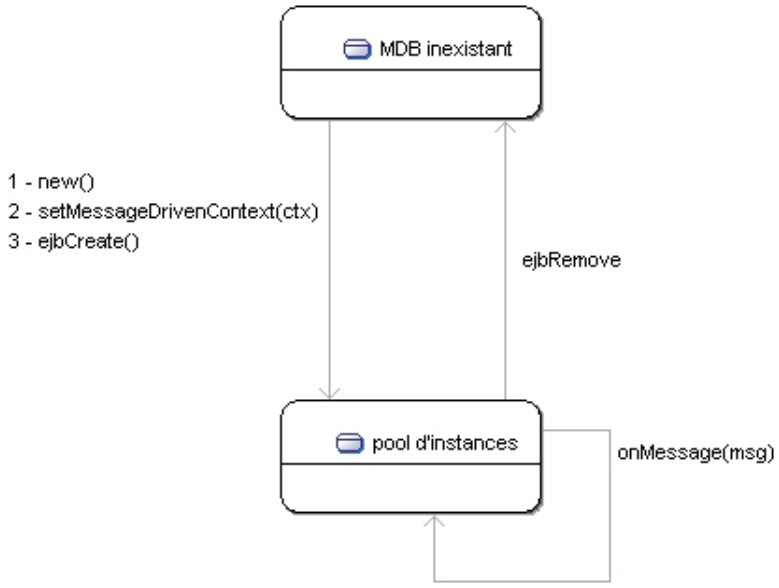
- Ils sont sans état.

Les MDB peuvent être des abonnés permanents ou non. Si le MDB est un abonné durable au service de messagerie, il recevra tous les messages, même si l'EJB orienté message est inactif. Si le MDB n'est pas un abonné permanent, les messages seront perdus si le l'EJB est inactif.

L'intérêt des MDB est de pouvoir envoyer ou recevoir des messages asynchrones vers ou provenant d'applications tiers, sans qu'il y ait de gros impact sur le reste de l'application. D'autres modes de communications avec des applications tiers sont, bien entendu, envisageables, comme les web services par exemple.

Cycle de vie

Comme l'EJB session sans état, le cycle de vie du MDB est très simple.



Méthodes du cycle de vie

L'implémentation de ces méthodes est ici triviale, le cycle de vie de ce type d'EJB étant très simple.

```
public class ReceptionConfirmationBean implements
javax.ejb.MessageDrivenBean, javax.jms.MessageListener
{

    private javax.ejb.MessageDrivenContext messageContext = null;

    public void setMessageDrivenContext(MessageDrivenContext
messageContext) throws javax.ejb.EJBException

    {
        this.messageContext = messageContext;
    }

    public void ejbCreate()
    {}

    public void ejbRemove()
    {
        messageContext = null;
    }
...
}
```

Ici, nous avons juste initialisé la propriété `messageContext`, de type `javax.ejb.MessageDrivenContext`. C'est en fait, l'outil XDoclet qui s'est chargé de la tâche.

Méthode métier

La méthode `onMessage(...)` présentée ici est triviale : un simple affichage dans la console du message reçu. Elle sera invoquée par le conteneur d'EJB lors de la réception d'un message.

```
public class ReceptionConfirmationBean implements
javax.ejb.MessageDrivenBean, javax.jms.MessageListener
{
...
    public void onMessage(javax.jms.Message message)
    {
        // begin-user-code
        System.out.println("Message Driven Bean got message " + message);
       // TODO:  do business logic here
```

```
        // end-user-code
    }
...
}
```

L'envoi du message est assuré, pour cet exemple, par une application Java indépendante. Cet envoi de message peut être effectué par toute application d'entreprise orientée message, sans lien avec le langage Java et les plates-formes Sun.

```
public class ConfirmationSejour
{

    private QueueConnectionFactory factory = null;
    private InitialContext ctx = null;
    public static final String JNDI_QUEUE = "queue/B";
    public static final String JNDI_QUEUE_FACTORY = "QueueConnectionFactory";
    private Queue queue = null;

    public ConfirmationSejour()
    {
        try
        {
            ctx = new InitialContext();
            factory = (QueueConnectionFactory) ctx.lookup
(JNDI_QUEUE_FACTORY);
            queue = (Queue) ctx.lookup(JNDI_QUEUE);
        }
        catch (NamingException e)
        {
            System.out.println("ERREUR SUR CONSTRUCTEUR : " + e);
        }
    }

    public void envoyerMessage(int numeroSejour)
    {
        QueueConnection con = null;
        try
        {
            con = factory.createQueueConnection();

            QueueSession session =
con.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);

            QueueSender sender = session.createSender(queue);

            MapMessage message = session.createMapMessage();
            Message.setInt("numeroSejour", numeroSejour);
            message.setString("action", "confirmation");
            sender.send(message);
            System.out.println(">>> MESSAGE ENVOYE : '" + message + "'");
        }
        catch (JMSException jmse)
        {
            System.out.println("Erreur lors de l'envoi des
messages : " + jmse.getMessage());
        }
        finally
        {
            if (con != null)
            {
                try
                {
                    con.close();
                }
                catch (JMSException jmse)
                {
                }
            }
```

```
        }
    }

    public static void main(String[] args) {
        ConfirmationSejour cl = new ConfirmationSejour();
        cl.envoyerMessage(12);
    }

}
```

Le descripteur de déploiement ejb-jar.xml

Le descripteur de déploiement précise le nom de l'EJB, la classe utilisée et le mode de gestion par le conteneur. Une seule classe est précisée puisqu'un EJB orienté message ne possède pas d'interface d'exposition des méthodes métier ou de cycle de vie. Attention, il y a des différences notables sur le descripteur d'un EJB orienté message entre les spécifications EJB 2.0 et EJB 2.1. Nous avons ici un descripteur aux spécifications EJB 2.1.

```
...
<message-driven id="MessageDriven_1">
  <ejb-name>ReceptionConfirmation</ejb-name>

  <ejb-class>org.bovoyage.ejb2.mdb.ReceptionConfirmationMdb</ejb-class>

  <messaging-type>javax.jms.MessageListener</messaging-type>
  <transaction-type>Container</transaction-type>
  <message-destination-type>
    javax.jms.Queue
  </message-destination-type>
  <activation-config>
    <activation-config-property>
      <activation-config-property-name>
        destinationType
      </activation-config-property-name>
      <activation-config-property-value>
        javax.jms.Queue
      </activation-config-property-value>
    </activation-config-property>
    <activation-config-property>
      <activation-config-property-name>
        acknowledgeMode
      </activation-config-property-name>
      <activation-config-property-value>
        Auto-acknowledge
      </activation-config-property-value>
    </activation-config-property>
  </activation-config>

</message-driven>

...
```

L'élément `<message-driver>` permet d'assembler l'EJB et contient, entre autres, les éléments fils suivants :

- `<ejb-name>` : nom de l'EJB ;
- `<ejb-class>` : classe d'implémentation de l'EJB ;
- `<messaging-type>` : définit la classe fournisseur de message ;
- `<transaction-type>` : transaction gérée par le conteneur ou le bean ;
- `<message-destination-type>` : destination utilisée par l'EJB : sujet (topic) ou file d'attente (queue) ;
- `<activation-config>`.

L'élément <activation-config> a été introduit dans la spécification EJB 2.1. Il remplace les éléments <acknowledge-mode> et <message-selector> de la spécification EJB 2.0. Cet élément est constitué d'une suite d'éléments fils <activation-property> qui permet de définir une propriété. Le nom de cette propriété est défini dans l'élément <activation-config-property-name> et sa valeur dans l'élément <activation-config-property-value>. Les noms des propriétés supportées pour les EJB MDB sont :

- acknowledgeMode qui spécifie le mode d'accusé de réception. Remplace la balise <acknowledge-mode>.
- messageSelector qui permet de filtrer les messages reçus. Si aucun filtre n'est précisé, l'EJB reçoit tous les messages. Remplace l'élément <message-selector>.
- destinationType précise la destination utilisée par le MDB : sujet (topic) ou file d'attente (queue) en indiquant l'interface qui devra être implémentée par la destination concrète. Les interfaces possibles sont : javax.jms.Queue et javax.jms.Topic. Remplace l'élément fils <destination-type> de l'élément <message-driven-destination>.
- subscriptionDurability précise l'abonnement à un sujet, il est permanent (Durable) ou on (NonDurable, par défaut). Remplace l'élément fils <subscription-durability> de l'élément <message-driven-destination>.

Le descripteur de déploiement jboss.xml

Le descripteur *jboss.xlm* est très simple, il précise le nom JNDI de la destination utilisée par l'EJB. De manière classique, le nom de l'EJB est celui défini dans le descripteur *ejb-jar.xml*.

```
...

<message-driven>
  <ejb-name>ReceptionConfirmation</ejb-name>
  <destination-jndi-name>queue/B</destination-jndi-name>
</message-driven>

....
```