# Architecture des applications distribuées

Les applications informatiques ont pris une place centrale dans la plupart des entreprises. Les premières applications étaient monolithiques, d'une seule pièce. Ce type d'architecture se prête mal à la demande des entreprises. L'entreprise va demander à l'application informatique de lui fournir de nouveaux services en fonction de l'évolution de son environnement. Il est complexe d'ajouter de nouvelles fonctionnalités à une application monolithique, sans provoquer une régression de certaines parties du code. De plus, Internet a révolutionné le besoin de connectivité vis-à-vis de l'entreprise. En quelques années, les équipes de conception logicielle ont dû revoir leur mode de travail : connectivité, modélisation objet, utilisation de frameworks, etc. L'architecture des applications d'entreprise a largement évolué car le mode d'utilisation de ses applications a changé.

De nombreux postes informatiques sont maintenant reliés en réseau, ce qui permet à plusieurs salariés de travailler sur la même application. Des succursales de l'entreprise peuvent se connecter sur l'application, via Internet, pour gérer des données techniques, suivre les statistiques de ventes, effectuer des saisies, etc. Les clients de l'entreprise peuvent avoir accès, via son site web, à des données les concernant : factures, vérification de leurs cordonnées, changement de compte bancaire… Ils peuvent aussi être prévenus par SMS ou mail du suivi de leur commande, état de leur compte…

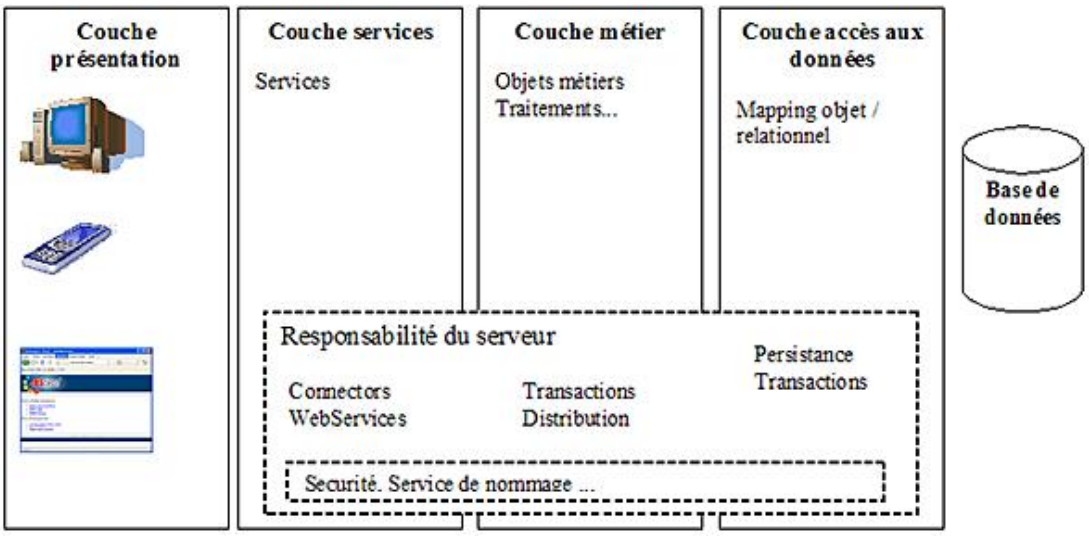
Les concepteurs d'applications doivent faire face à plusieurs défis :

- des types variés de moyens de saisie et d'affichage de l'information : clients lourds, navigateurs, téléphone mobile…
- des fonctionnalités qui évoluent : de nouveaux traitements doivent être ajoutés, des modifications de règles législatives, de nouveaux services client…
- des règles de sécurité différentes pour les salariés, les succursales, les clients finaux, les partenaires…
- Internet qui a considérablement modifié l'architecture avec l'utilisation de serveurs web, un nombre de connexions difficile à prévoir, la gestion des montées en charges, les connexions aux bases de données…

La conception d'une application amène à réfléchir sur :

- ce qui est du domaine du métier de l'entreprise : gestion des clients, de la fabrication, des stocks…
- ce qui est du domaine du support de l'application : base de données, gestion de la sécurité, transactions, réseau… Il faut remarquer que ces derniers points sont communs à toutes les applications.

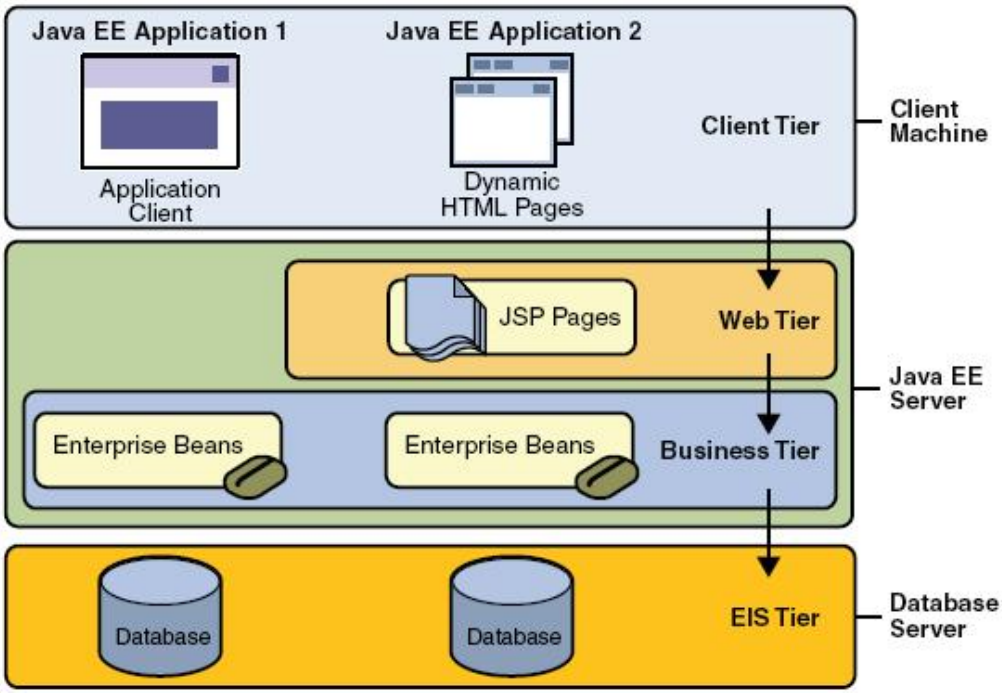
Une architecture en couche permet de mieux maîtriser l'évolution de l'application, que ce soit au niveau fonctionnel, ou au niveau logique.




Le développeur n'a pas à se soucier du codage des contextes de transactions ou de ceux de l'authentification et sécurisation. Toutes ces fonctionnalités, présentes quelle que soit l'application et transversales aux couches applicatives, sont mises à disposition par le serveur. Il peut se consacrer au développement métier en utilisant les services précédemment décrits.

Sun, via la plate-forme JEE *(Java Enterprise Edition)*, a mis en place une standardisation de ses services et la manière de les utiliser. Le développeur pourra se consacrer au codage des couches de présentation et métier, grâce à des

composants pris en charge par le serveur d'application.




Ces composants sont principalement :

- Les servlets, qui décodent les en-têtes des requêtes HTTP et codent les en-têtes de réponses HTTP, gèrent les sessions HTTP, mettent à disposition du développeur des objets prenant en charge le cycle de vie de l'application Web. Les servlets sont complétées par d'autres composants et technologies : les JSP *(Java Server Page)*, les balises personnalisées, l'EL *(Expression Language)*, les JSF *(Java Server Face)*...

  Ces composants sont pris en charge par le serveur Web.

- Les EJB *(Enterprise Java Bean)* qui sont des composants dont le cycle de vie est géré par le conteneur d'EJBs. Le conteneur va, par ces composants, gérer les objets et services métier, la persistance, l'envoi et la réception de messages, les transactions... Le développeur doit se préoccuper uniquement de la logique métier :

  - comment calculer la facture
  - gérer un panier d'achat

  Le conteneur prend en charge les aspects comme la sécurité ou les transactions.

  - l'utilisateur a-t-il le droit de calculer la facture ?
  - le paiment n'est pas validé sur le panier d'achat si un problème réseau survient.

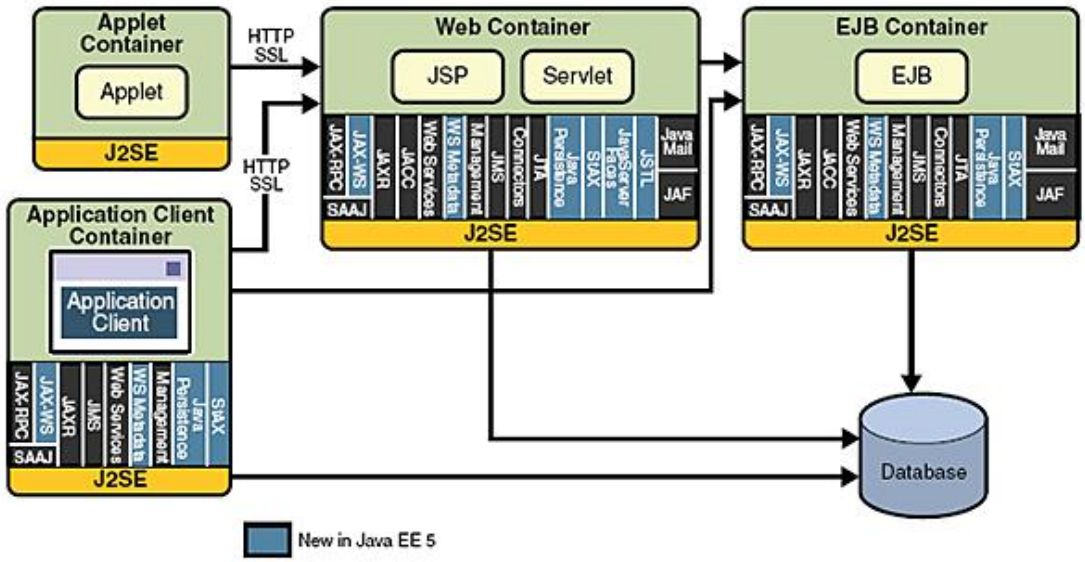
  Il existe deux grandes spécifications pour les EJB : celle des EJB 2 liée à J2EE, et celle des EJB 3 liée à JEE.

  La connaissance de ces deux spécifications est actuellement nécessaire car un certain nombre de projets ayant vu le jour avant les EJB 3, il faut donc continuer à les faire vivre, les déployer et les administrer.

Les technologies incluses dans JEE permettent, entre autres :

- la communication entre objets ;
- la gestion des objets de réponses aux requêtes HTTP ;
- la gestion des transactions ;

- la persistance des objets avec les EJB entité ;
- la communication asynchrone par message avec les EJB orienté message ;
- la réalisation d'interface graphique ;
- la gestion de la sécurité ;
- la gestion de la répartition de charge.




Un serveur d'application JEE peut être vu comme la réunion :

- d'un conteneur Web qui gère le cycle de vie des servlets et JSP ;
- d'un conteneur EJB qui gère le cycle de vie des objets métier ;
- d'un ensemble de services : accès aux bases de données, gestion des transactions…

## 1. L'invocation de méthodes distantes : RMI

Les objets des différentes couches sont susceptibles d'être invoqués par des objets situés sur une autre machine virtuelle. L'objet dont les méthodes peuvent être invoquées, est dit "objet serveur", l'objet invoquant les méthodes de l'objet serveur est dit "objet client". Il est évident que dans la réalité, les objets sont couramment serveur et client.

L'invocation directe d'une méthode dans une même machine virtuelle est simple : il s'agit d'un appel en mémoire vers une adresse, les arguments de la méthode appelée et sa valeur de retour sont passés en tant que références. C'est simple, efficace et rapide.

```
MonCalendrier cal = new MonCalendrier();
Integer annee = new Integer(2008);
Date paques = cal.getDatePaques(annee);
```

L'invocation d'une méthode d'un objet situé dans une machine virtuelle différente est plus complexe. Nous n'avons comme lien entre les deux machines virtuelles que le réseau, sous protocole TCP/IP.

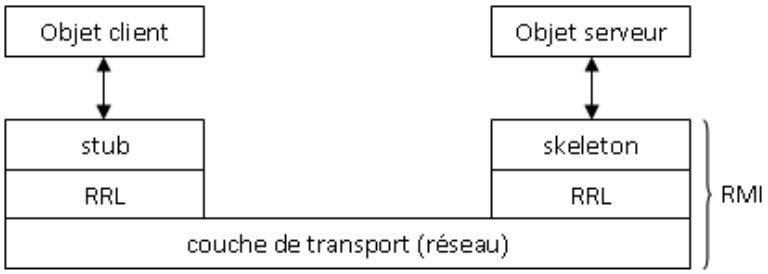
Il faut donc :

- connaître l'IP de la machine où se trouve l'objet serveur ;
- que l'objet serveur soit à l'écoute des demandes sur un socket de type serveur ;

- que l'objet client initie une demande vers l'objet serveur via un socket ;
- que les échanges soient effectués sur un protocole commun.

Donc, nous sommes loin de l'utilisation de l'opérateur point (.) pour invoquer la méthode distante. Le développeur passera sûrement plus de temps à élaborer un protocole de communication entre objets, qu'à se consacrer à l'écriture de son application !

Heureusement, nous pouvons utiliser la technologie RMI *(Remote Method Invocation)* qui va masquer, pour nous, l'ensemble des opérations décrites ci-dessus.




Une interface expose les méthodes distantes. Elle est implémentée par l'objet serveur et est utilisée par l'objet client. RMI est construit sur trois couches.

La première couche comprenant les stubs et skeletons est une couche proxy, ce qui permet au développeur de ne pas connaître les détails d'implémentation de RMI et d'utiliser l'opérateur point (.) pour l'invocation de la méthode distante. La classe `stub`, côté client, est créée par le compilateur `rmic`, présent dans le répertoire *bin* du JDK. Cette classe va sérialiser les paramètres passés à la méthode distante, et désérialiser la valeur de retour de la méthode distante. La classe `skeleton`, côté serveur, est chargée de désérialiser les paramètres reçus pour les transmettre à la méthode appelée, et de sérialiser la valeur de retour. Il faut donc que les classes des paramètres et de la valeur de retour soient sérialisables. Le passage des paramètres se fait alors par copie et non par référence.

La seconde couche, RRL *(Remote Reference Layer)*, est chargée de la localisation de l'objet distant. Elle permet d'obtenir une référence à l'objet distant, à partir de la référence locale (le stub). Ce service est lié à un service de nommage, qui peut être un service JNDI *(Java Naming and Directory Interface)* ou un service de base appelé RMI registry, lancé avec la commande `rmiregisty`, situé dans le répertoire *bin* du JDK.

La troisième couche de transport est basée sur TCP/IP. Cette couche utilise les classes `Socket` et `SocketServer`.

La compréhension de RMI est primordiale pour aborder les applications distribuées en Java car RMI est utilisé pour la communication avec les EJB. Il faut noter qu'il existe des différences notables entre l'implémentation de RMI dans le JDK 1.1 et dans JDK 1.5. Les lecteurs intéressés pourront trouver plus de renseignements sur le site de Sun (*http://java.sun.com*).

Le processus de développement d'une application RMI est le suivant :

- définir l'interface exposant les méthodes distantes ;
- implémenter les méthodes distantes ;
- développer le client ;
- compiler les classes ;
- générer les classes stub avec `rmic` ;
- lancer`rmiregistry` ;
- lancer l'application serveur ;
- lancer l'application cliente.

L'ensemble des sources peut être téléchargé sur le site ENI. L'archive téléchargée est composée de plusieurs projets Eclipse qui vous permettront de tester les configurations proposées. Ces projets illustrent les différents points abordés dans cet ouvrage, et vous permettront de tester au fur et à mesure de la lecture les concepts abordés. Le projet comporte trois classes :

- `ITime` qui est l'interface exposant la méthode qui peut être appelée par RMI : `getDate()` ;
- `TimeServeur` qui est l'implémentation de `ITime`, et qui possède une méthode `main()` permettant de lancer le serveur ;
- `TimeClient` qui invoque la méthode distante.

L'ensemble de ces classes est volontairement simple et sans package, afin de faciliter l'utilisation en ligne de commande.

L'interface `ITime` se présente ainsi :

```
import java.rmi.*;
import java.util.*;

public interface ITime extends Remote
{
    public Date getDate() throws RemoteException;
}
```

La méthode distante est susceptible de déclencher une `RemoteException`. L'interface, qui expose nos méthodes métier, doit étendre l'interface `java.rmi.Remote`.

La classe d'implémentation `TimeServeur` se présente ainsi :

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

public class TimeServeur extends UnicastRemoteObject implements Itime
{

    public TimeServeur() throws RemoteException
    {
          super();
    }

    public Date getDate()
    {
        log();
        return new Date();
    }

    public static void main(String[] args)
    {
        System.out.println("SERVEUR DEMARRE");

        try
        {
            ITime time = new TimeServeur();
            Naming.rebind("TimeServeur", time);

        } catch (Exception e)
        {
            System.out.println("ERREUR : " + e);
        }
    }

    private void log()
    {
        try
        {
            System.out.println("Connexion du client : " + getClientHost());
        } catch (Exception e)
        {
            System.out.println("ERREUR : "+e);
        }
    }
}
```

La classe d'implémentation doit étendre la classe `java.rmi.server.UnicastRemoteObject` qui permet l'export du stub, qui communiquera avec l'objet distant. La méthode `main(...)` de la classe permet l'inscription de l'instance de `TimeServeur` auprès du service de nommage, qui aura été lancé via rmiregistry. Cette inscription s'effectue par l'instruction `Naming.rebind("TimeServeur", time)`, qui va lier l'instance `time` à la chaîne de caractère `"TimeServeur"`. Ainsi, le client pourra récupérer, auprès du service de nommage, une instance de type `ITime` par la clé `"TimeServeur"`. L'implémentation de la méthode `getDate()` ne fait que renvoyer la date système et appeler une méthode `log()`, qui affichera sur la console du serveur, les informations de connexion du client.

La classe cliente `TimeClient` se présente ainsi :

```
import java.rmi.*;
import java.util.*;

public class TimeClient {
    public static void main ( String[] args ) {

      try {
        ITime service = (Itime)
          Naming.lookup("rmi://localhost/TimeServeur");

        Date serverdate = service.getDate();

        System.out.println("ServiceTime : " + serverdate);
      }
      catch (Exception e) {
        System.out.println("\nErreur\n" + e);
      }
    }
}
```

Cette classe recherche, auprès du service de nommage, un stub du type `ITime`.

Les méthodes `main(...)` des classes `TimeServeur` et `TimeClient` seront lancées dans des JVM différentes, donc, des consoles différentes.

Le lancement d'une console en ligne de commande s'effectue, sous Windows, par le menu **démarrer - Exécuter**, puis en entrant cmd. Ce lancement peut aussi s'effectuer par **démarrer - Accessoires...**

Le lancement d'une console, sous Ubuntu, s'effectue par **Application - Accessoires - Terminal**.

Ouvrez une première console pour la compilation et la génération du stub. Les classes sont compilées avec `javac`, puis le stub de la classe `TimeServeur` est généré avec `rmic`. L'exécution de ses deux lignes est effectuée dans le même répertoire que les classes.

**Vérifiez que votre variable PATH contienne le répertoire *bin* du JDK installé sur votre machine.**

```
javac *.java
rmic TimeServeur
```

Un fichier `TimeServeur_Stub.class` a été créé par `rmic`. Vous trouverez ce fichier avec les autres classes générées par `javac`.

Avant de lancer le serveur, il faut lancer le service de nommage. Pour cela, ouvrez une nouvelle console puis exécutez la commande `rmiregistry`.

```
start remiregistry
```

Dans une nouvelle console, vous pouvez maintenant lancer le serveur.

```
java TimeServeur
```

Le lancement du serveur permet d'enregistrer la classe `TimeServeur` auprès du service de nommage, grâce à la méthode `rebind(…)` de la classe `Naming`. Il faut noter qu'il existe aussi, une méthode `bind(…)` qui permet l'enregistrement d'un objet ; la méthode `rebind(…)` permettant, en plus, de remplacer l'objet déjà existant.

```
public static void main(String[] args)
    {
        System.out.println("SERVEUR DEMARRE");

        try
```

```
        {
            ITime time = new TimeServeur();
            Naming.rebind("TimeServeur", time);

        } catch (Exception e)
        {
            System.out.println("ERREUR : " + e);
        }
    }
```

Dans une autre console, vous pouvez maintenant lancer l'application cliente.

```
java TimeClient
```

Le client peut maintenant récupérer une référence vers l'objet distant par un `Naming.lookup()` et l'utiliser.

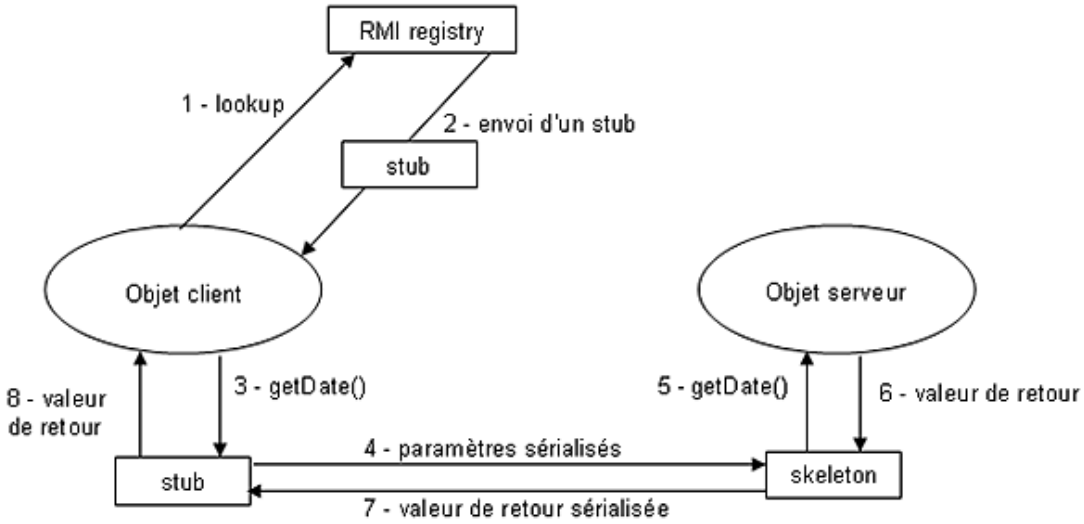
```
public static void main ( String[] args ) {

      try {
          ITime service = (Itime)
            Naming.lookup("rmi://localhost/TimeServeur");

          Date serverdate = service.getDate();

          System.out.println("ServiceTime : " + serverdate);
      }
      catch (Exception e) {
          System.out.println("\nErreur\n" + e);
      }
    }
```

La méthode `lookup()` attend une URL du type :

`rmi://<hote>[<:port>]/<nom_du_service>`

où `<hote>` identifie une machine sur le réseau par son nom ou son adresse IP, par défaut, le port est en 1099. Ici, `localhost` est utilisé car les tests sont effectués sur une même machine.

Le schéma suivant résume les appels effectués.




Il faut toujours avoir présent à l'esprit qu'un appel d'une méthode par RMI est plus consommateur de ressources et moins optimisé qu'un appel direct de la méthode au sein de la même machine virtuelle. Par RMI, les valeurs des paramètres envoyés à la méthode sont passées par copie, via la sérialisation. Il en est de même pour les valeurs de retour des méthodes. Lors d'un appel direct, ces mêmes valeurs auraient été passées par valeur. De plus, un appel au sein de la mémoire de la machine virtuelle sera toujours plus rapide qu'un appel via le réseau. Ce constat n'est pas sans conséquence pour la technologie des EJB 2, comme nous pourrons le voir par la suite.

# Rappels sur XML

Nous allons utiliser XML dans tous les descripteurs de déploiement, aussi nous allons voir brièvement les caractéristiques de cette spécification. Pour plus de précisions, reportez-vous aux spécifications XML gérées par le consortium W3C.

XML *(eXtensible Markup Language)* est un langage à base de balises. Les balises XML décrivent des contenus. L'objectif est de structurer le document, ce que ne fait pas HTML, qui décrit comment présenter le document dans un navigateur.

Un élément XML est structuré de la manière suivante :

```
<nom age="25">          => balise ouvrante
  Toto                  => corps
</nom>                  => balise fermante
```

- une balise ouvrante : ici, la balise `nom`. Le nom de la balise est libre, il peut être fixé par un schéma. La balise ouvrante est encadrée des signes inférieur (<) et supérieur (>).
- des attributs et leur valeur : ici, nous n'avons qu'un attribut age. Le signe égal sépare l'attribut et sa valeur, la valeur d'un attribut doit être entre guillemets (") ou apostrophe ('). Les attributs ne sont pas obligatoires. Ils n'apparaissent que dans la balise d'ouverture.
- le corps de l'élément : ici, `Toto`. Le corps de l'élément peut être vide, contenir un élément fils, du texte ou un élément fils et du texte.
- la balise fermante, qui reprend le nom de la balise ouvrante précédée par le signe barre oblique (/). Elle ne comporte pas d'attributs. Elle est encadrée des signes inférieur (<) et supérieur (>).
- les noms de balises ne peuvent pas comporter d'espace et commencer par un chiffre.

Toute balise ouvrante doit être fermée. Un élément qui ne comporte pas de corps peut être fermé directement.

`<br></br>` est équivalent à `<br />`

Tout document XML doit comporter un élément racine. Un document XML vide est un document dont l'élément racine est vide, et non pas un fichier vide.

```
<server>
</server>
```

XML est sensible à la casse, les différences minuscule/majuscule sont prises en compte.

L'exemple suivant est incorrect :

```
<nom>
</NOM>
```

Pour éviter l'analyse de certaines parties du document, le texte libre est mis dans des sections CDATA qui débutent par `<![CDATA[` et finissent par `]]>`.

```
<description>
  <![CDATA[Chap 2 - Calculette EJB2 generated by eclipse
wtp xdoclet extension.]]>
</description>
```

Les éléments ne peuvent pas se chevaucher. L'exemple suivant est incorrect :

```
<balise1>
<balise2>
</balise1>
</balise2>
```

Les commentaires XML commencent par `<!--` et finissent par `-->`.

```
<!-- commentaire valide -->
```

Les éléments peuvent être mis en commentaire. Dans l'exemple suivant, l'élément `<session>` n'est plus pris en compte.

```
<!--
      <session>
         <ejb-name>Calculette</ejb-name>
         <jndi-name>ejb2/calculette/remote</jndi-name>
         <local-jndi-name>ejb2/calculette/local</local-jndi-name>

        <method-attributes>
        </method-attributes>
      </session>
-->
```

Attention : les commentaires ne peuvent pas être imbriqués, ce qui peut arriver lorsque vous mettez en commentaire des parties de fichier de configuration.

```
<!--
      <session>
         <ejb-name>Calculette</ejb-name>
         <jndi-name>ejb2/calculette/remote</jndi-name>
         <local-jndi-name>ejb2/calculette/local</local-jndi-name>

        <method-attributes>
        </method-attributes>
      </session>

    <!--
      write a merge file jboss-webservices.ent for webservice-description
    -->

   </enterprise-beans>
-->
```

Si l'ensemble de ces règles est observé, le document XML est dit "bien formé" (*well formed*).

Les documents XML sont analysés par des outils appelés parseurs (*parsers*).

Les spécifications Sun et JBoss imposent un vocabulaire pour les noms de balises et d'attributs. Ces mêmes spécifications imposent aussi l'ordre d'apparition des balises, leur nombre, le caractère obligatoire ou non d'un attribut...

Ces spécifications sont appelées schéma. Il en existe deux types :

- les DTD *(Document Type Description)*, les plus simples ;
- les XMLSchemas, plus complexes mais beaucoup plus riches.

Vous retrouverez les schémas en en-tête des fichiers XML.

Mise en place d'une DTD :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jboss PUBLIC "-//JBoss//DTD JBOSS 4.0//EN"
"http://www.jboss.org/j2ee/dtd/jboss_4_0.dtd">

<jboss>

   <enterprise-beans>

...

</jboss>
```

Mise en place d'un XMLSchema :

```
<?xml version="1.0" encoding="UTF-8"?>

<ejb-jar id="ejb-jar_1"
xmlns="http://java.sun.com/xml/ns/j2ee"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd" version="2.1">

...

</ejb-jar>
```

Les parseurs, à l'aide des schémas, vont pouvoir analyser si votre fichier de configuration obéit bien aux spécifications. Un document bien formé et conforme à son schéma est dit valide.

Il est important de vérifier dans les logs si le non fonctionnement d'un EJB, ou d'une application Web, n'est pas dû à un fichier de configuration non valide. Le parseur de JBoss vérifie les fichiers XML avant de monter le module dans son conteneur.

# Rappels sur J2EE 1.4

Avec la spécification J2EE 1.4, un EJB est un composant constitué, au minimum de (ou des) :

- une (ou deux) interface(s) de création ;
- une (ou deux) interface(s) d'exposition des méthodes ;
- classes qui implémentent les interfaces ;
- un fichier de déploiement **ejb-jar.xml** ;
- un (ou plusieurs) fichier(s) de configuration - propre(s) au serveur d'application.

L'objet qui est codé par le développeur contient l'implémentation des méthodes métier. Cette classe doit implémenter une interface du type`javax.ejb.Sessionbean` ou `javax.ejb.EntityBean`. La spécification EJB 2 demande ensuite deux types d'interfaces pour l'exposition :

- des méthodes de création ;
- des méthodes métier.

N'oubliez pas que le cycle de vie de l'EJB est géré par le conteneur d'EJB. Le client de l'EJB ne l'instancie pas directement, il passe par les méthodes de création (`create()`, `findByPrimaryKey()`, etc) qui sont exposées via une interface, étendant l'interface`javax.ejb.EJBHome`.

De même, les appels vers les méthodes métier implémentées dans l'EJB passeront par l'interface, exposant les méthodes métier. Le développeur doit donc, déclarer les méthodes métier accessibles en les exposant dans une interface, étendant l'interface `javax.ejb.EJBObject`.

Les interfaces, qui étendent `javax.ejb.EJBHome` et`javax.ejb.EJBObject`, sont dites des interfaces distantes car les appels sur les méthodes exposées par ces interfaces sont effectués via RMI. Or, de nombreux appels sont effectués entre objets qui sont déployés sur le même serveur, donc dans la même machine virtuelle. Par exemple, une servlet d'une application Web qui utilise un EJB de session, ou un EJB de session qui utilise un EJB entité. Donc dans ce cas, l'invocation des méthodes via RMI est très couteuse, car complètement inutile. Pour y remédier, la spécification EJB 2 ajoute deux interfaces qui permettent de déclarer les méthodes de création et les méthodes métier, qui seront accessibles localement, au sein de la même machine virtuelle :

- l'interface `javax.ejb.EJBLocalHome` pour les méthodes du cycle de vie ;
- l'interface `javax.ejb.EJBLocalObject` pour les méthodes métier.

Le développeur doit donc :

- créer une classe d'implémentation pour implémenter une interface `javax.ejb.SessionBean` ou `javax.ejb.EntityBean`.
- exposer les méthodes du cycle de vie dans des interfaces qui permettront un accès :
  - distant via RMI (remote) en étendant l'interface `javax.ejb.Home` ;
  - au sein d'une même machine virtuelle (local) en étendant l'interface `javx.ejb.LocalHome`.
- exposer les méthodes métier dans des interfaces qui permettront un accès :
  - distant en étendant l'interface `javax.ejb.EJBObject` ;
  - local en étendant une interface `javax.ejb.EJBLocalObject`.

- écrire les descripteurs de déploiement de l'EJB. Ce fichier au format XML décrit les propriétés des EJB, qui constitueront l'archive. Ce fichier s'appelle le fichier ejb-jar.xml.

- écrire un fichier de déploiement spécifique au serveur, le fichier jboss.xml pour JBoss, qui décrit, entre autres, les noms JNDI des EJB.

Nous pouvons voir que l'écriture d'un EJB peut paraître assez complexe, bien que le véritable travail ne concerne qu'une seule classe : la classe d'implémentation. Heureusement, il existe des outils pour automatiser la création du composant :

- soit en utilisant des scripts Ant

- soit en utilisant les XDoclets, comme c'est le cas sous Éclipse Europa.

Vous trouverez en annexe les paramétrages à effectuer pour l'utilisation de XDoclet. Le développeur crée uniquement sa classe d'implémentation, ensuite, l'outil créera les interfaces et les fichiers XML nécessaires.

Les objets clients n'interagissent jamais avec la classe implémentant les interfaces. C'est le serveur qui gère, par l'intermédiaire de classes d'interpositions, le cycle de vie et les invocations des méthodes de l'EJB.

Il est important de noter que les accès aux EJB peuvent se faire de deux façons :

- par RMI, si le client de l'EJB est un objet déployé sur une machine virtuelle différente de celle de l'EJB ;

- sans RMI, si le client de l'EJB est dans la même machine virtuelle.

Il faut donc, très tôt, se demander comment sera déployée l'application. Si l'application Web est déployée sur le même serveur que la couche métier, il faut privilégier les appels sans RMI. Un appel par RMI est toujours plus coûteux en terme de performance (passage des paramètres par copie, sérialisation) qu'un appel direct en mémoire.

Il faut noter que sous JBoss, si le client et l'EJB sont dans la même machine virtuelle, même si le client appelle les méthodes via les interfaces distantes (donc par RMI), JBoss invoquera les méthodes via les interfaces locales, si elles existent.