



JavaEE c'est quoi ?

- <http://java.sun.com/javaee>
anciennement, J2EE (Java2 Enterprise Edition)
- Architecture / Modèle de programmation
- Spécifications
- Implémentation de référence
- Suite(s) de tests
- Label Java EE Sun (qualification de plateformes)

JavaEE

Au sommaire aujourd'hui

- Architecture JEE
 - 1. JavaEE Introduction
 - 2. JavaEE Architecture
 - 3. Des spécifications sous forme d'APIs
 - 4. Composants Web

Des objets aux composants d'entreprise

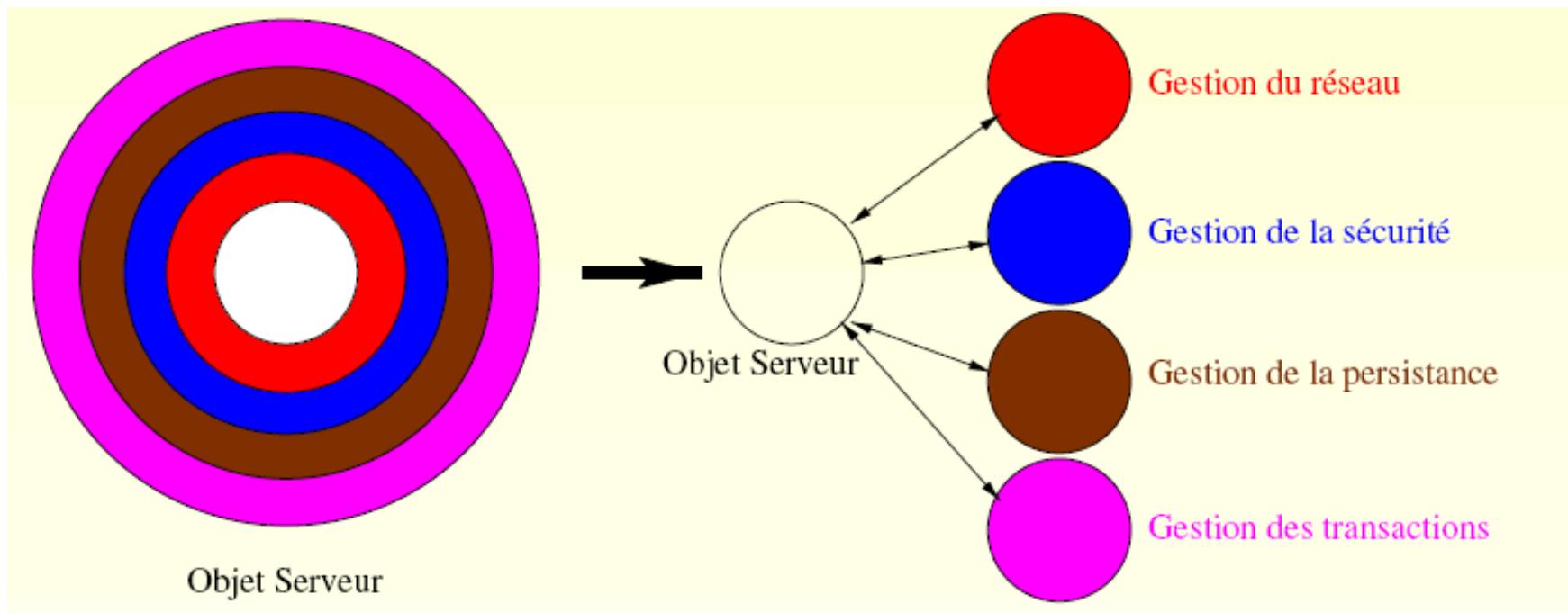
■ Des objets trop complexes

```
public class ObjetServeur extends UnicastRemoteObject {  
    private int x ;  
    public void setX(int value) throws RemoteException {  
        Tx.beginTransaction();  
        if (User.getAuthentification()...  
            this.x=value ;  
        // code JDBC : stockage de X  
        ...  
        Tx.commitTransaction();  
    }  
    public int getX() throws RemoteException {  
        Tx.beginTransaction();  
        if (User.getAuthentification()...  
        //code JDBC : restauration de X  
        ...  
        Tx.commitTransaction();  
        return this.x ;  
    }  
    public ObjetServeur() throws RemoteException { ... }  
}
```

Gestion du réseau
Gestion de la sécurité
Gestion de la persistance
Gestion des transactions

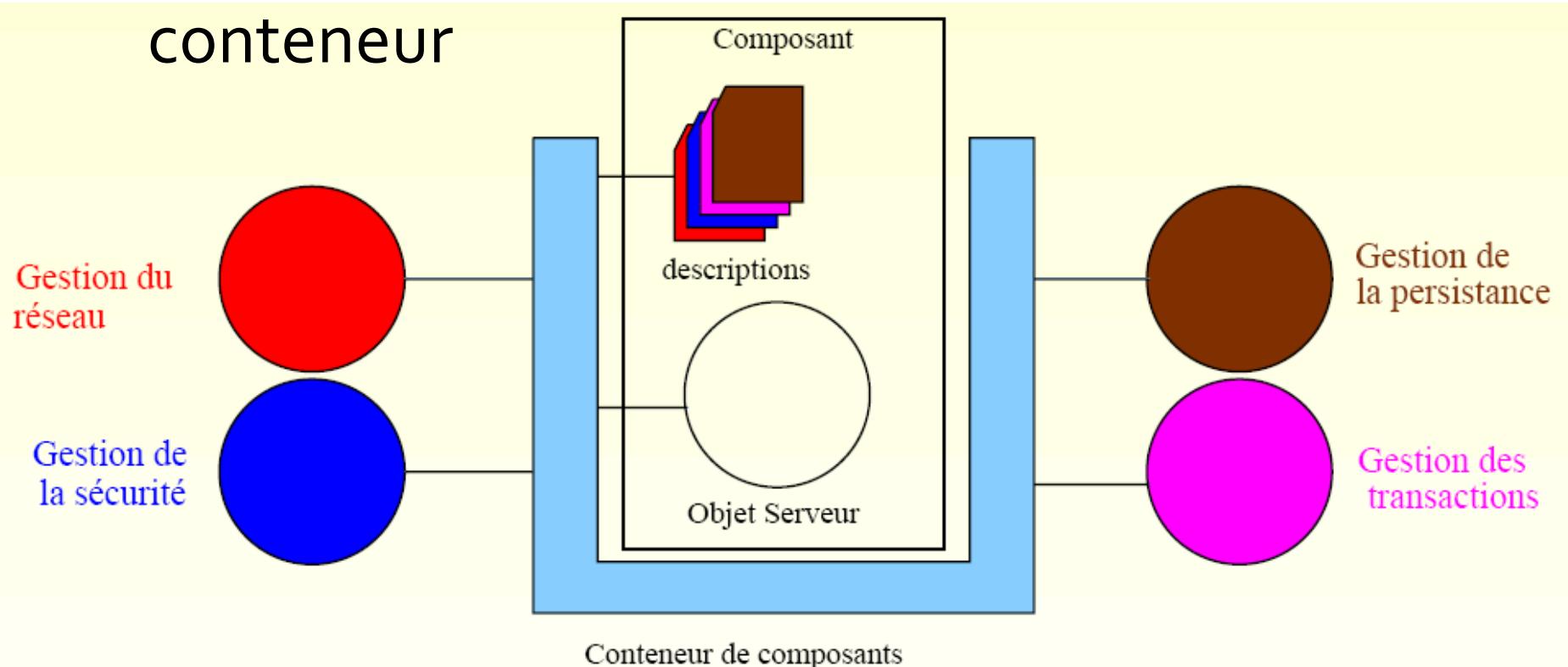
Des objets aux composants d'entreprise

- 1ère étape : une meilleure structuration objet



Des objets aux composants d'entreprise

- 2nde étape : complète séparation, notion de conteneur



Plan de la présentation

1. JavaEE Introduction
2. JavaEE Architecture
3. Des spécifications sous forme d'APIs
4. Composants Web
5. Composants d'entreprise (EJB)
6. Services techniques (du container)
7. Déploiement
8. Clients EJB
9. Développement J2EE
10. Design pattern pour EJB
11. Conclusion



JavaEE c'est quoi ?

- <http://java.sun.com/javaee>
- anciennement, J2EE (Java2 Enterprise Edition)
- Architecture / Modèle de programmation
- Spécifications
- Implémentation de référence
- Suite(s) de tests
- Label Java EE Sun (qualification de plateformes)

1. JavaEE Introduction

1. Java EE - Objectifs

- Faciliter le développement de nouvelles applications à base de composants
- Intégration avec les systèmes d'information existants
- Support pour les applications « critiques » de l'entreprise
 - Disponibilité, tolérance aux pannes, montée en charge, sécurité ...

1. Java EE - C'est quoi?

- <http://java.sun.com/javaee>
anciennement, J2EE (Java2 Enterprise Edition)
- Architecture / Modèle de programmation
- Spécifications
 - Programmation, assemblage, déploiement
 - Serveur et services
- Implémentation de référence opérationnelle
- Suite(s) de tests de conformance
 - Certification Sun
 - Accès au processus de certification payant (cher !!)
 - Lourd (> 20.000 tests)
- Label Java EE Sun (qualification de plateformes)

1. Offre commerciale

- BEA WebLogic (haut de gamme)
- IBM Websphere (no 1)
- Sun Java System App Server
- Borland Enterprise Server
- Oracle Application Server
- Macromedia jRun
- SAP Web application server
- Iona Orbix E2A
- ...

1. Offre open-source

- JBoss (no 1 en nombre de déploiements)
- ObjectWeb JOnAS (no 2, intégré à plusieurs distro Linux Entreprise)
- Sun JS App Server (Platform Edition)
- Apache Geronimo (démarrage fin 2003)
- openEjb
- ejBean

1. Evolution de Java EE

- Standard en évolution depuis 1997
 - J2EE 1.0 à 1.4 en 2003, etc...
- Au départ, applications Web n-tiers
 - Présentation (Servlets puis JSP), essentiellement HTTP
 - Logique métier : EJB
 - Données : JDBC
- Puis infrastructure de support standard pour EAI
 - Facteurs de rationalisation majeurs (JTA, JMS, JCA, Web Services)
 - Evolution de progiciels existants vers Java EE

1. Vers Java EE 5

- Simplification du développement
 - Utilisation des annotations amenées par Java SE 5
 - Relâchement des contraintes / programmation des composants
- Enrichissement fonctionnel
 - Amélioration du support du tiers de présentation avec JSF
 - Support complet des Web Services
 - Sessions exposables avec RMI ou avec Web Services
 - Amélioration du support des objets persistants avec Java
 - Persistence API (un grand pas vers JDO !!!)

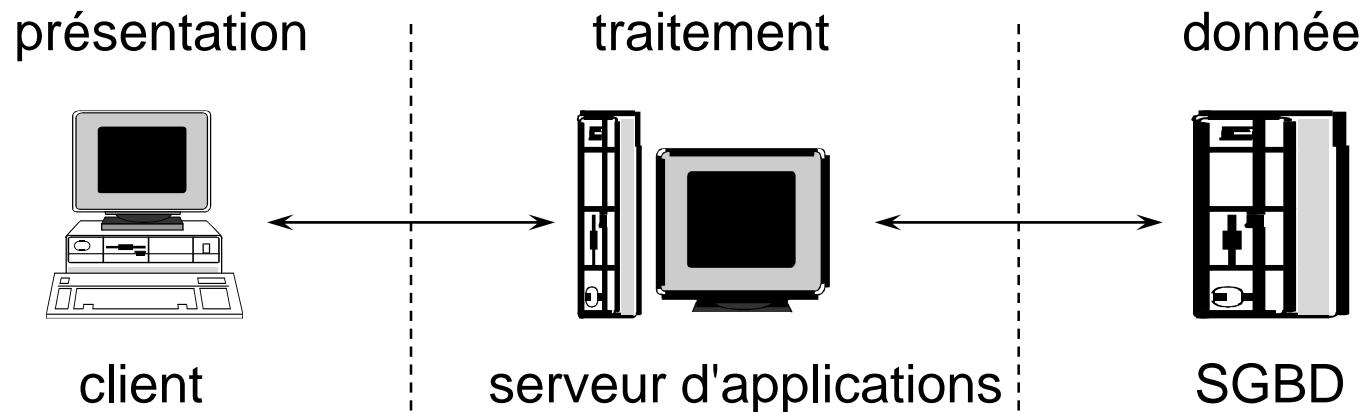
2. Java EE - Architecture

JavaEE c'est quoi ?

- <http://java.sun.com/javaee>
anciennement, J2EE (Java2 Enterprise Edition)
- **Architecture / Modèle de programmation**
- Spécifications
- Implémentation de référence
- Suite(s) de tests
- Label Java EE Sun (qualification de plateformes)

2. Java EE - Architecture

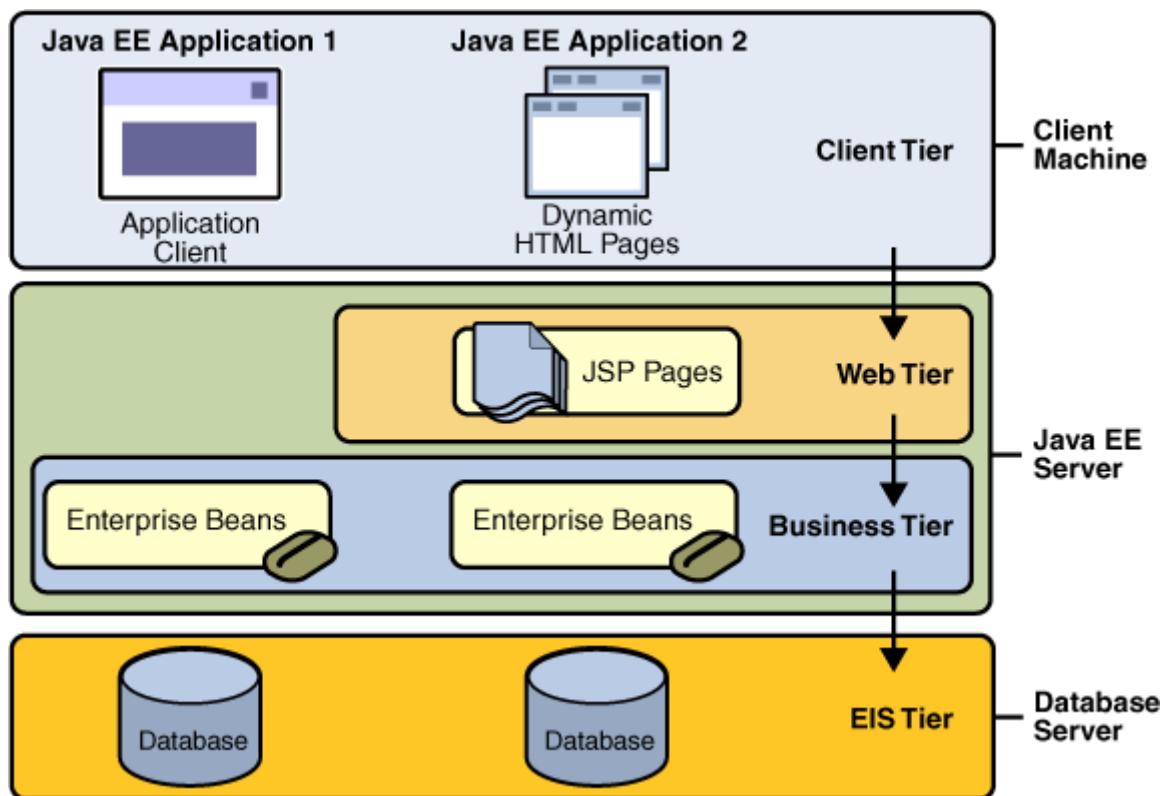
Architecture 3 tiers



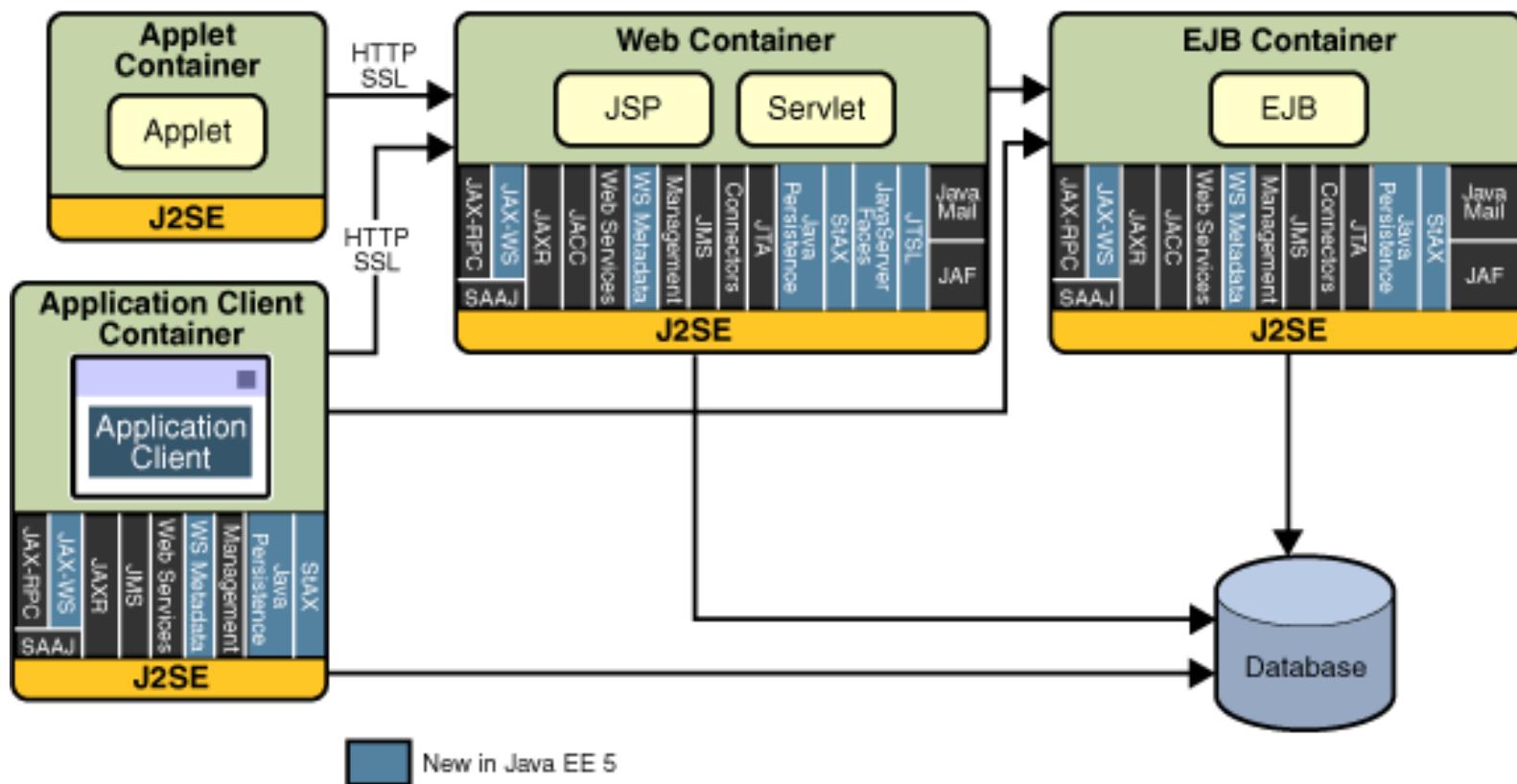
La technologie EJB réutilise un nombre important de librairies Java

- RMI-IIOP : invocation de méthodes distantes
- JNDI : accès à un service de nommage
- JDBC : connexion à des bases de données
- JTS : gestion des transactions (spec. basées sur CORBA OTS)
- JMS : communications asynchrones
- JSP/servlet : clients web

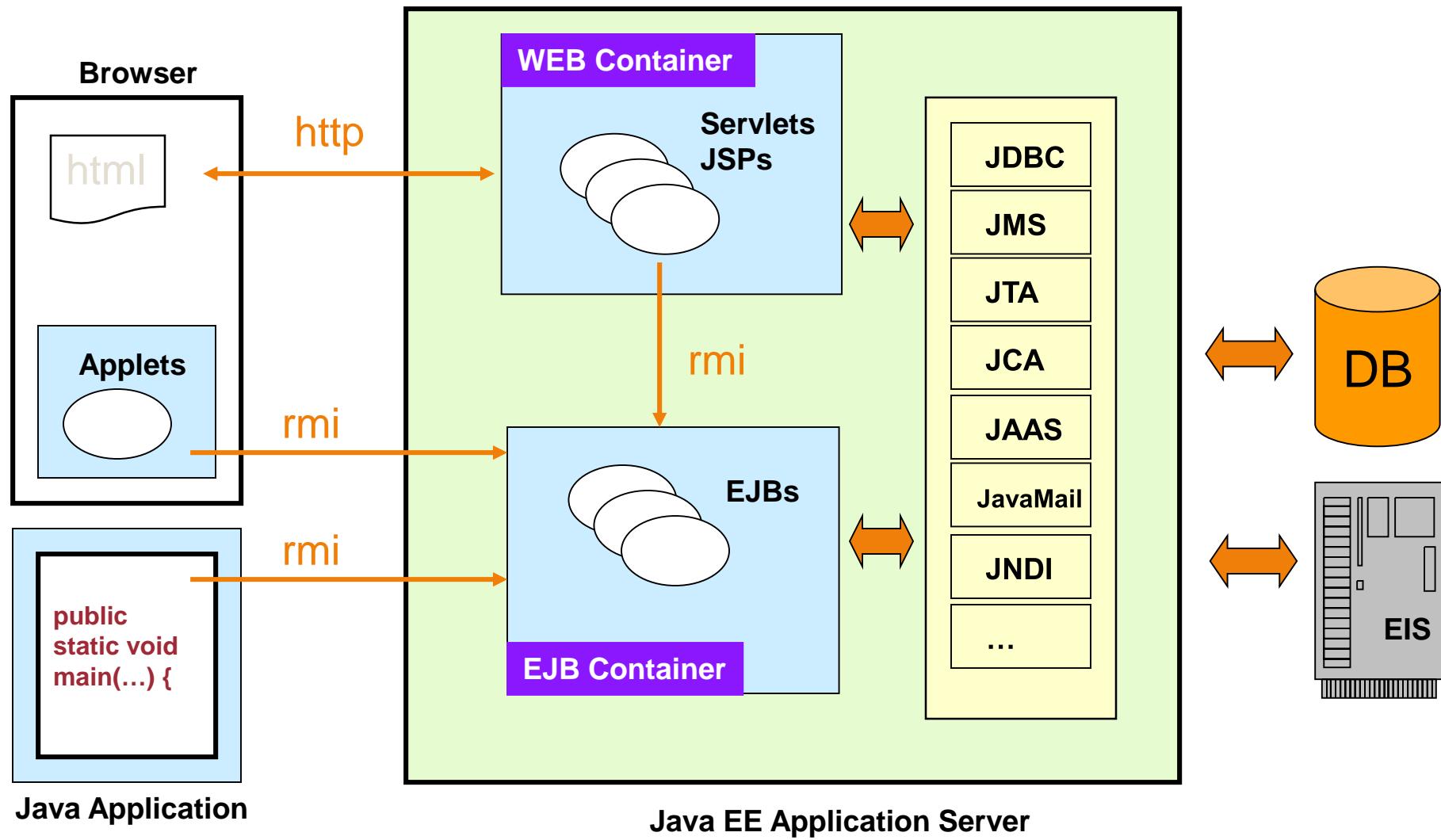
2. JEE Distributed Multitiered Application



2. Java EE Architecture



2. Java EE - Architecture



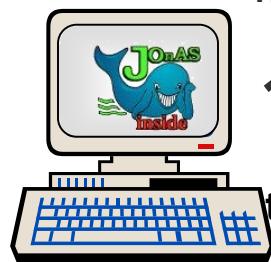
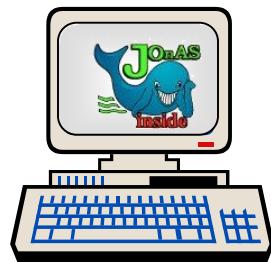
2. Architecture multi-tiers

- Client
 - Léger (Web, browser)
 - Lourd (Application java, Applet...)
 - Architecture orientée service (Application répartie sans présentation)
- Serveur d 'applications
 - Conteneur EJB + logique métier
 - Services non fonctionnels
- EIS (*Executive Information System*) ou Base de données

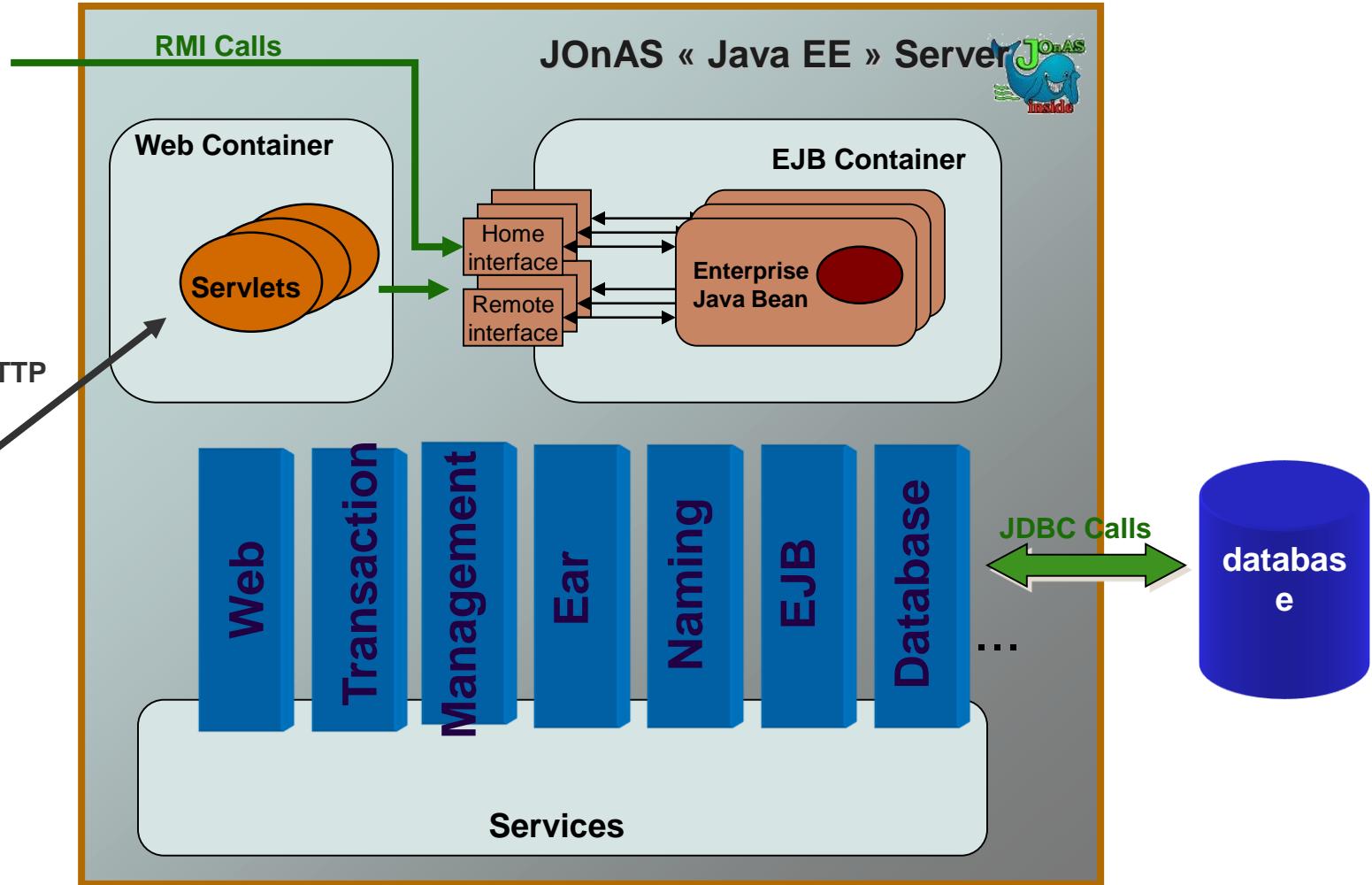
2. Un serveur Java EE

Source : Bull/ObjectWeb (JOnAS)

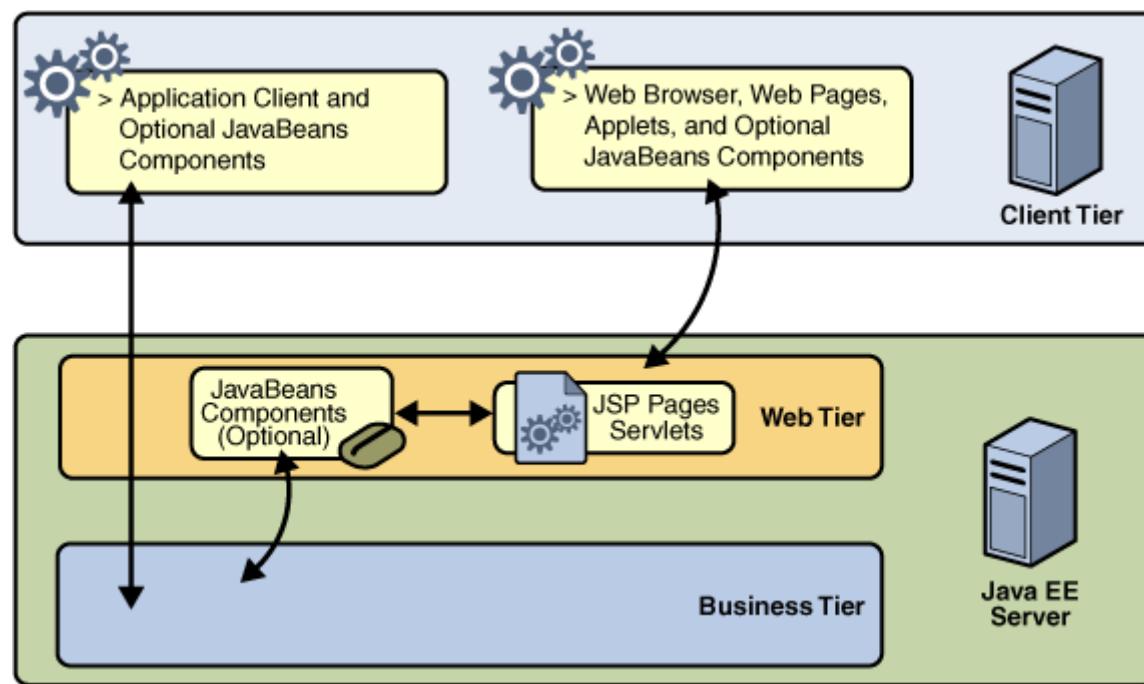
Java Client



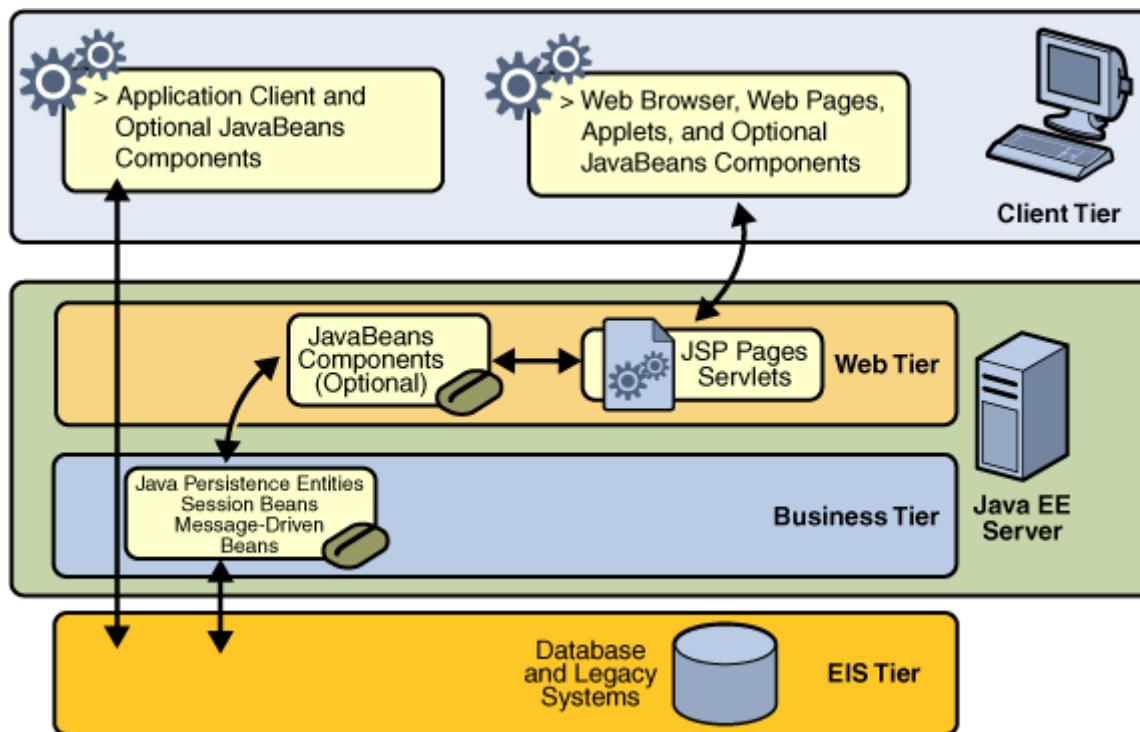
HTTP



2. Web-Tier overview



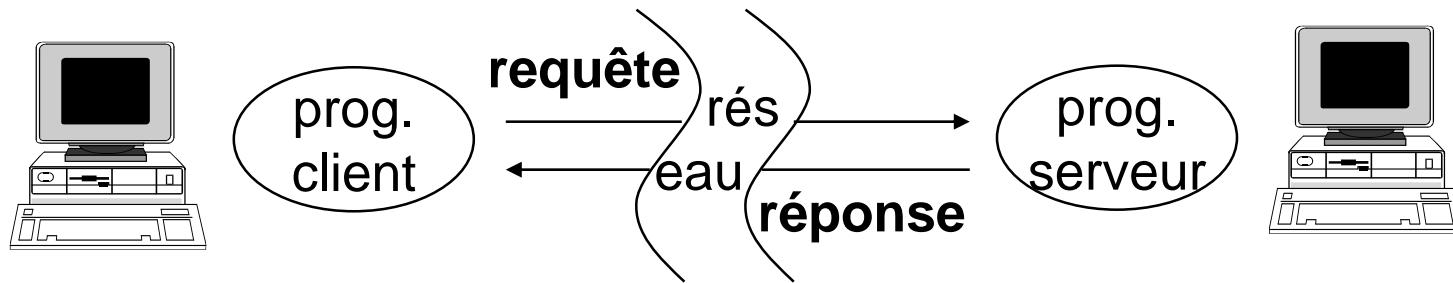
2. Business-Tier overview



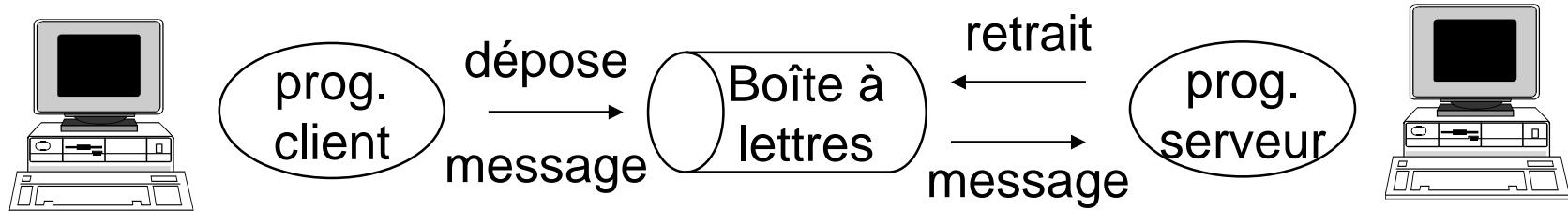
2. Java EE - Communications

Communications distantes

requête/réponse



orientées message (MOM : Message-Oriented Middleware)



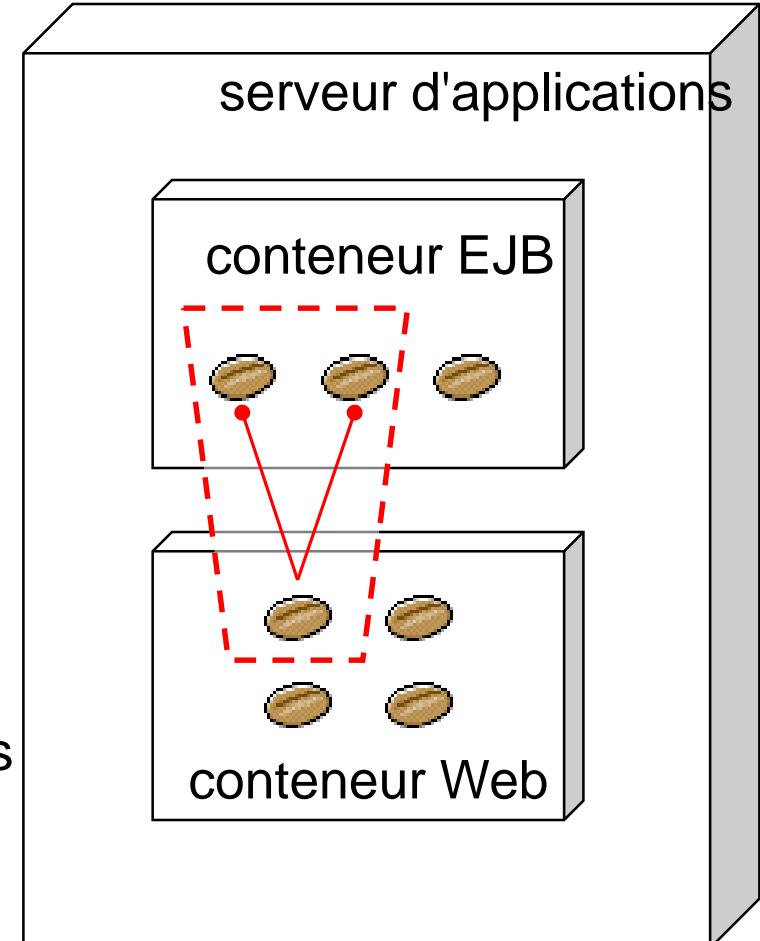
2. Serveur d'applications EJB

Application EJB =

- 0, 1 ou + sieurs composants EJB
- 0, 1 ou + sieurs composants Web
- reliés par un schéma d'assemblage

+ sieurs rôles dans le développement

- développeur de composants Web
- développeur de composants EJB
- assembleur d'applications
- déployeur et gestionnaire d'applications



2. Serveur d'applications EJB

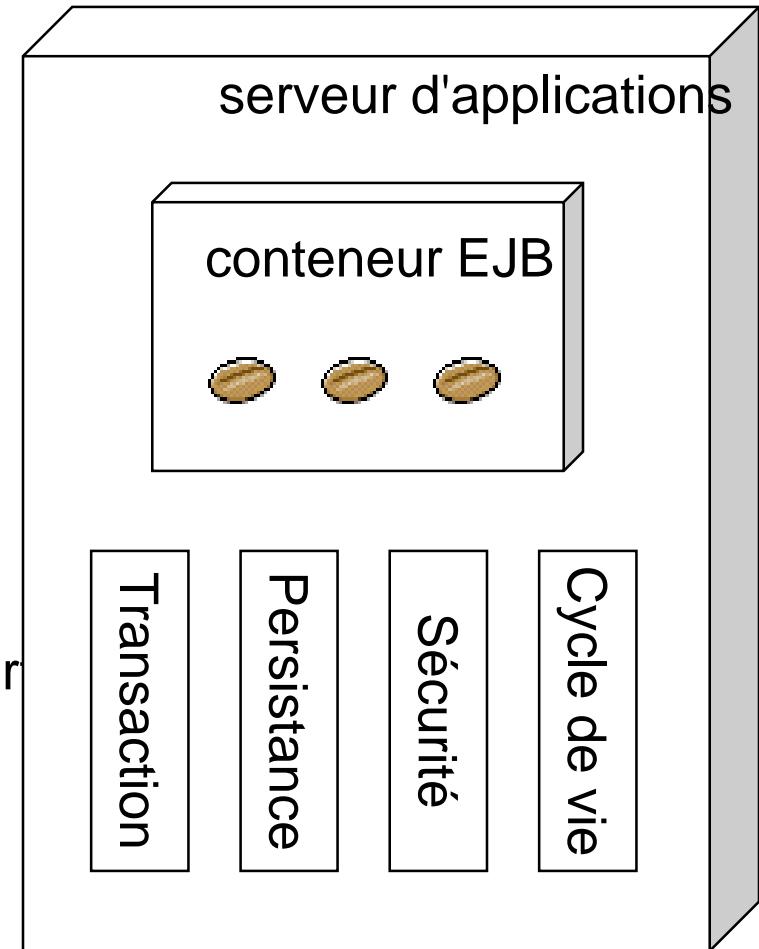
Serveur d'applications EJB

4 services fournis par le serveur au conteneur EJB

- transaction
- persistance
- sécurité
- cycle de vie

≠ *middleware style CORBA*

ces services sont intégrés dès le départ à la plate-forme



4. Des spécifications sous forme d'APIs

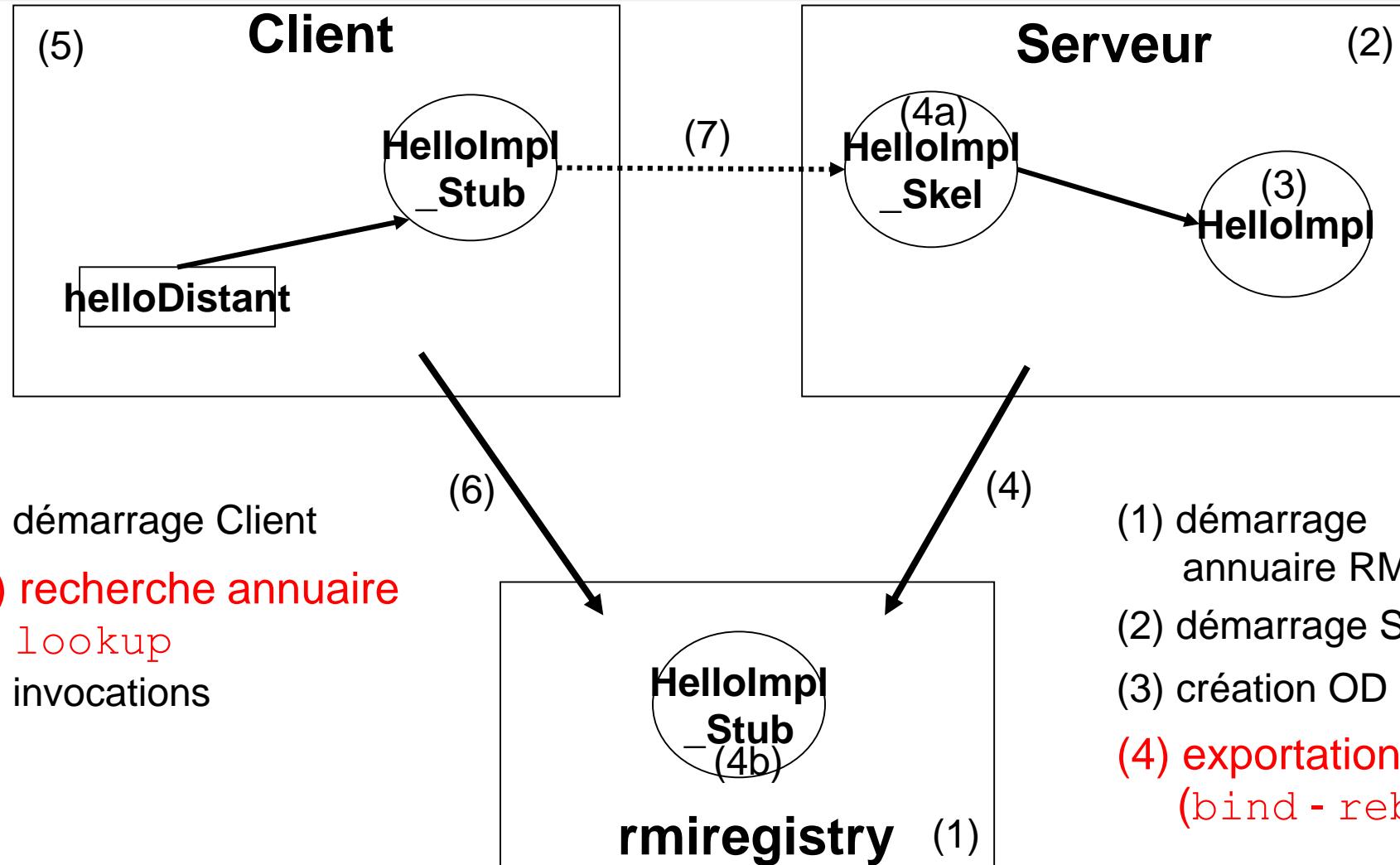
JavaEE c'est quoi ?

- <http://java.sun.com/javaee>
anciennement, J2EE (Java2 Enterprise Edition)
- Architecture / Modèle de programmation
- Spécifications
- Implémentation de référence
- Suite(s) de tests
- Label Java EE Sun (qualification de plateformes)

3.1. RMI

- Remote Method Invocation
 - Java seulement, mais passerelles
- « RPC objet » (appels sur objets distants)
- Service de nommage (RMI registry)
- Sécurité paramétrable (SecurityManager)
- Garbage Collection distribuée
- Téléchargement de code
- Fonctions avancées
 - Activation d'objets persistants, RéPLICATION

3.2 JNDI le besoin d'un service de nommage



3.2. JNDI - Concepts de base

■ La notion de nommage:

- Un service de nommage permet d'associer un nom unique à un objet et faciliter ainsi l'obtention de cet objet.
- L'identification doit être aisée et précise.
- La localisation et l'accès à une ressource sont donc facilitées par le nommage.
- Beaucoup de services de nommage sont étendus avec un service d'annuaire.
- Exemple: Internet et son système de nommage, le DNS (*Domain Name System*)

3.2. JNDI - Concepts de base

- Concept de gestion des ressources par annuaire:
 - L'annuaire est un service de nommage.
 - Il possède en plus une représentation hiérarchique des objets contenus.
 - Il possède un mécanisme de recherche permettant de rendre transparent l'accès aux ressources pour l'utilisateur.
 - Exemple: Le service de gestion d'annuaire LDAP (*Lightweight Directory Access Protocol*).

3.2. JNDI - Concepts de base

■ La concept de contexte:

- Un contexte est un ensemble de liens nom-objet.
- Chaque contexte a une convention de nommage associé.
- Il fournit un moyen de consultation (résolution) qui, reçoit l'objet et peut fournir d'autres opérations telles que:
 - La liaison de nom (Binding)
 - La suppression de cette liaison (unbinding)
- Un nom peut-être lié à un objet de contexte, appelé alors sous-contexte.
- JNDI manipule des contextes avec l'interface *context*.

3.2. JNDI - Présentation

- JNDI (Java Naming and Directory Interface)
- JNDI est une spécification qui fournit une interface unique pour utiliser différents services de noms ou d'annuaires.
- Ces services peuvent être:
 - *LDAP (Lightweight Directory Access Protocol)*
 - *DNS (Domain Naming Service)*
 - *NIS (Network Information Service) de SUN*
 - *Service de nommage CORBA*
 - *Service de nommage RMI, etc...*

3.2. Quand utiliser JNDI ?

- JNDI est un composant important de J2EE que plusieurs technologies comme les EJB, JDBC ou JMS utilisent.
- On l'utilise pour localiser les objets Java sur un serveur et accéder aux objets externes d'une façon portable.

3.2. Pourquoi JNDI ?

- A mesure que l'utilisation du langage Java pour développer des applications distribuées dans un environnement réseau augmente, la faculté d'accéder à des services d'annuaire devient essentielle.
- En effet, l'utilisation d'un service d'annuaire permet de simplifier les applications et leur administration en centralisant le stockage d'information partagée.
- C'est pourquoi JNDI est un outil de plus en plus utilisé par les développeurs.

3.2. JNDI

- API accès aux annuaires
 - javax.naming
 - « Service Provider » par annuaire cible (LDAP, NIS, RMI registry...)
- Utilisation avec les EJB
 - Accès à l'interface « home » pour initialiser
 - Accès à diverses ressources (UserTransaction, Queues JMS, DataSources...)

3.3. JMS

- Java Messaging Service
- JMS Provider : inclus dans J2EE
 - Transport synchrone ou asynchrone, Garantie de livraison
 - « Messaging domains » point à point ou « publish/subscribe »
- Lien avec EJB : « message-driven bean »
 - Pour échanges asynchrones

3.4. API JavaEE de transactions : JTA

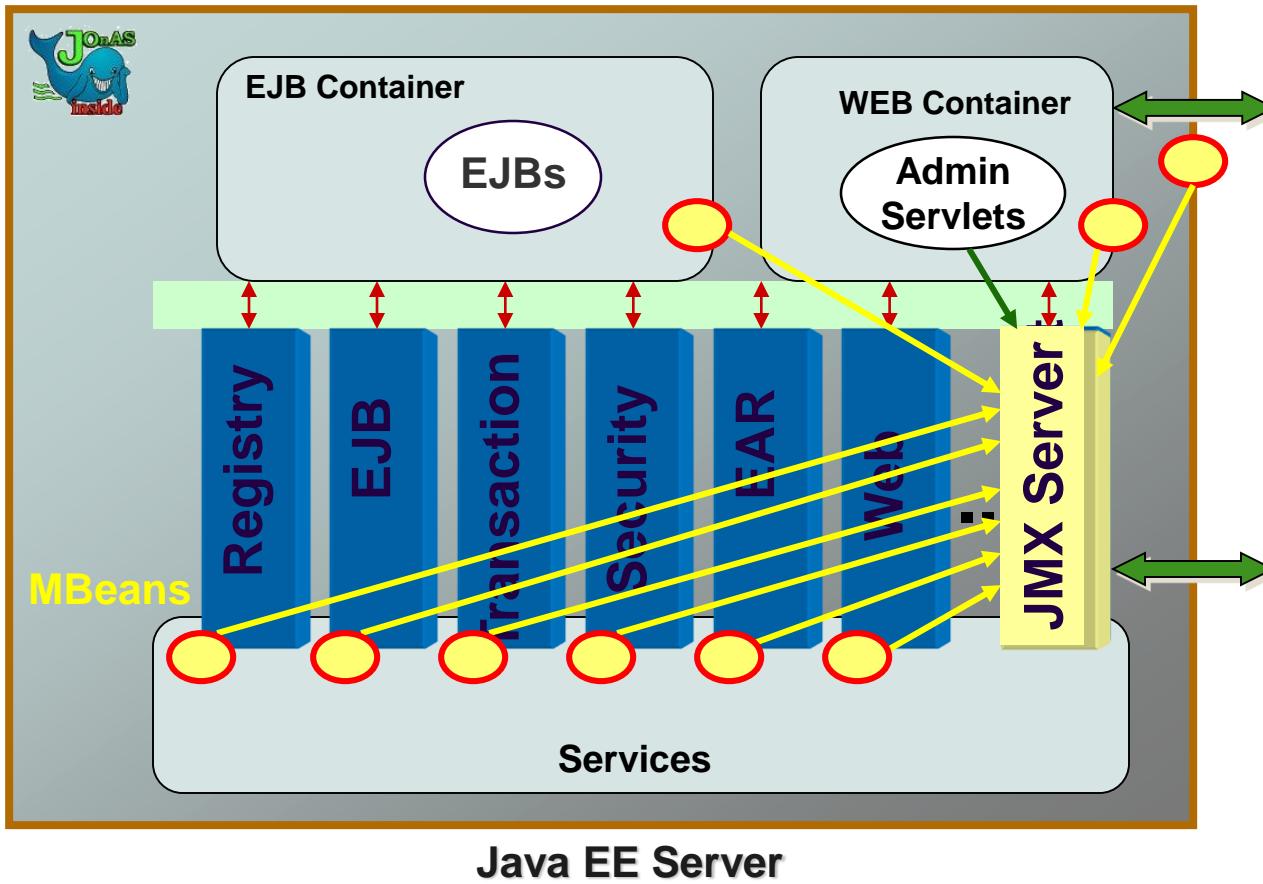
- Java Transaction API
- Package javax.transaction
 - TransactionManager : begin(), commit(), rollback() ...
 - Transaction : commit(), rollback(), enlistResource(XAResource), registerSynchronisation(Synchronization) ...
 - Synchronization : beforeCompletion(), afterCompletion(commit | rollback)

3.5. JMX

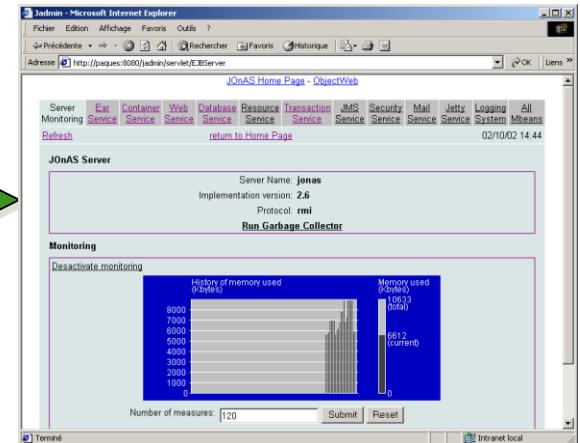
- Java Management eXtensions
 - API unique pour applications de management
- Mbeans avec accesseurs get/set
 - Typage faible, attributs nommés
- Serveur JMX
 - Enregistrement des Mbeans
 - Les applis d 'administration dialoguent avec le serveur JMX
- Instrumenter un composant
 - Fournir un ou des Mbeans
 - Les enregister auprès du serveur JMX

3.5. JMX : Exemple d'un serveur JavaEE

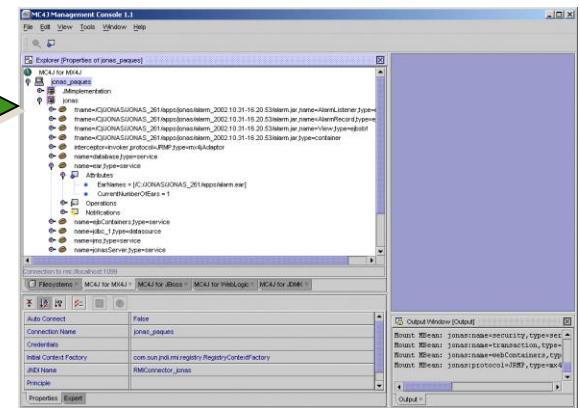
Source : ObjectWeb JOnAS



Admin console



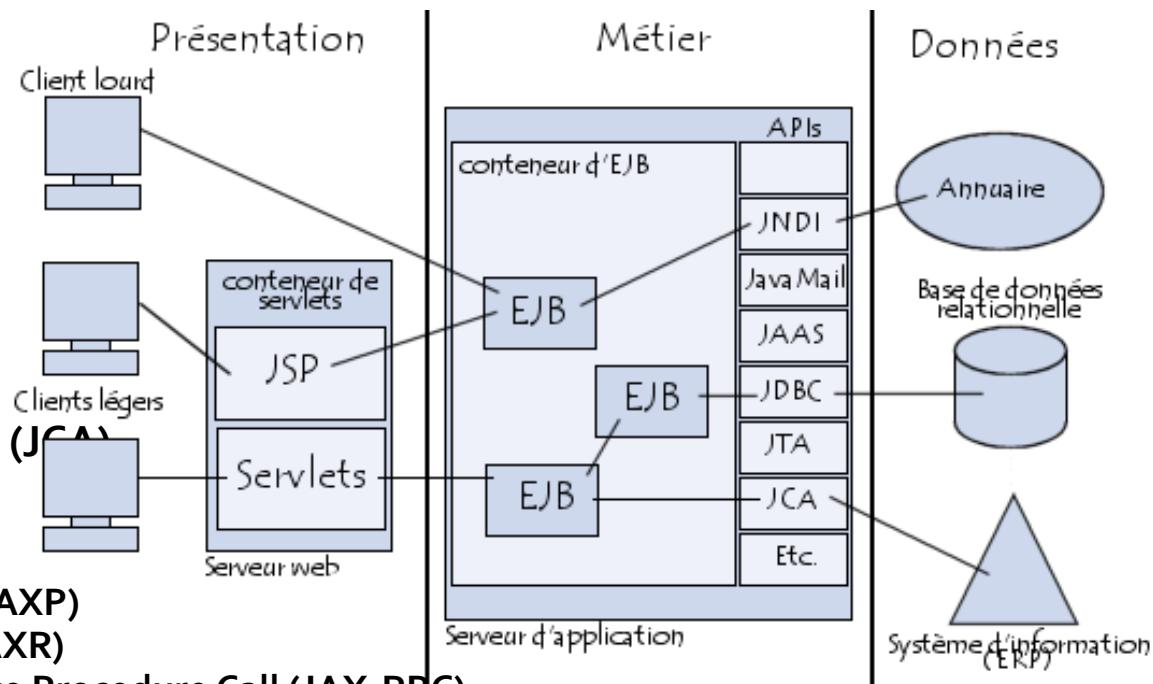
MC4J



3.6. Les APIs de JavaEE

(standards connexes)

- Java Servlets
- JavaServer Pages (JSP)
- Enterprise JavaBeans
- Java Message Service (JMS)
- JDBC - JPA
- Transactions
 - JTA
 - JTS
- J2EE Connector Architecture (JCA)
- Corba (Java IDL)
- JavaMail
- XML/SOAP
 - Java API for XML Processing (JAXP)
 - Java API for XML Registries (JAXR)
 - Java API for XML-Based Remote Procedure Call (JAX-RPC)
 - SOAP with Attachments API for Java (SAAJ)
- J2EE Deployment Specification (JSR-88)
- J2EE Management Specification (JSR-77)
- JMX



Java EE : Composants Web



Définition d'une Servlet

- Composant logiciel écrit en Java s'exécutant du côté serveur. Traite des requêtes HTTP et fournit aux clients des réponses HTTP.
- Comparable à :
 - CGI (« Common Gateway Interface »),
 - langages de script côté serveur PHP, ASP...
- Une Servlet s'exécute dans un « moteur de Servlet » (« conteneur de Servlet ») permettant d'établir le lien entre la Servlet, la machine virtuelle java et le serveur Web.

Ex : Tomcat, Weblogic, Glassfish...

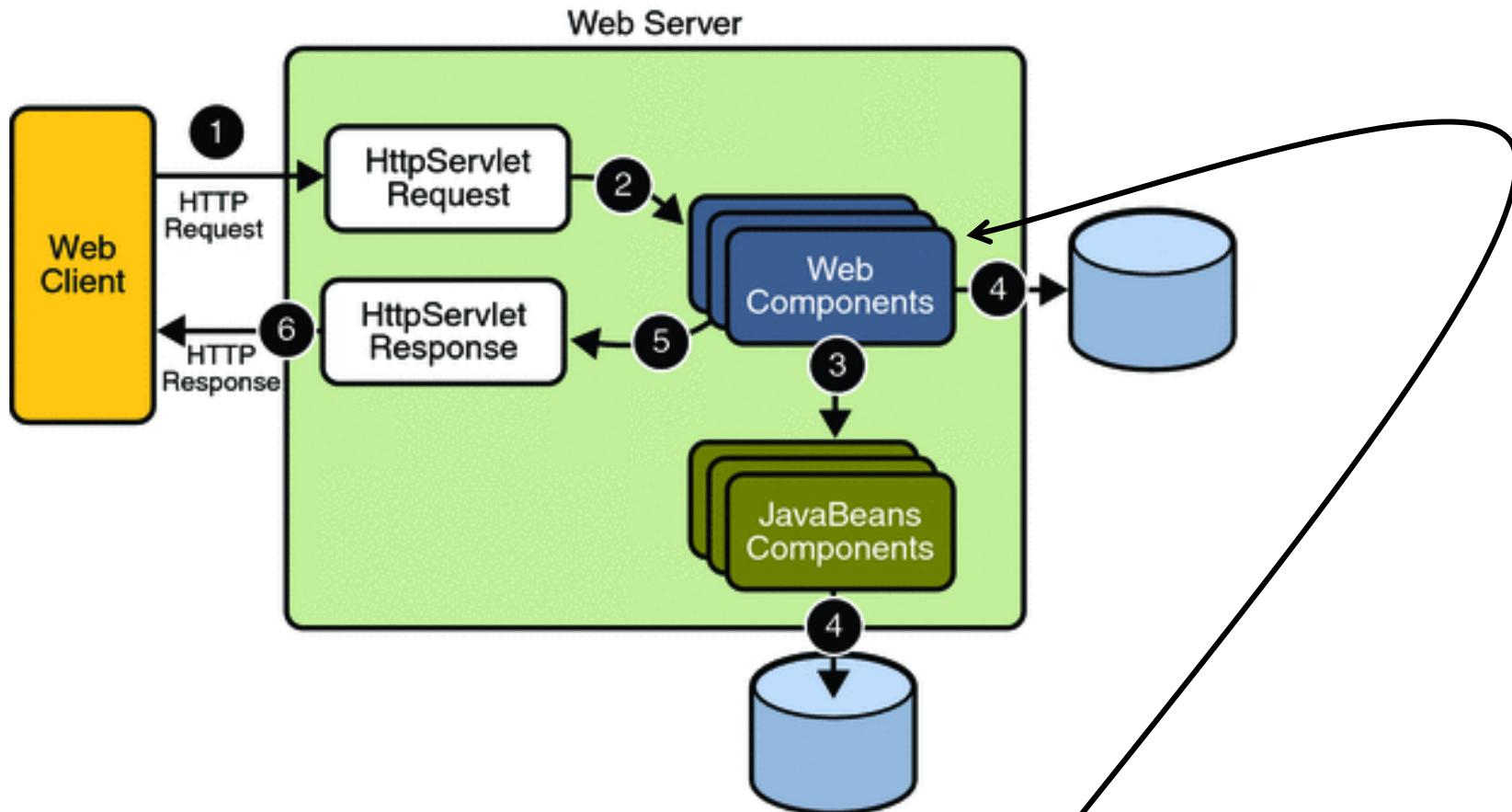
Utilité des servlets

- Créer des application Web dynamiques (pages dynamiques).
- Effectuer des traitements applicatifs coté serveur Web et bénéficier de leur puissance de calcul et de l'accès aux bases de données.
- Écrire une application Java dont l'interface utilisateur est dans le client :
 - navigateur,
 - applet,
 - téléphone portable...

Avantages des servlets

- **Portabilité**
 - Technologie indépendante de la plate-forme et du serveur.
 - Un langage (Java) et plusieurs plates-formes.
- **Puissance**
 - Disponibilité des API de Java.
 - Manipulation d'images, connectivité aux bases de données (JDBC)...
- **Efficacité**
 - Utilisent des threads (processus légers) plutôt que des processus système comme les CGI
- **Sûreté**
 - Typage fort de Java.
 - Gestion des erreurs par exceptions.
 - Fonctionnement dans une machine virtuelle.
- **Faible coût**
 - Nombreux serveurs gratuits.

Architecture



Une servlet peut être appelée par un client alors qu'elle s'exécute déjà pour un autre client. L'objet servlet unique s'exécute alors dans un autre thread. Certaines parties du code doivent être synchronisées avec les techniques prévues en Java pour ce faire.

Une première servlet

- Dérive de *HttpServlet*.
- Implante au moins une des méthodes : *doGet*, *doPost*, *doPut*, *doDelete*...

```
public class HelloWorld extends HttpServlet {  
  
    @Override  
    protected void doGet(HttpServletRequest request,  
                         HttpServletResponse response)  
        throws ServletException, IOException {  
        PrintWriter out=response.getWriter();  
        out.println("Hello World");  
    }  
  
}
```

Code java
de la
servlet



http://localhost:8084/CoursWeb/hello

Appel de la servlet (URL ou lien)

Hello World

Résultat affiché
dans le
navigateur

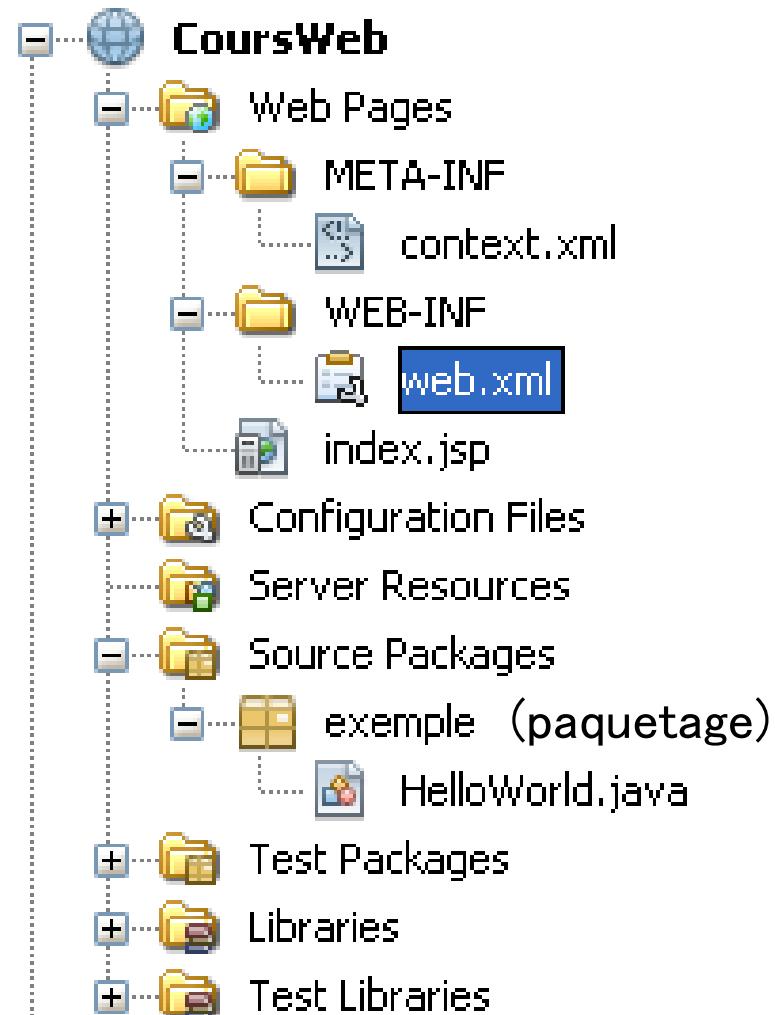
- Plus couramment la réponse au client est une page HTML

:

```
out.println("<HTML>");  
out.println("<HEAD><TITLE>Bonjour tout le  
monde</TITLE></HEAD>");  
out.println("<BODY>");  
out.println("<BIG>Hello World</BIG>");  
out.println("</BODY></HTML>");
```

Développement de l'application web

- Sources java
 - + ressources statiques (ex: images)
 - + librairies (.jar)
 - + fichier de déploiement (*web.xml*)
- Création d'une archive *.war* (Web Application ARchive) à déposer sur le serveur (ex : dans le répertoire *webapps* de Tomcat)



Fichier de déploiement : *web.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation = "http://java.sun.com/xml/ns/javaee
                        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" >

    <servlet>
        <servlet-name>HelloWorld</servlet-name>
        <servlet-class>exemple.HelloWorld</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>HelloWorld</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>
</web-app>
```

The diagram illustrates the deployment descriptor *web.xml* with annotations explaining the meaning of specific XML elements:

- nom de la servlet**: Points to the *servlet-name* element within a *servlet* declaration.
- chemin de la classe**: Points to the *servlet-class* element within a *servlet* declaration.
- URL d'appel**: Points to the *url-pattern* element within a *servlet-mapping* declaration.

Une application web peut comporter plusieurs servlets.

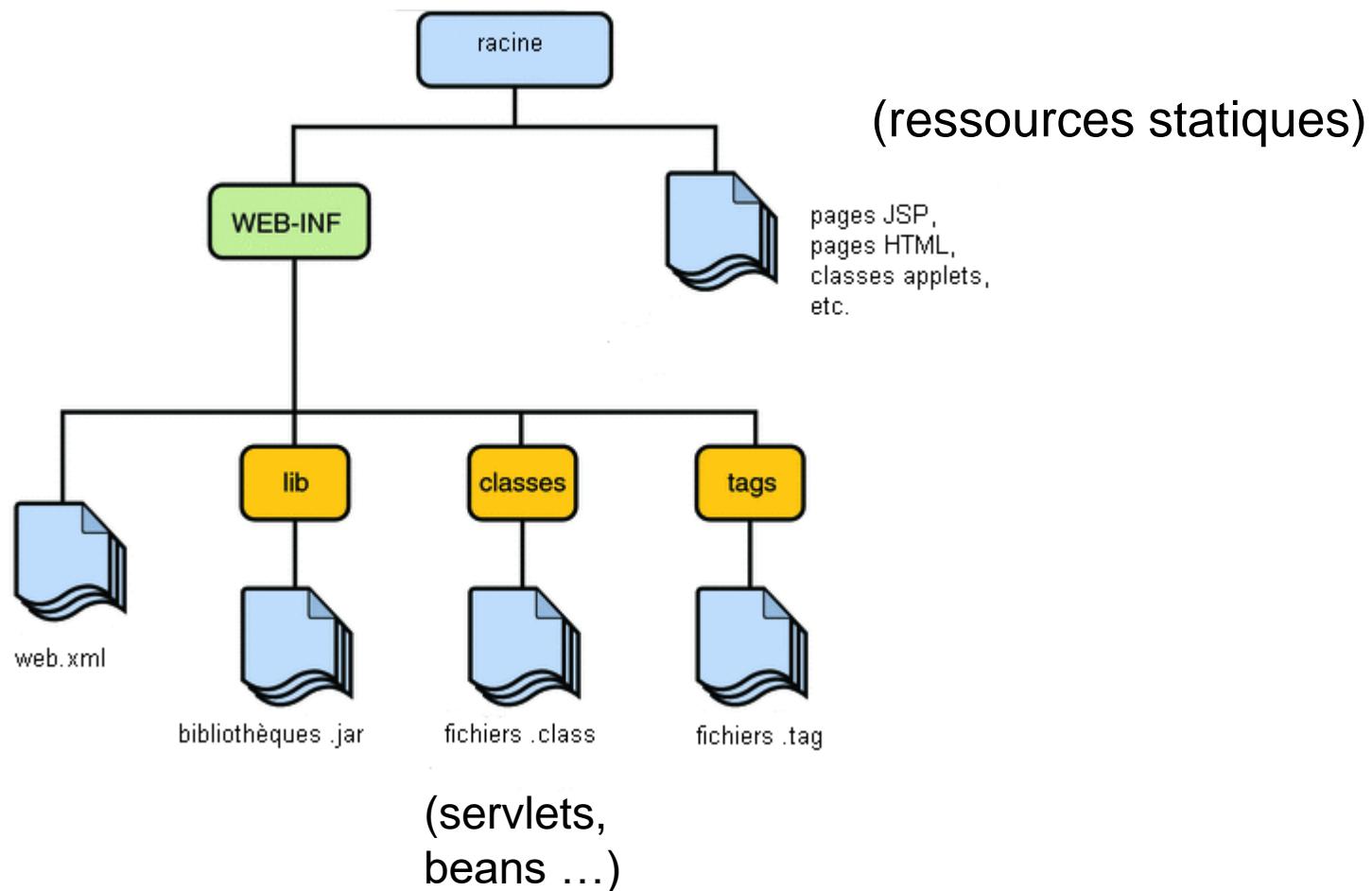
Déploiement du .war sur le serveur

Après déploiement
(automatique) :

Fichier ZIP

(nom du fichier .war)

(ressources statiques)

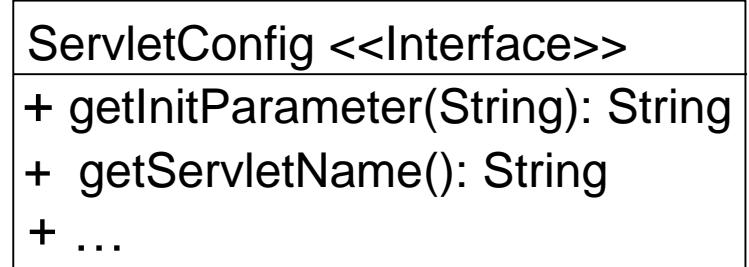
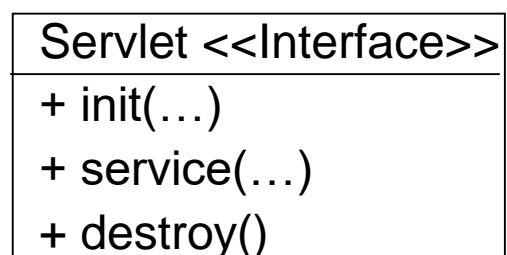


L'API Servlet

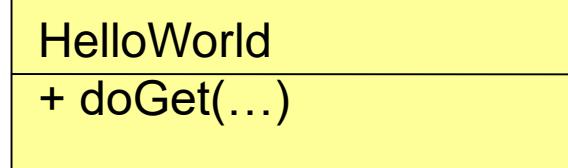
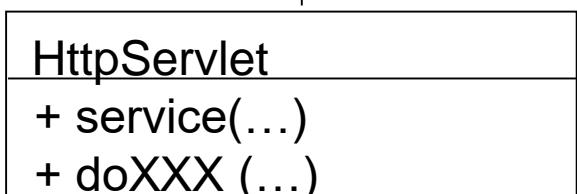
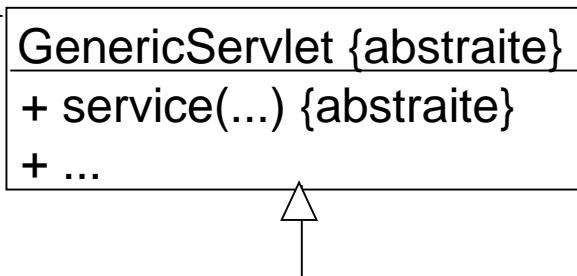
- Une Servlet doit implémenter les interfaces *javax.servlet.Servlet* et *javax.servlet.ServletConfig*
- L'API fournit 2 classes proposant une implantation :
 - *GenericServlet* pour la conception de Servlets indépendantes du protocole.
 - *HttpServlet* pour la conception de Servlets spécifiques au protocole HTTP. C'est cette forme qui est presque toujours utilisée.

Un bref rappel sur le protocole HTTP ('HyperText Transfer Protocol') avant de détailler l'API Servlet.

- HTTP décrit les échanges entre navigateur Web (client) et serveur Web : le navigateur effectue une requête HTTP; le serveur la traite et envoie une réponse HTTP.
- Une requête HTTP a le format suivant :
 - Ligne de commande (Commande, URL, Version de protocole)
 - En-tête de requête
 - [Ligne vide]
 - Corps de requête
- Une réponse HTTP a le format suivant :
 - Ligne de statut (Version, Code-réponse, Texte-réponse)
 - En-tête de réponse
 - [Ligne vide]
 - Corps de réponse
- Les commandes essentielles sont GET (obtenir un document) et POST (envoyer du contenu). Les données de la requête sont passées dans l'URL pour un GET et dans le corps pour un POST.



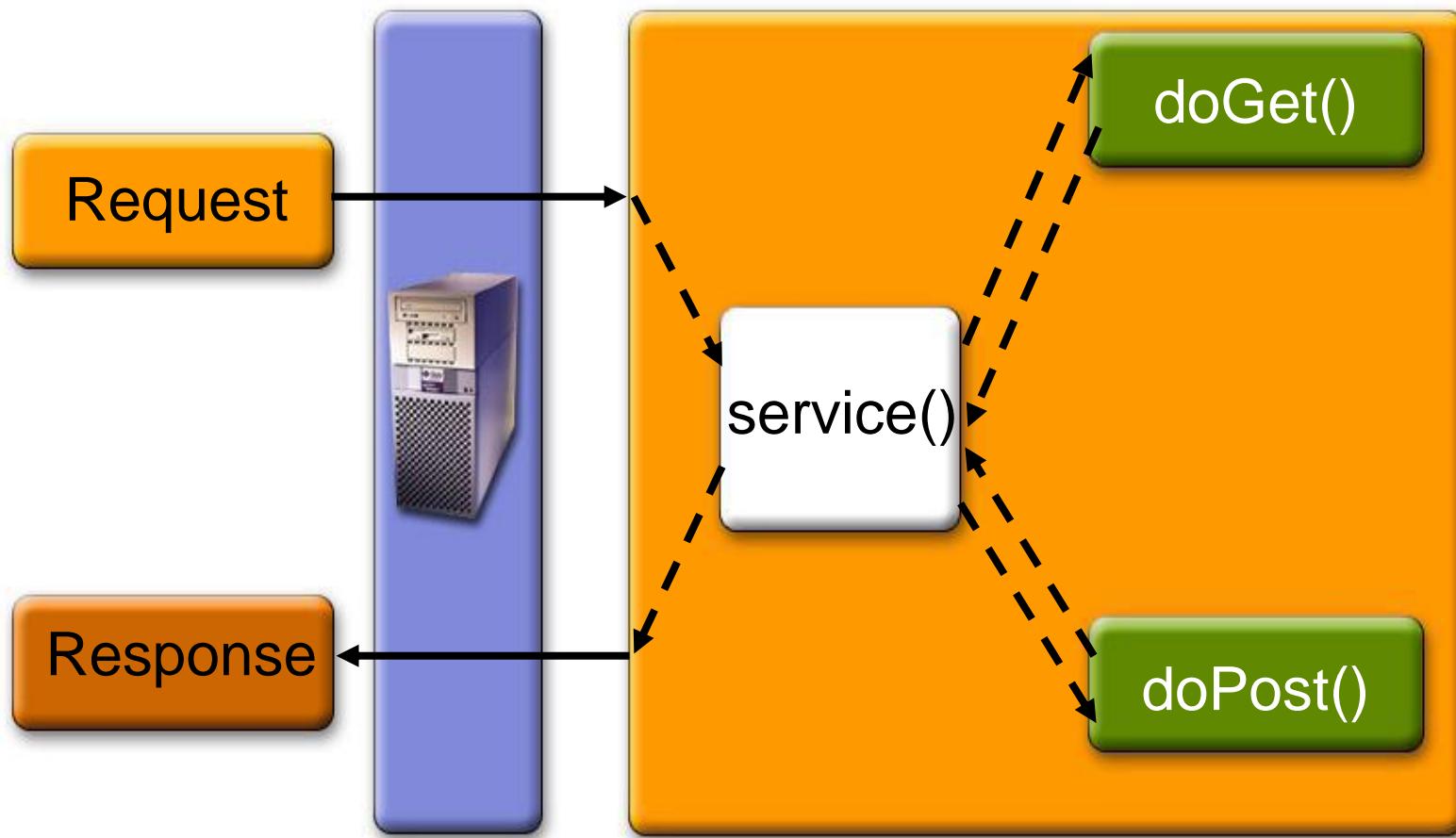
HttpServlet définit la méthode *service* qui appelle *doXXX* selon la requête HTTP reçue (*doGet*, *doPost*...). L'implantation par défaut de *doXXX* rend une erreur HTTP 405 (méthode non supportée).



Dériver *HttpServlet* pour construire des Servlets spécifiques au protocole HTTP en redéfinissant *doGet*, *doPost*...

Serveur

Sous classe HttpServlet



HttpServletRequest & HttpServletResponse

La méthode *service()* et les méthodes de traitement de requêtes *doGet()*, *doPost()* ont en paramètre :

- un objet requête (*HttpServletRequest*),
- un objet réponse (*HttpServletResponse*).

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SampleServlet extends HttpServlet {
    public void doGet(HttpServletRequest req,
                      HttpServletResponse res)
        throws ServletException, IOException {
        ...
    }
    public void doPost(HttpServletRequest req,
                      HttpServletResponse res)
        throws ServletException, IOException {
        ...
    }
}
```

HttpServletRequest encapsule la requête HTTP et fournit des méthodes pour accéder aux informations sur la requête du client au serveur.

Méthodes :

- *String getMethod()* : retourne le type de requête
- *String getServerName()* : retourne le nom du serveur
- *String getParameter(String name)* : retourne la valeur d'un paramètre
- *Enumeration getParameterNames()* : retourne le nom des les paramètres
- *String[] getParametersValues()* : retourne les valeurs des paramètres
- *String getRemoteHost()* : retourne l'IP du client
- *String getServerPort()* : retourne le port sur lequel le serveur écoute
- *String getQueryString()* : retourne la chaîne d'interrogation ... (voir l'API Servlets pour le reste)

HttpServletResponse est utilisé pour construire le message de réponse HTTP renvoyé au client. Il contient :

- les méthodes nécessaires pour définir le type de contenu, en-tête et code de retour,
- un flot de sortie pour envoyer des données (par exemple HTML) au client.

Méthodes :

- *java.io.PrintWriter getWriter()* : pour récupérer un PrintWriter qui permet d'envoyer du texte au client
- *void setStatus (int)* : définit le code de retour de la réponse
- *void setContent-Type (String)* : définit le type de contenu MIME
- *void sendRedirect (String)* : redirige le navigateur vers l'URL
... (voir l'API Servlets pour le reste)

Types MIME : text/plain, text/html, text/xml, text/css, image/gif, audio/x-wav, application/pdf, application/javascript...

Réponse : envoi de texte

```
public class HelloWorldPrintWriter extends HttpServlet {  
    public void doGet(HttpServletRequest req,  
                      HttpServletResponse res)  
        throws ServletException, IOException {  
        res.setContentType("text/plain");  
        PrintWriter out = res.getWriter();  
        out.println("Premier Message");  
        out.println("Coucou voilà comment écrire un  
message");  
        out.println("Second Message");  
    }  
}  
Fichier HelloWorldPrintWriter.java du projet HelloWorld  
Appel par : http://localhost:8080/javaee.servlet.HelloWorld/  
printwriter
```

Réponse : redirection vers un site Web

```
public class SendRedirect extends HttpServlet {  
    public void doGet(HttpServletRequest req,  
                      HttpServletResponse res)  
        throws ServletException, IOException {  
        res.sendRedirect("http://www.google.fr");  
    }  
}
```

Fichier SendRedirect.java du projet HelloWorld

Appel par :

`http://localhost:8080/javaee.servlet.HelloWorld/
sendredirect`

```
public class DownloadFileServlet extends HttpServlet {  
    protected void doGet(HttpServletRequest arg0,  
                         HttpServletResponse arg1)  
        throws ServletException, IOException {  
        try {  
            InputStream is = new FileInputStream("c:/fich.txt");  
            OutputStream os = arg1.getOutputStream();  
            arg1.setContentType("text/plain");  
            arg1.setHeader("Content Disposition",  
                "attachment;filename=fich.txt");  
            int count;  
            byte buf[] = new byte[4096];  
            while ((count = is.read(buf)) > -1)  
                os.write(buf, 0, count);  
            is.close();  
            os.close();  
        } catch (Exception e) {}  
    }  
}
```

Fichier DownloadFileServlet.java du projet HelloWorld
Appel par : http://localhost:8080/javaee.servlet.HelloWorld/
down

Servlets et formulaires : côté HTML

- Utilisation de la balise <FORM> </FORM>
 - Option METHOD : type de requête (GET ou POST)
 - Option ACTION : URL où envoyer les données
- Utilisation de composants IHM pour saisir des données
 - Contenu à l'intérieur de la balise FORM
 - Chaque composant est défini au moyen d'une balise particulière INPUT, SELECT, TEXTAREA ...
 - A l'intérieur de chaque balise du composant (SELECT par exemple) plusieurs options et notamment une (NAME) qui permet d'identifier le composant : NAME="mon_composant"
 - Les données sont envoyées quand l'utilisateur clique sur un bouton de type SUBMIT

Le paramètre :

Servlets et formulaires : côté Servlet

```
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet MyForm</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("Hello " +request.getParameter("nom"));
        out.println("</body>");
        out.println("</html>");
    } finally {
        out.close();
    }
}

<servlet>
    <servlet-name>MyForm</servlet-name>
    <servlet-class>exemple.MyForm</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>MyForm</servlet-name>
    <url-pattern>/myForm</url-pattern>
</servlet-mapping>
```

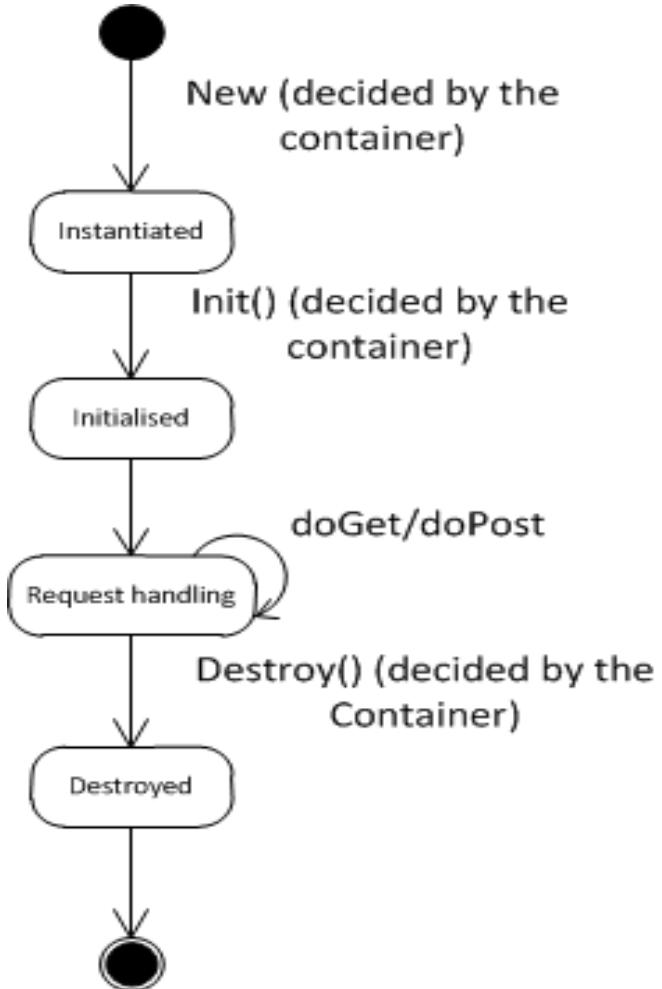


<http://localhost:8084/CoursWeb/formulaire.html>

Hello rototo

Le cycle de vie d'une servlet

1. Le conteneur appelle la méthode *init()* (initialisation de variables, connexion à des bases de données...).
2. La servlet répond aux requêtes des clients.
3. Le conteneur détruit la servlet. La méthode *destroy()* est appelée (fermeture des connexions...).



Le cycle de vie d'une servlet

Caractéristiques :

- Au moment où le code d'une Servlet est chargé le conteneur ne crée qu'**une seule instance** de classe.
- L'instance (unique) traite les requêtes.
- Entre les requêtes les Servlets **persistent** sous forme d'**instances d'objet**.

Intérêts :

- Le surcoût en temps lié à la création d'un nouvel objet à chaque requête est éliminé.
- L'empreinte mémoire reste petite.
- La persistance, c'est-à-dire la possibilité de conserver l'état de la servlet, est facilitée.

Le cycle de vie d'une servlet

Exemple : Servlet qui incrémente un compteur à chaque requête du client

```
public class SimpleCounterServlet extends  
HttpServlet {  
    private int count = 0;  
    protected void doGet(HttpServletRequest req,  
    HttpServletResponse res)  
    throws ServletException, IOException {  
        res.setContentType("text/plain");  
        PrintWriter out = res.getWriter();  
        count++;  
        out.println("Depuis son chargement, on a  
accédé à cette Servlet " + count + " fois.");  
    }  
}
```

La méthode init()

- Il n'y a **pas de constructeur** dans une Servlet.
- L'initialisation des attributs se fait par la méthode init().
- init () ne possède pas de paramètre.
- init() est définie et implémentée dans la classe abstraite GenericServlet.
- init() peut être appelée à différents moments :
 - Lorsque le conteneur de Servlets démarre,
 - Lors de la première requête à la Servlet,
 - Sur demande explicite de l'administrateur du serveur Web.

La méthode init()

Exemple : initialisation du compteur à 6.

```
public class InitCounterServlet extends HttpServlet {  
    private int count;  
    public void init() throws ServletException {  
        count = 6;  
    }  
    protected void doGet(HttpServletRequest req,  
                         HttpServletResponse res)  
        throws ServletException, IOException {  
        res.setContentType("text/plain");  
        PrintWriter out = res.getWriter();  
        count++;  
        out.println("Depuis son chargement, on a accédé à  
        cette Servlet " + count + " fois.");  
    }  
}
```

La méthode init()

Possibilité d'utiliser des paramètres d'initialisation définis dans le fichier **web.xml** de l'application Web.

Exemple :

```
<web-app ...>
  <display-name>Servlet compteur</display-name>
  <servlet>
    <servlet-name>ConfigFileCounterServlet</servlet-name>
    <servlet-class>ConfigFileCounterServlet</servlet-
class>
    <init-param>
      <param-name>initial_counter_value</param-name>
      <param-value>50</param-value>
      <description>Valeur initiale</description>
    </init-param>
  </servlet>
...
</web-app>
```

La méthode init()

La servlet correspondante :

```
public class InitConfigFileCounterServlet extends  
HttpServlet {  
private int count;  
public void init() throws ServletException {  
    String initial =  
        this.getInitParameter("initial_counter_value");  
    try {  
        count = Integer.parseInt(initial);  
    } catch(NumberFormatException e) {  
        count = 0;  
    }  
}  
protected void doGet(HttpServletRequest req,  
HttpServletResponse res)  
throws ServletException, IOException {  
    ...  
    count++;  
    out.println("Depuis son chargement, on a accédé à  
    cette Servlet " + count " fois.");  
}
```

La méthode *destroy()*

- La méthode *destroy()* permet de libérer toutes les ressources acquises comme des connexions à une base de données ou à des fichiers.
- La méthode *destroy()* est également utilisée pour sauvegarder des informations qui seront lues lors du prochain appel à *init()*.

Les sessions

- Comment garder des informations d'état au cours d'une série de requêtes d'un même utilisateur pendant un temps donné ?
- HTTP est un protocole sans état qui n'offre donc pas de solution.
- On peut utiliser les solutions classiques comme la réécriture d'URL, les champs cachés dans les formulaires ou les cookies.
- L'API *HttpSession* fournit des fonctions pour gérer les sessions.

Les cookies

- Un cookie est une information envoyée au navigateur (client) par un serveur Web qui peut ensuite être relue par le client.
- Lorsqu'un client reçoit un cookie, il le sauve et le renvoie ensuite au serveur chaque fois qu'il accède à une page sur ce serveur.
- La valeur d'un cookie pouvant identifier de façon unique un client, ils sont souvent utilisés pour le suivi de session.
- L'API Servlet fournit la classe `javax.servlet.http.Cookie` pour travailler avec les Cookies :
 - `Cookie(String name, String value)` : construit un cookie
 - `String getName()` : retourne le nom du cookie
 - `String getValue()` : retourne la valeur du cookie
 - `setValue(String new_value)` : donne une nouvelle valeur au cookie
 - `setMaxAge(int expiry)` : spécifie l'âge maximum du cookie

Les cookies

- Pour créer un nouveau cookie il faut l'ajouter à la réponse (*HttpServletResponse*) :
addCookie(Cookie mon_cook) : ajoute à la réponse un cookie.
Exemple :

```
Cookie cookie = new Cookie("Id", "123");
res.addCookie(cookie);
```

- La Servlet récupère les cookies du client en exploitant la requête (*HttpServletRequest*) :

Cookie[] getCookies() : récupère l'ensemble des cookies du site.

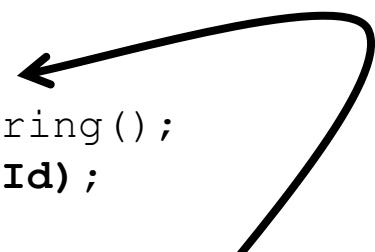
Exemple :

```
Cookie[] cookies = req.getCookies();
if (cookies != null) {
    for (int i = 0; i < cookies.length; i++) {
        String name = cookies[i].getName();
        String value = cookies[i].getValue();
    }
}
```

Les cookies

Exemple : gestion de session (identifier un client d'un autre par les cookies)

```
public class CookiesServlet extends HttpServlet {  
protected void doGet(HttpServletRequest req,  
HttpServletResponse res)  
throws ServletException, IOException {  
    ...  
    String sessionId = null;  
    Cookie[] cookies = req.getCookies();  
    if (cookies != null) {  
        for (int i = 0; i < cookies.length; i++) {  
            if (cookies[i].getName().equals("sessionid")) {  
                sessionId = cookies[i].getValue();  
            } } }  
    if (sessionId == null) {  
        sessionId = new java.rmi.server.UID().toString();  
        Cookie c = new Cookie("sessionid", sessionId);  
        res.addCookie(c);  
        out.println("Bonjour le nouveau");  
    } else {  
        out.println("Encore vous"); ... }  
    }  
}
```



Génère un identifiant unique pour chaque client

Les cookies

Inconvénients des cookies :

- Les navigateurs ne les acceptent pas toujours.
- L'utilisateur peut configurer son navigateur pour qu'il refuse ou pas les cookies.
- Le nombre et la taille des cookies peuvent être limités par le navigateur.

Suivi de session : *HttpSession*

- Mécanisme très puissant permettant de stocker des objets et non de simples chaînes de caractères comme les cookies.
- Méthode de création (de *HttpServletRequest*) :
 - *HttpSession getSession()* : retourne la session associée à l'utilisateur.
- Gestion d'association (de *HttpSession*) :
 - Enumeration *getAttributNames()* : retourne les noms de tous les attributs.
 - Object *getAttribut(String name)* : retourne l'objet associé au nom.
 - *setAttribut(String na, Object va)* : donne la valeur va à l'attribut na.
 - *removeAttribut(String na)* : supprime l'attribut de nom na.
- Destruction (de *HttpSession*) :
 - *logout()* : termine la session.

Suivi de session : *HttpSession*

Exemple : suivi de session pour le compteur

```
public class HttpSessionServlet extends  
    HttpServlet {  
protected void doGet(HttpServletRequest req,  
HttpServletResponse res)  
throws ServletException, IOException {  
    res.setContentType("text/plain");  
    PrintWriter out = res.getWriter();  
HttpSession session = req.getSession();  
    Integer count =  
        (Integer)session.getAttribute("count");  
    if (count == null)  
        count = new Integer(1);  
    else  
        count = new Integer(count.intValue() + 1);  
session.setAttribute("count", count);  
    out.println("Vous avez visité cette page " +  
        count + " fois.");  
}  
}
```

Partage d'information dans une application

- Le *ServletContext* est un conteneur d'informations unique **de l'application web.**
- Une Servlet retrouve le ServletContext de son application web par appel à *getServletContext()*.
- Exemples de méthodes de *ServletContext* :
 - *void setAttribute(String name, Object o)* : stocke un objet sous le nom indiqué
 - *Object getAttribute(String name)* : retrouve l'objet sous le nom indiqué
 - *Enumeration getAttributeNames()* : retourne l'ensemble des noms de tous les attributs stockés
 - *void removeAttribute(String name)* : supprime l'objet stocké sous le nom indiqué

...

```
public class PizzasAdmin extends HttpServlet {  
    protected void doGet(HttpServletRequest req,  
                         HttpServletResponse res)  
        throws ServletException, IOException {  
        res.setContentType("text/plain");  
        PrintWriter out = res.getWriter();  
        ServletContext context = this.getServletContext();  
        context.setAttribute("Specialite", "saumon");  
        context.setAttribute("Date", new Date());  
        out.println("La pizza du jour a été définie.");  
    } }
```

PizzasAdmin.java du projet **ServletContext**

```
public class PizzasClient extends HttpServlet {  
    protected void doGet(HttpServletRequest req,  
                         HttpServletResponse res)  
        throws ServletException, IOException {  
        ServletContext context = this.getServletContext();  
        String pizza_spec =  
            (String)context.getAttribute("Specialite");  
        Date day = (Date)context.getAttribute("Date");  
        DateFormat df =  
            DateFormat.getDateInstance(DateFormat.MEDIUM);  
        String today = df.format(day);  
        out.println("Aujourd'hui (" + today +  
                  "), notre spécialité est : " + pizza_spec);  
    } }
```

PizzasClient.java du projet **ServletContext**

Partage d'information entre applications

- Première solution : partage d'un conteneur d'informations externe (une base de données par exemple).
- Deuxième solution : la Servlet recherche un autre contexte du même serveur à partir de son propre contexte.
ServletContext getContext(String uripath) : obtient le contexte à partir d'un chemin absolu.
- Cette communication inter contextes doit être autorisée.

Partage d'information entre applications

Exemple : affichage de la spécialité du jour de l'application précédente.

```
public class ReadSharePizzas extends HttpServlet {  
    protected void doGet(HttpServletRequest req,  
    HttpServletResponse res)  
    throws ServletException, IOException {  
        res.setContentType("text/html");  
        PrintWriter out = res.getWriter();  
        ServletContext my_context = this.getServletContext();  
        ServletContext pizzas_context =  
            my_context.getContext("/ServletContext");  
        String pizza_spec =  
            (String)pizzas_context.getAttribute("Specialite");  
        Date day = (Date)pizzas_context.getAttribute("Date");  
        DateFormat df =  
            DateFormat.getDateInstance(DateFormat.MEDIUM);  
        String today = df.format(day);  
        out.println("Aujourd'hui (" + today +  
            "), notre specialite est : " + pizza_spec);  
    }  
}
```

ReadSharePizzas.java du projet **CrossServletContext**

En résumé



Niveau de la requête client



Niveau de la session client



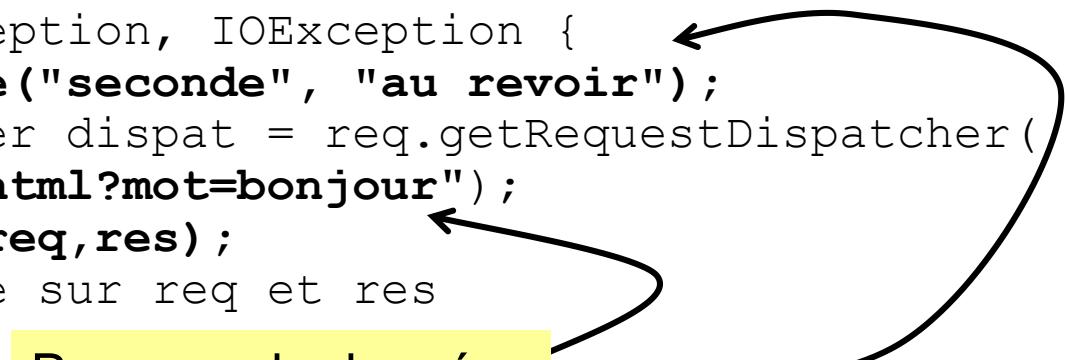
Niveau de l'application web

Partage du contrôle dans une application

- Une Servlet peut laisser à une autre Servlet de la même application tout ou partie du traitement. Utilisé souvent pour structurer une application avec une servlet contrôleur et des servlets spécialisées.
- Plus précisément une Servlet peut déléguer une requête entière ou inclure la réponse d'un autre programme.
- Pour déléguer une requête, il faut obtenir un objet *RequestDispatcher*. On peut ensuite 'forwarder' la requête vers une autre servlet (ou le plus souvent vers une page JSP). Le contrôle ne revient plus à la servlet d'origine.
Des données peuvent être passées par l'URL ou par le ServletContext.

Partage du contrôle dans une application

```
public class SenderServlet extends HttpServlet {  
    protected void doGet(HttpServletRequest req,  
                         HttpServletResponse res)  
        throws ServletException, IOException {  
        req.setAttribute("seconde", "au revoir");  
        RequestDispatcher dispat = req.getRequestDispatcher(  
            "/recepteur.html?mot=bonjour");  
        dispat.forward(req, res);  
        // Ne rien faire sur req et res  
    } }
```



SenderServlet.java du projet **ForwardInclude**

```
public class ReceiverServlet extends HttpServlet {  
    protected void doGet(HttpServletRequest req,  
                         HttpServletResponse res)  
        throws ServletException, IOException {  
        res.setContentType("text/plain");  
        PrintWriter out = res.getWriter();  
        out.println(req.getParameter("mot"));  
        out.println(req.getAttribute("seconde"));  
    } }
```



ReceiverServlet.java du projet **ForwardInclude**

Partage du contrôle dans une application

- La méthode *include()* de *RequestDispatcher* inclut le contenu d'une ressource dans la réponse courante.

```
RequestDispatcher dispat =  
    req.getRequestDispatcher("/index.html");  
dispat.include(req,res);
```

- La différence avec un *forward()* est :
 - la Servlet appelante garde le contrôle de la réponse,
 - elle peut inclure du contenu avant et après le contenu inclus.
- Il y a également possibilité de transmettre des informations lors de l'inclusion
 - dans l'URL,
 - par *setAttribute()*.

Autres fonctionnalités

- Les listeners invoqués en fonction du cycle de vie de la servlet.
- L'accès aux bases de données par SQL via l'API JDBC ('Java DataBase Connectivity').
- La gestion de la sécurité (authentification liée à HTTP ou externe, gestion de rôles).
- La gestion des pages d'erreurs (spécifiée dans web.xml).
- Les filtres pour le prétraitement (contrôle des paramètres, log des appels...) ou le post-traitement des requêtes (compression gzip, transformation par XSLT...).

Services Web

5. Web-services

■ Definition:

- "A *Web service* is a **method of communication** between two electronic devices **over the World Wide Web.**"

Wikipedia

- *Web service* is "a software system designed to support **interoperable machine-to-machine interaction over a network.**"

W3C

5. Web-services

- Definition:
 - A Web service has an interface described in a machine-processable format (specifically Web Services Description Language, aka **WSDL**).
 - Other systems interact with the Web service in a manner prescribed by its description using **SOAP** messages, typically conveyed using **HTTP** with an **XML serialization** in conjunction with other Web-related standards.

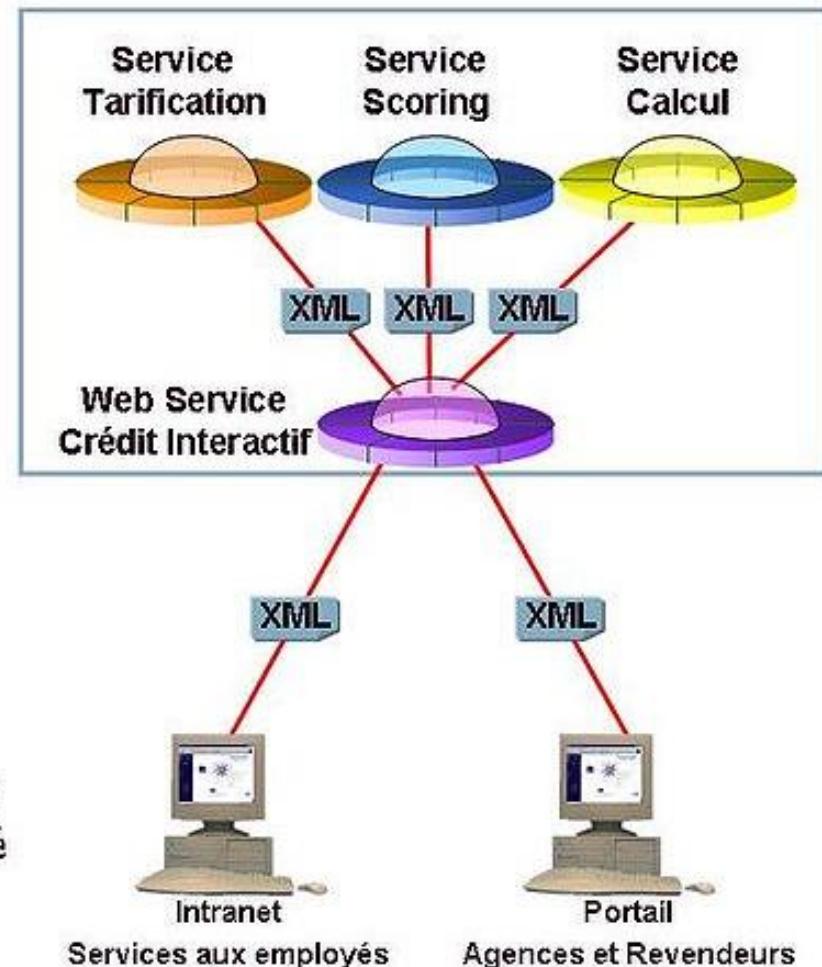
5. Service-Oriented Architecture

"A Web service is a software application or component that can be accessed over the Internet using a platform/language-neutral data interchange format to invoke the service and supply the response, using a rigorously defined message exchange pattern, and producing a result that is sufficiently well-defined to be processed by a software application."



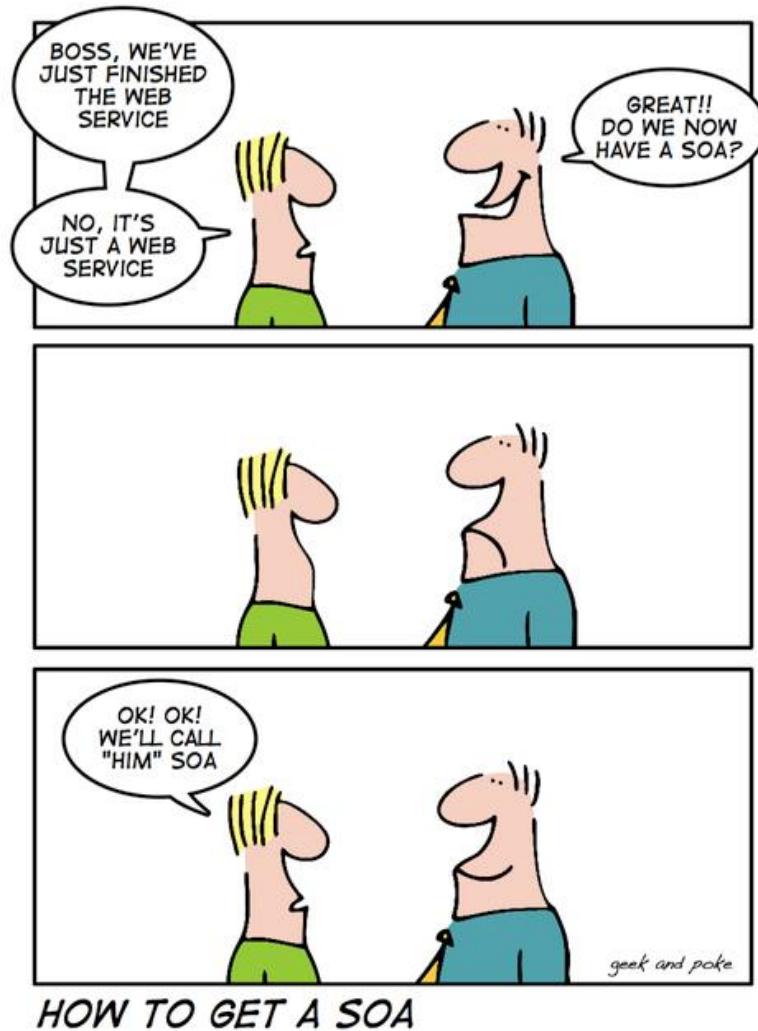
Architecture
domain

- Les Web Services sont:
 - des composants métiers
 - auto suffisants
 - auto descriptifs
 - exécutés sur Internet ou Intranet
 - respectant des contrats de qualité

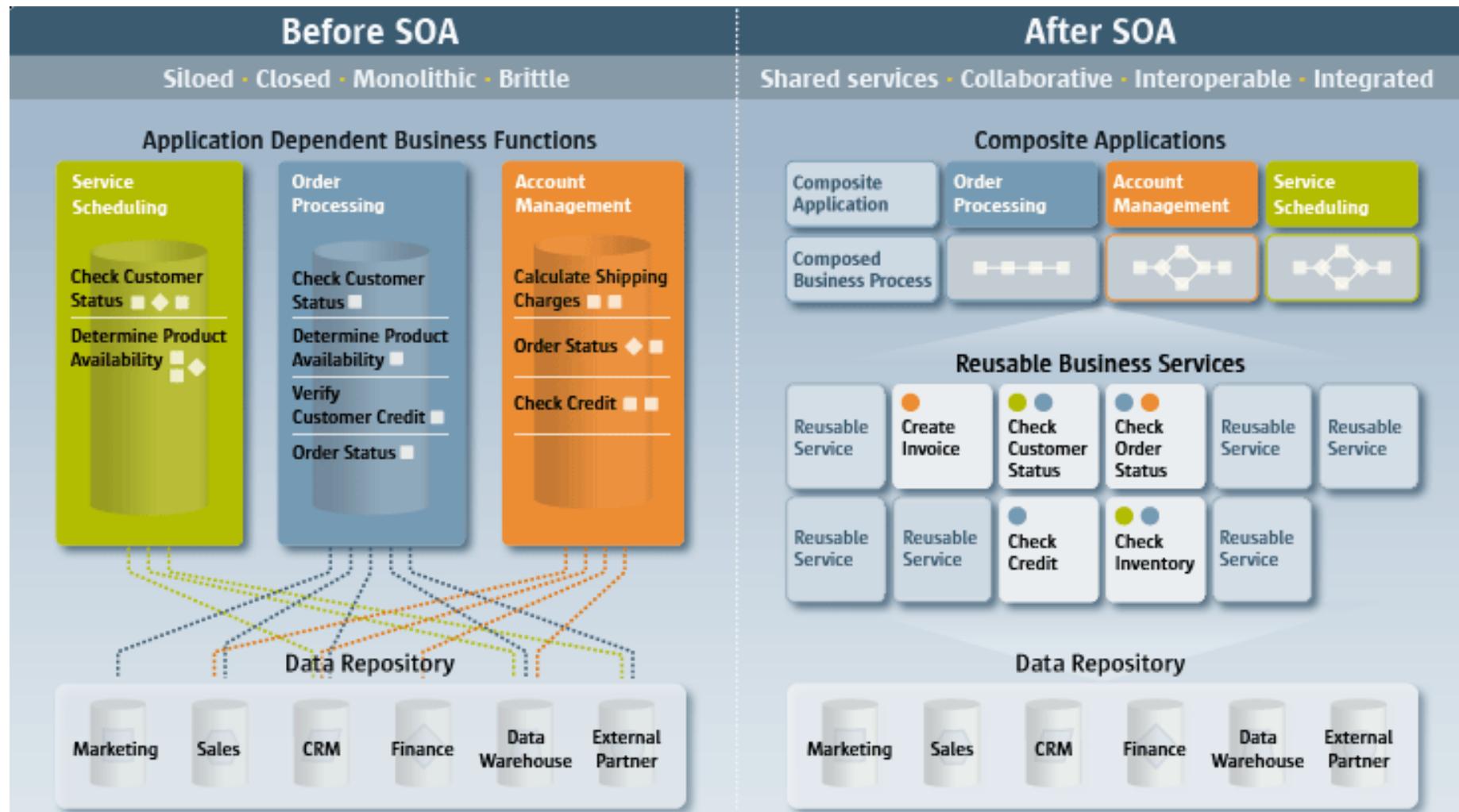


5. Service-Oriented Architecture

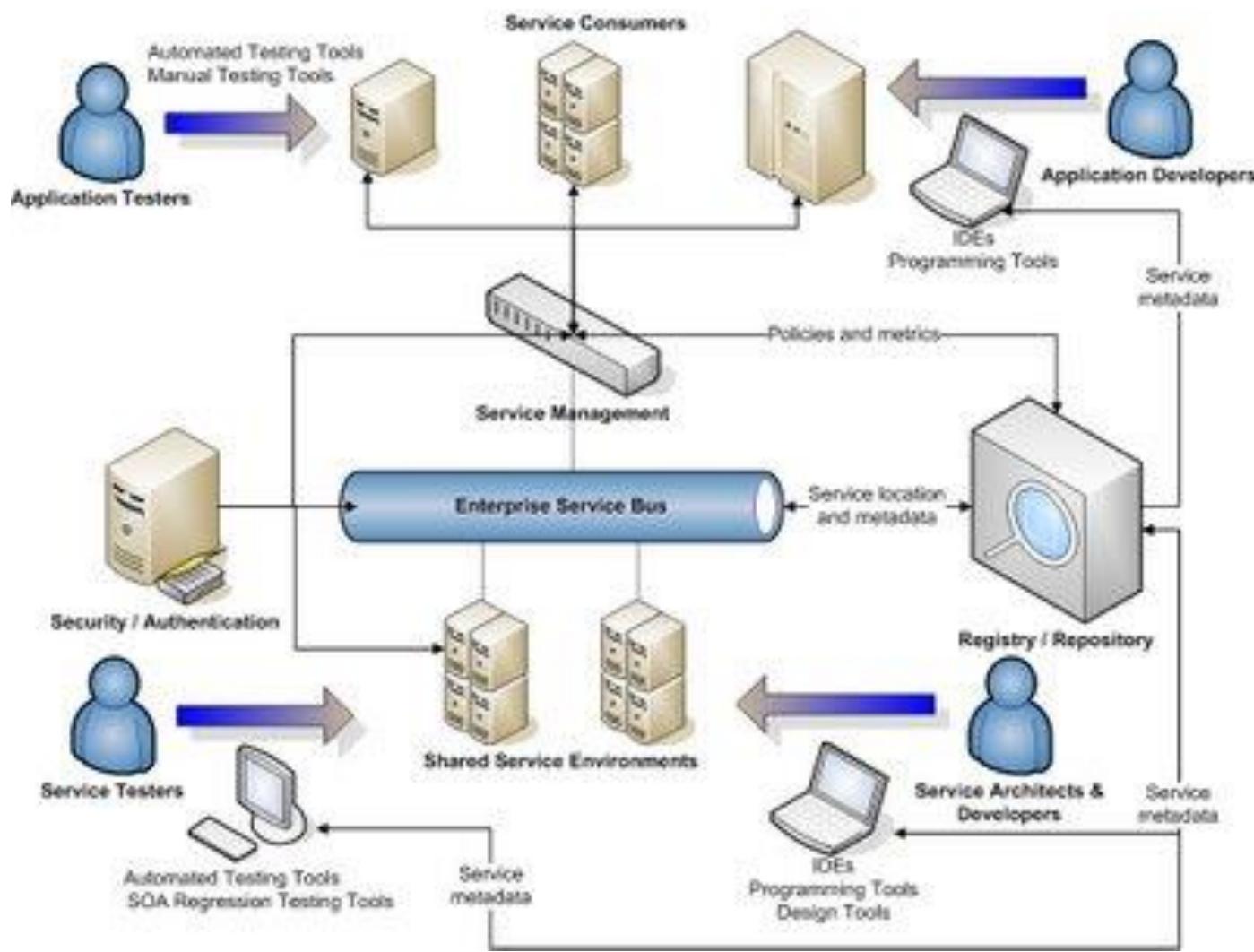
- SOA is a set of principles and **methodologies** for designing and developing software in the form of **interoperable** services.
- These services are well-defined **business functionalities that are built as software components** that can be **reused** for different purposes.



5. SOA example



5. SOA example



5. Web-services

- Definition:

- W3C has identified 2 major classes of Web services
 - REST-compliant Web services
 - the primary purpose of the service is to **manipulate XML representations of Web resources** using a uniform set of **stateless** operations;
 - It's the **Web (2.0) API**
 - a defined set of HTTP request messages along with a definition of the structure of response messages, typically expressed in JSON or XML.
 - Arbitrary Web services
 - the service may expose an arbitrary set of operations

5. HTTP

- HyperText Transfer Protocol
 - Stateless request/response client-server protocol
- Requests:
 - Method: GET, POST, HEAD, TRACE, OPTIONS, PUT, DELETE

5. HTTP

- Requests, continued
 - URI (required in HTTP/1.1)
 - Header Fields
 - E.g. how the response should be returned, under what conditions, identification and characterization of client, accounting data
 - Body
 - POST data
 - Empty for GET

5. HTTP

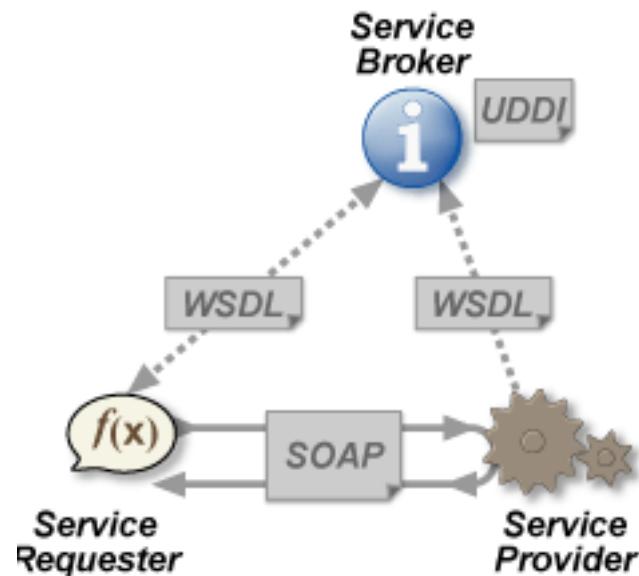
- Response:
 - Status code (machine), reason (human)
 - Header
 - Metadata, e.g. Content-Type (Media type), Content-Length, Last-Modified, Etag
 - Body
 - (X)HTML, other XML, text, binary data ...

5. URL Connections

- `java.net` also -- connections extend `Socket`
- Encapsulates HTTP and FTP connections
 - `URI`, `URL`, `URLConnection`, `HttpURLConnection`

5. What is SOAP?

- The *de facto* standard for Web Service communication that provides support for:
 - *Remote procedure call* (RPC) to invoke methods on servers
 - *Messaging* to exchange documents
 - Extensibility
 - Error handling
 - Flexible data encoding
 - Binding to a variety of transports (e.g., SOAP, SMTP)



5. HTTP Binding

- Both POST and GET are used to transport SOAP messages

POST /fareService/getFareOp HTTP/1.1

Host: www.SlowHawk.com

Content-Type: application/soap+xml

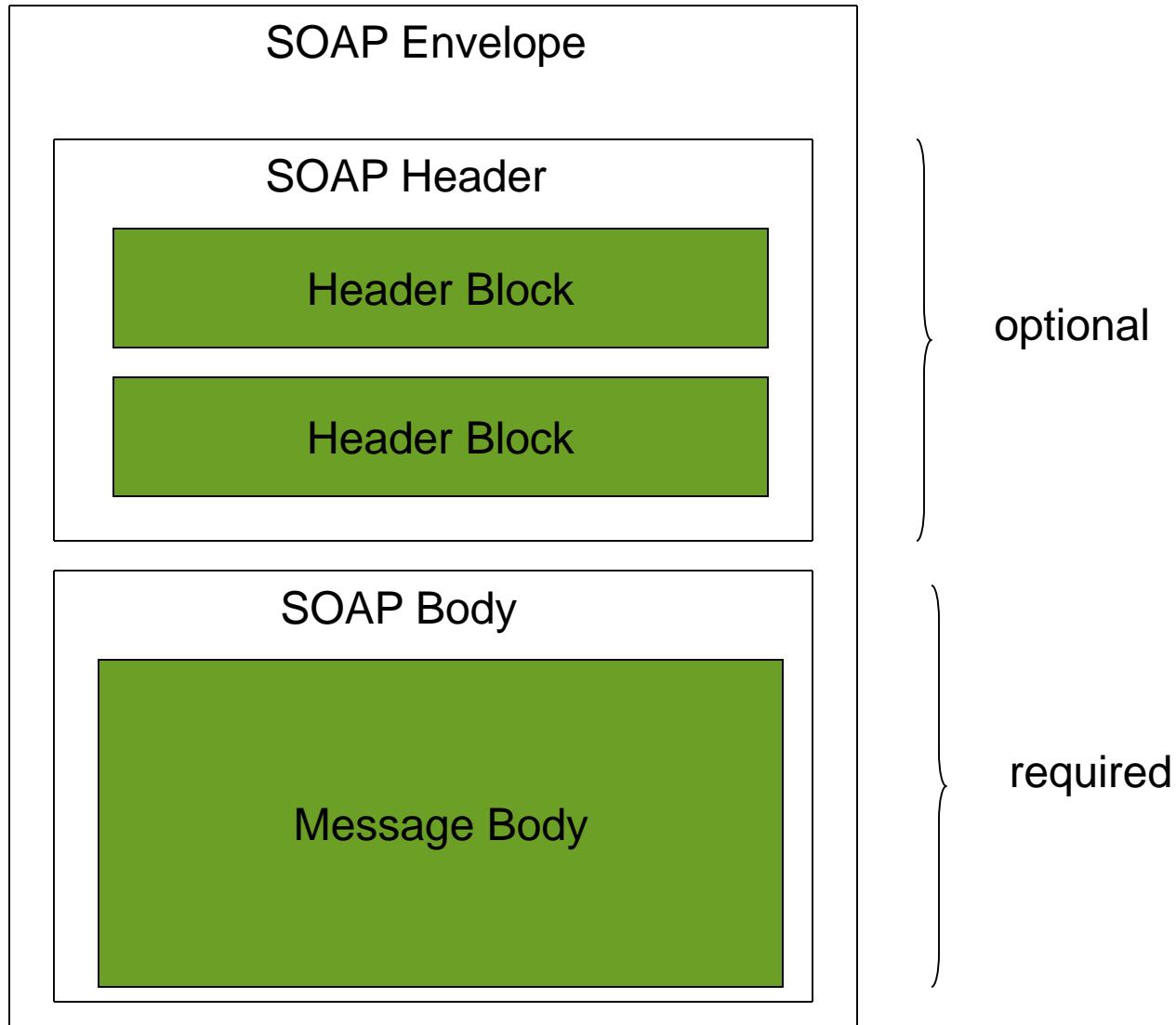
Content-Length: xxx

<!-- the SOAP message goes here -->

5. SOAP and XML

- Since XML is language and platform independent, it is the *lingua franca* for the exchange of information in a heterogeneous distributed system.
 - SOAP supports the transmission of arbitrary XML documents
 - For RPC, SOAP provides a message format for invoking a procedure and returning results in XML

5. SOAP Message



5. SOAP Envelope

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope">
  <s:Header>
    <!-- header blocks go here -->
  </s:Header>

  <s:Body>
    <!-- an XML document goes here -->
  </s:Body>
</s:Envelope>
```

`http://www.w3.org/2003/05/soap-envelope` identifies a name space that defines the structure of a SOAP message

5. Using SOAP

- For document exchange, the XML document being exchanged is nested directly in SOAP envelope.
 - Referred to as **document-style** SOAP
 - Conversational mode of message exchange
- For RPC, SOAP defines the format of the body of messages to be used for invocation and response.
 - Referred to as **RPC-style** SOAP
 - Uses a request-response pattern
 - Parameters are passed by value/result

5. RPC Request Message

Client invocation of procedure:

```
public Float getQuoteOp(String symbol);
```

generates SOAP request message:

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <s:Body>
    <n:getQuoteOp xmlns:n="http://www.shearson.com/quoteService">
      <n:stockSymbol xsi:type="xsd:string">
        IBM
      </n:stockSymbol>
    </n:getQuoteOp>
  </s:Body>
</s:Envelope>
```

5. RPC Response Message

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <s:Body>
        <n:getQuoteOpResponse
            xmlns:n="http://www.shearson.com/quoteService">
            <n:value xsi:type="xsd:float">
                30.45
            </n:value>
        </n:getQuoteOpResponse>
    </s:Body>
</s:Envelope>
```

5. RPC Request/Response Messages

- Conventions:
 - Name of the request structure is same as method name.
 - Name of response structure is same as method name concatenated with “Response”
 - Name and order of in and in/out parameters in request structure same as name and order in signature
 - Value of method (if returned) is first child element of response structure; out and in/out parameters follow, their name and order same as name and order in signature

5. Data Encoding

- **Problem:** SOAP provides a language/platform independent mechanism for invoking remote procedures
 - Arguments are carried in an XML document
 - Caller and callee may use different representations of the same types (e.g., Java, C)
 - A mechanism is needed for mapping from caller's format to XML syntax to callee's format (referred to as **serialization/deserialization**)
 - Example: mapping a Java array to XML syntax

5. Serialization

- Serialization is simple for simple types (integer, string, float,...) since they correspond to XML Schema types.
 - Translate from binary to ASCII using XML schema specified format
- Serialization not so simple for complex types
 - Ex: What tags will be used for an array? Will it be stored by rows or columns?

5. Encoding Style

- **encodingStyle** attribute used to identify the serialization rules to encode the data contents of an element
 - An arbitrary set of rules can be used
 - SOAP defines its own set of rules
 - Message is referred to as **RPC/encoded**
- SOAP defines its own graphical data model for describing complex types and rules for transforming instances of the model into serialized ASCII strings
 - Vendors provide serializers (and deserializers) which map local types to instances of the model and then transform the local representation to the encoded data using the SOAP rules

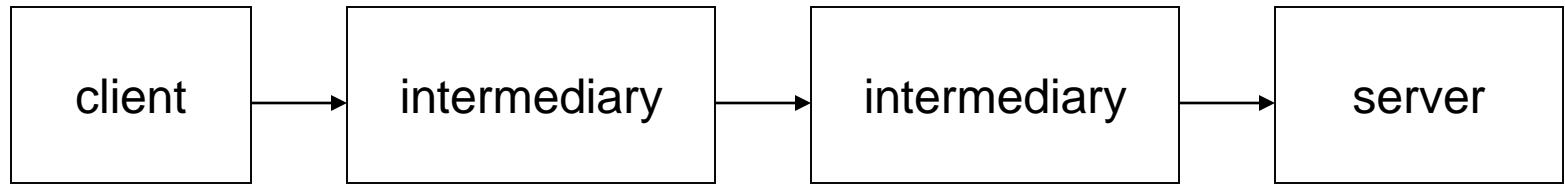
5. Data Encoding

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"  
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">  
  
    <s:Body>  
        <n:getQuoteOp  
            xmlns:n="http://www.shearson.com/quoteService"  
            s:encodingStyle="http://www.w3.org/2003/05/soap-  
            encoding">  
            <n:symbol xsi:type="xsd:string">  
                IBM  
            </n:symbol>  
        </n:getQuoteOp>  
    </s:Body>  
</s:Envelope>
```

5. SOAP Extensibility

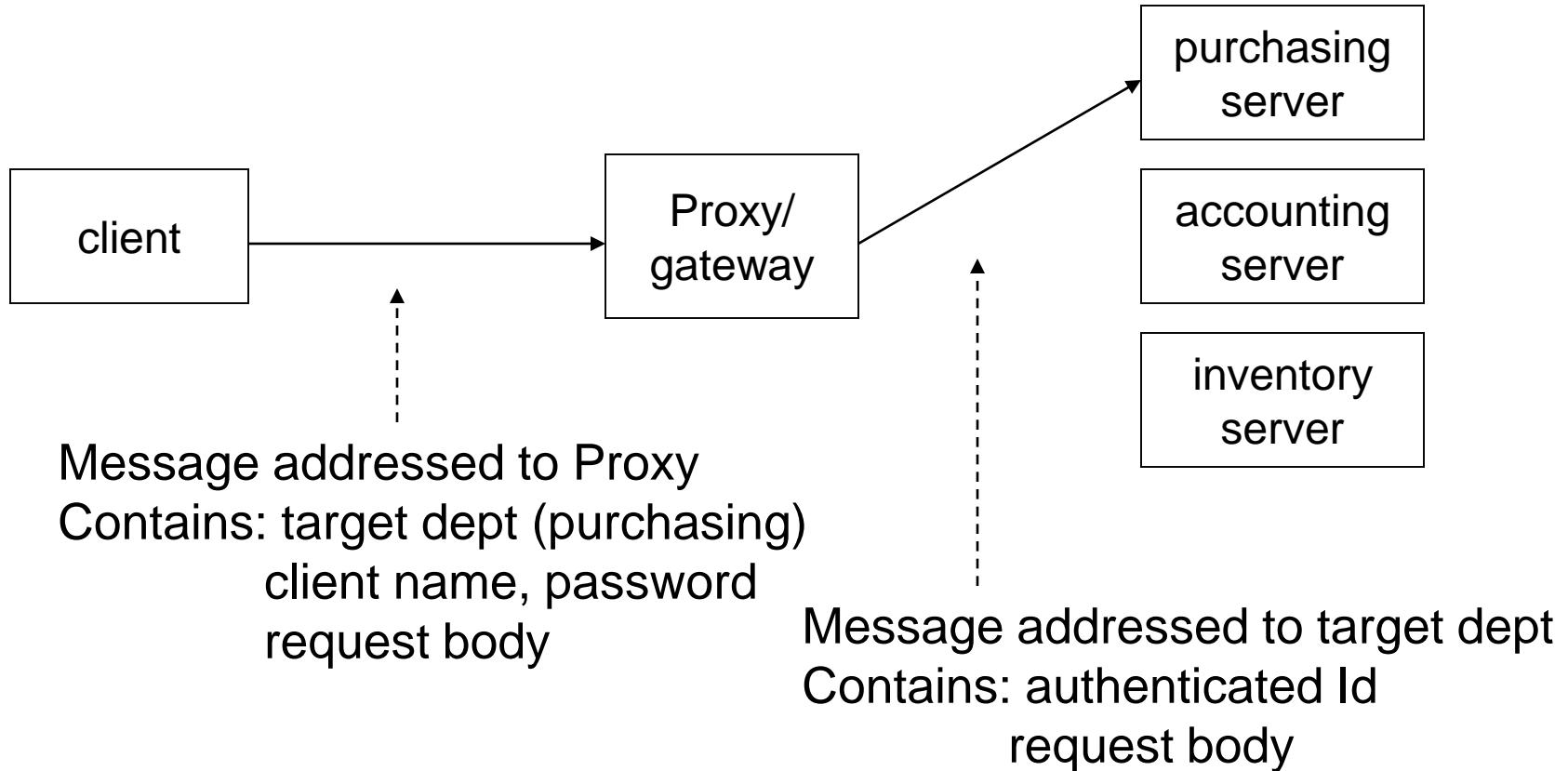
- A SOAP message goes from client to server to advance some application related cause.
- It is often the case that some orthogonal issues related to the message must be handled:
 - Security: encryption, authentication, authorization
 - Transaction management
 - Tracing
 - Logging

5. Intermediaries



- To support scalability and decentralization, these issues need not be handled by the server.
 - **Intermediaries** between client and server are used
- Intermediaries perform orthogonal services as the message passes along a route

5. Example



5. Requirements

- Information directed to each intermediary and to final destination kept separate
 - Intermediaries can be easily added/deleted, route changed
- SOAP does not specify how routing is to be done (although protocols are being developed for this purpose)
 - It is up to each node along the chain to know where to send the message next

5. Header

- SOAP envelope defines an optional **header** containing an arbitrary number of **header blocks**. Each block:
 - Has an optional **role** and should be processed by an intermediary that can perform that role
 - Can have its own namespace declaration
 - Eliminates the possibility of interference between groups that independently design headers.

5. Example – Client Message

POST /purchasing/retailSale HTTP/1.1

Host: proxy.yourcompany.com

.....

```
<s:Envelope xmlns:s=....>
  <s:Header>
    <td:targetdept xmlns:td=“....”
      s:role=“company-proxy.com”
      s:mustUnderstand=“true”
      purchasing
    </td:targetdept>
    <auth:authinfo=“....”
      s:role=“company-proxy.com”
      s:mustUnderstand=“true” >
      <auth:name> madonna </auth:name>
      <auth:passwd> xxxxxx </auth:passwd>
    </auth:authinfo>
  </s:Header>
  <s:Body> ..... </s:Body>
</s:Envelope>
```

-- method invoked at final destination

-- initial destination intermediary

-- identifies intermediary

-- this header better be processed

-- identifies next node

-- identifies intermediary

-- this header better be processed

5. Processing Model

- An intermediary has an assigned set of roles
- On receiving a message, it identifies the blocks whose role attribute matches an element of its set (or has value *next*)
 - Block without a role attribute targeted for final destination
- The intermediary
 - can modify/delete its block
 - can insert new blocks
 - should retarget the message to the next destination
 - can do anything (frowned upon)

5. Must Understand

- An intermediary can choose to ignore a block directed to it
- If **mustUnderstand** attribute has value “*true*” intermediary *must* process the block or else abort the message and return a fault message

5. Example – Proxy Message

POST /purchasing/retailSale HTTP/1.1
Host: purchasing.yourcompany.com

-- method invoked at destination
-- initial intermediary

```
.....  
<s:Envelope xmlns:s=....>  
  <s:Header>  
    <cc:ClientCredentials xmlns:cc=“....”  
      s:mustUnderstand=“true” >  
        .....  
        <cc:clientId> 122334 </cc:clientId>  
    </ cc:ClientCredentials >  
  </Header>  
  <s:Body> ..... </s:Body>  
</s:Envelope>
```

-- this block better be processed
-- destination (no role specified)

-- same body

5. Example – Proxy Message

- Proxy has deleted the two headers
 - Verified that user is valid using *<name>* and *<passwd>* and determined Id
 - Retargeted message to final destination using *<targetdept>*
- Proxy has inserted a new header containing Id
 - Final destination uses Id to determine authorization

5. SOAP Faults

- SOAP provides a message format for communicating information about errors containing the following information:
 - Fault category identifies error (not meant for human consumption) –
 - VersionMismatch
 - MustUnderstand – related to headers
 - Sender – problem with message content
 - Receiver – error had nothing to do with the message
 - human readable explanation
 - node at which error occurred (related to intermediaries)
 - application specific information about Client error

Embedding SOAP in HTTP: POST

- For document-style SOAP, the envelope is the body of an HTTP POST.
 - The HTTP response message simply acknowledges receipt of HTTP request message
- For RPC-style SOAP, the envelope containing the SOAP request is the body of an HTTP POST; the envelope containing the SOAP response (or fault) is the body of the corresponding HTTP response.

5. Embedding SOAP in HTTP

POST /StockQuote HTTP/1.1

Content-Type: text/xml

Content-Length:

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <s:Body>
        <n:getQuoteOp
            xmlns:n="http://www.shearson.com/quoteService"
            s:encodingStyle="http://www.w3.org/2001/06/soap-
            encoding">
            <n:stockSymbol xsi:type="xsd:string">
                IBM
            </n:stockSymbol>
        </n:getQuoteOp>
    </s:Body>
</s:Envelope>
```

Java EE : Architecture REST Introduction à Jersey



Representational State Transfer (**REST**)

PLAN

- Introduction à REST
- Les principes clefs
- Développer un service web RESTful
- REST et les Web Services WS-*
- Cas d'utilisations
- Conclusion

REST c'est quoi ?

Representational State Transfer (REST)

- Thèse de Roy Fielding (2000)
 - REST a été introduit en 2000 par Roy Fielding, président de la fondation Apache et membre actif de l'IETF et éditeur de HTTP 1.1 dans le cadre d'une thèse en... philosophie!
- Principes architecturaux pour applications Web
 - série de principes architecturaux permettant aux applications Web de respecter les principes architecturaux du Web et donc de tirer pleinement partie de son potentiel.
- Affirme le caractère sans état et sans connexion
 - promotion du caractère « sans état et sans connexion » du protocole HTTP = principe de base que les applications ne devraient pas chercher à contourner.

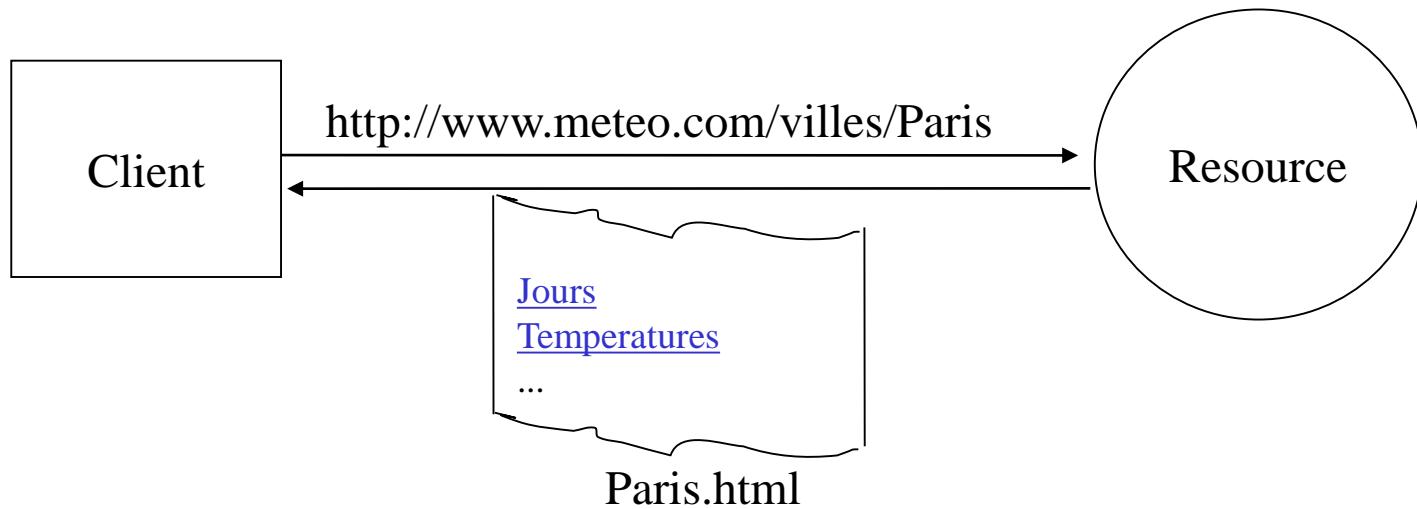
REST c'est quoi ?

- Thèse de Roy Fielding en 2000
- Un style d'architecture
- Un ensemble de contraintes
 - Client /serveur
 - Sans états (Stateless)
 - Cache
 - Interface uniforme
- La plus connue des implémentations de REST est HTTP

Les principes clefs

- Une ressource
- Un identifiant de ressource
- Une représentation
- Interagir avec les ressources
 - Exemple avec HTTP : GET, POST, PUT et DELETE

Pour résumer



Un service RESTful

- Identifier les ressources
- Définir les URLs
- Spécifier les méthodes des interfaces
- Lier les ressources

- REST utilise des standards. En particulier :
 - URI comme syntaxe universelle pour adresser les ressources,
 - HTTP un protocole sans état (stateless) avec un nombre très limité d'opérations,
 - Des liens hypertextes dans des documents (X)HTML et XML pour représenter à la fois le contenu des informations et la transition entre états de l'application,
 - Les types MIME comme text/xml, text/html, image/jpeg, application/pdf, video/mpeg pour la représentation des ressources.
- REST concerne l'architecture globale d'un système. Il ne définit pas la manière de réaliser dans les détails. En particulier, des services REST peuvent être réalisés en .NET, JAVA, ...

REST GET préféré au POST

- Deuxième principe de l'architecture REST : le HTTP GET est "sûr", c'est à dire que l'utilisateur ou son agent peut suivre des liens sans obligations.
- La conséquence de l'idempotence du GET (deux accès successifs donnent le même résultat) rend l'utilisation du Web plus fiable car un deuxième clic sur un lien ne modifie pas le résultat.
 - L'utilisation de GET est "sûre" (safe), c'est à dire que l'état de la ressource ne doit pas être modifiée par un GET. Ceci autorise les liens, la mise en cache, les favoris. Il faut donc utiliser GET pour des opérations qui ressemblent à des questions ou à des lectures de l'état de la ressource.
 - En revanche, il faut utiliser POST quand la demande ressemble à une commande, ou quand l'état de la ressource est modifié ou quand l'utilisateur est tenu pour responsable du résultat de l'interaction.

REST

Identification par URI logique

- L'identification des ressources est la pierre angulaire d'une architecture REST, elle passe par l'utilisation systématique d'URI
- Les URI doivent être spécifiées au moment de la conception.

- **préférer un nommage logique à un nommage physique** pour masquer une implémentation spécifique. Sous Apache, il existe un module mod_rewrite qui permet de manière transparente de rediriger une URL vers une autre. Sous IIS, c'est un peu plus compliqué car il n'y a rien en standard. On peut soit écrire un filtre ISAPI, soit utiliser un composant d'une tierce partie.
- **Les ressources doivent être identifiées par des noms, pas par des verbes**
- **Les URI ne doivent pas changer** car vous ne saurez jamais qui détient des vieilles références : liens dans d'autres pages, utilisateurs dans leurs favoris, notes sur un bout de papier.

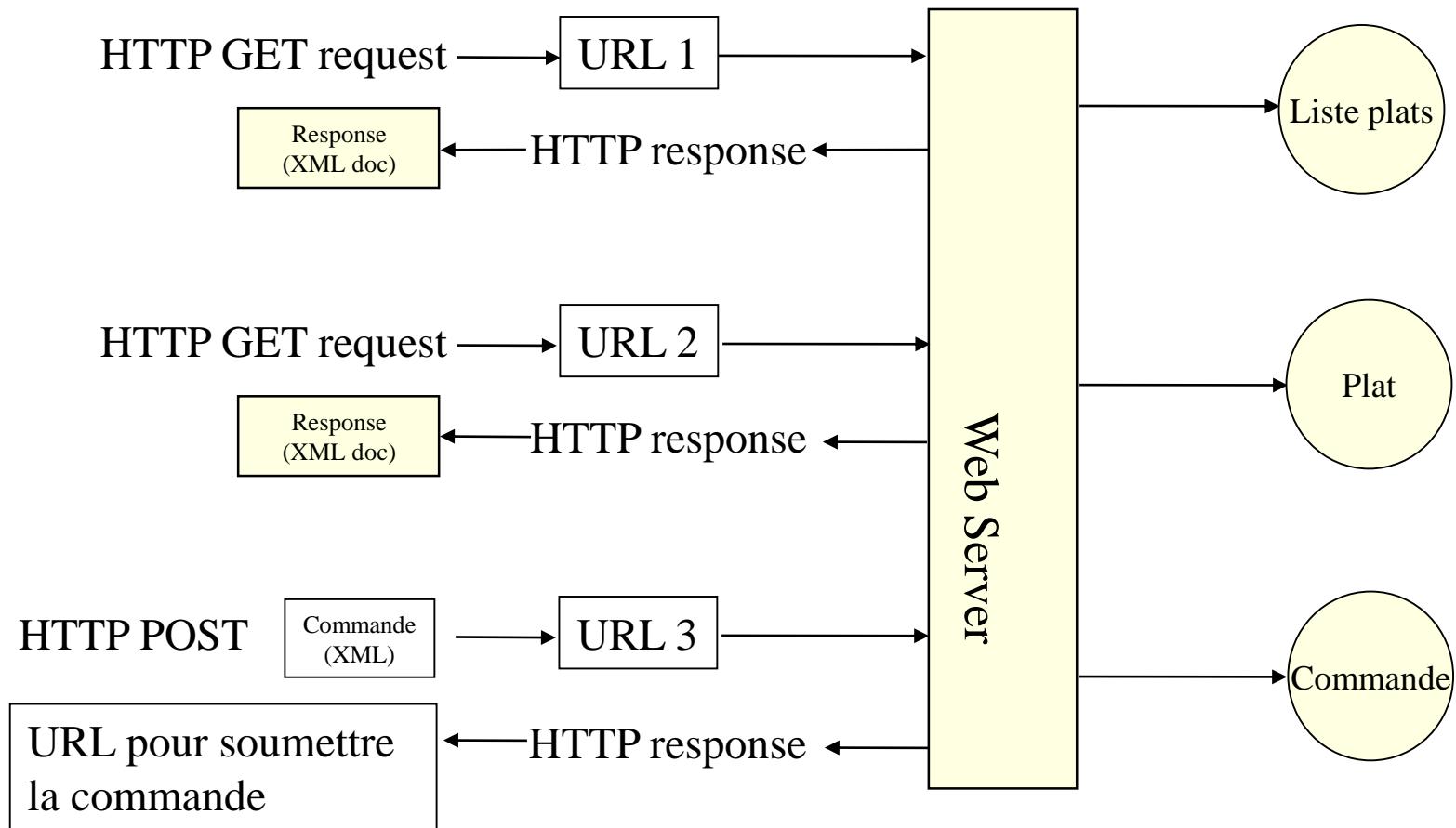
- **La représentation des données doit s'appuyer sur des standards** comme utf-8 pour le jeu de caractères, le XML pour la syntaxe des documents et des vocabulaires spécifiques (Dublin Core, RSS, Atom...) pour la sémantique des données. C'est le rôle de l'architecte de bien spécifier tous les standards de l'application.
- Remarques :
 - Une URI logique ne contient pas d'indication sur la manière dont chaque ressource élabore ses réponses. C'est ce qu'on appelle un couplage lâche (loosely coupled).
 - Quand la requête est complexe, il faut quelquefois réaliser un formulaire qui construira l'URI à partir de ses données.

- Avec REST et les URI logiques, il devient très facile de mettre en place un système de contrôle d'accès aux ressources.
 - Par exemple, imaginons un système où toutes les URI d'accès aux ressources d'un groupe commencent par /group/exemple_rest/ .
- Les avantages de cette méthode sont nombreux :
 - **indépendance totale entre autorisations et applications** (couplage lâche)
 - **Point d'entrée unique** permettant de créer facilement logs et audits.
 - **Granularité des autorisations possible du niveau le plus global au niveau le plus détaillé**, l'ensemble des informations nécessaires étant disponibles dans l'URI logique et la méthode HTTP utilisée.

Exemple (1/6)

- Un traiteur propose sur son site plusieurs services à ses clients :
 - Obtenir la liste des plats disponibles
 - Obtenir des informations sur un plat précis
 - Passer une commande

Exemple (2/6)



Exemple (3/6)

- La liste des plats est disponible à l'URL suivante : <http://www.monresto.com/plats/>
- Le client reçoit une réponse sous la forme suivante :

```
<?xml version="1.0"?>
<p:Plats xmlns:p="http://www.monresto.com/"
xmlns:xlink="http://www.w3.org/1999/xlink">
    <Plat id="0001" xlink:href="http://www.monresto.com/Plats/0001"/>
    <Plat id="0002" xlink:href="http://www.monresto.com/Plats/0002"/>
    <Plat id="0003" xlink:href="http://www.monresto.com/Plats/0003"/>
[...]
</p:Plats>
```

Exemple (4/6)

- Les détails d'un plat se trouvent à l'URL :
<http://www.monresto.com/plats/0002>
- D'où la réponse :

```
<?xml version="1.0"?>
<p:Plat xmlns:p="http://www.monresto.com"
xmlns:xlink="http://www.w3.org/1999/xlink">

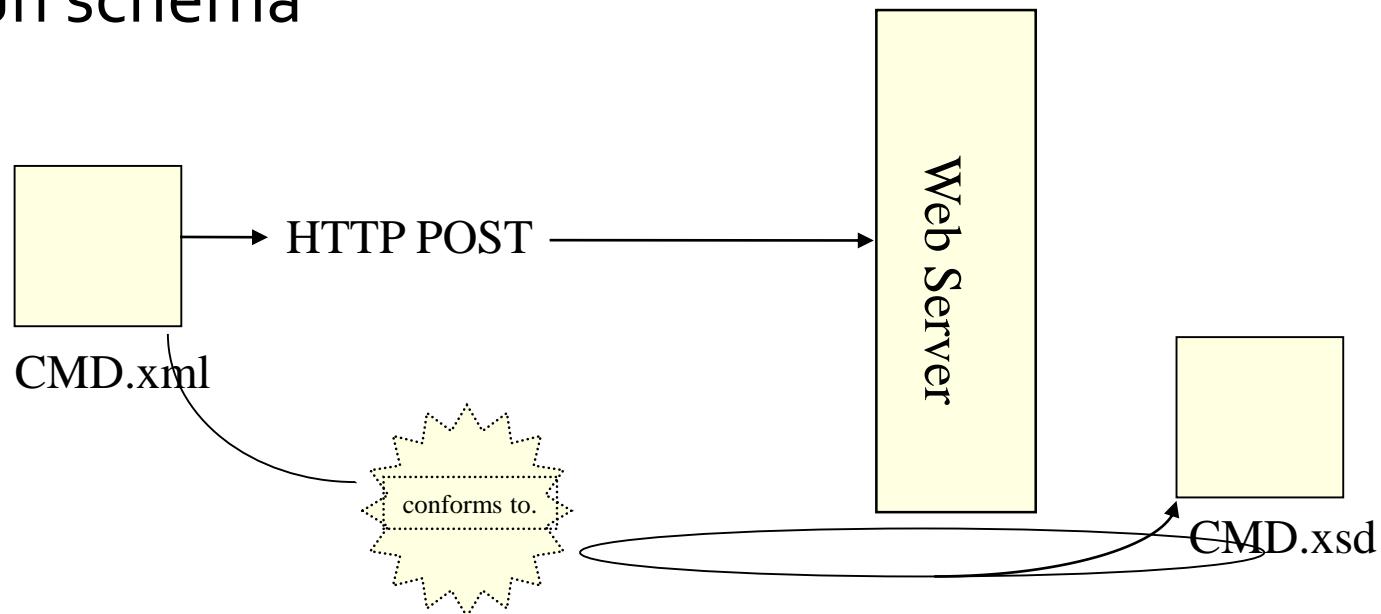
    <Plat-ID>0002</Plat-ID>
    <Nom>Rouleaux de Printemps</Nom>
    <Description>Entrée</Description>
    <Details xlink:href="http://www.monresto.com/plats/00002/details"/>
    <CoutUnitaire monnaire="EUR">3</CoutUnitaire>

</p:Plat>
```

Exemple (5/6)

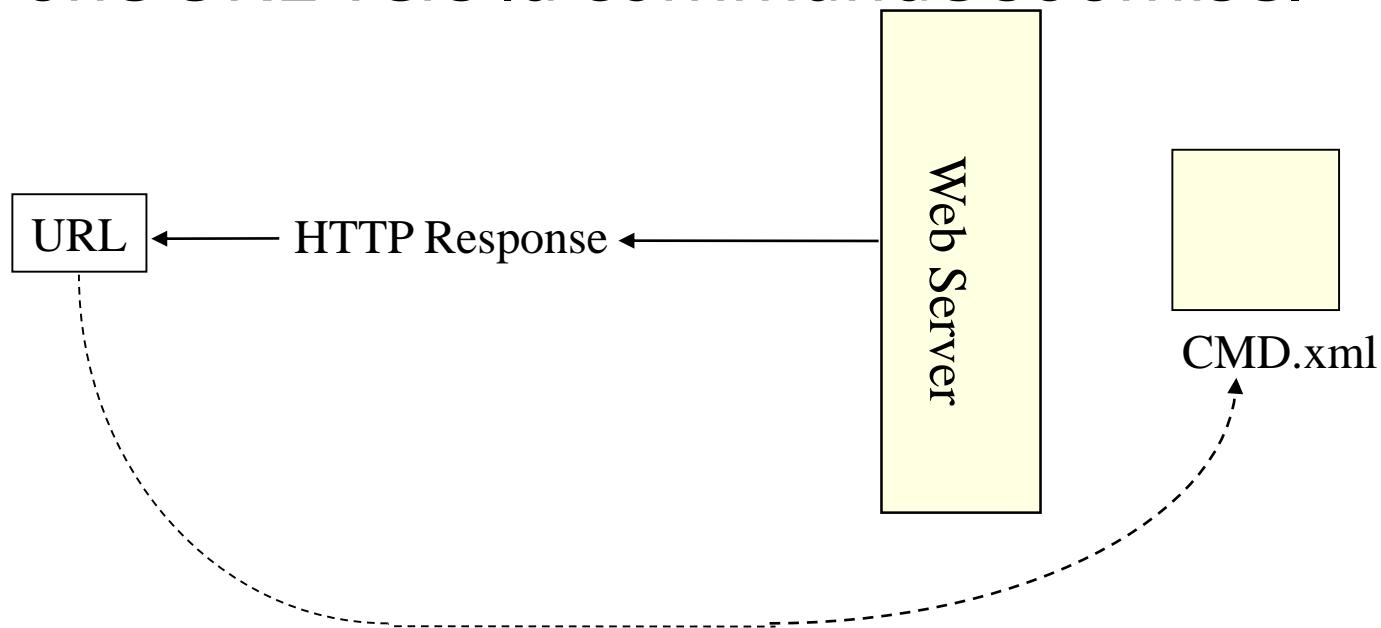
■ Le service « Passer commande »

- Créer une instance de « commande » conforme à un schéma



Exemple (6/6)

- Le service « Passer une commande » répond par une URL vers la commande soumise.



Cas d'utilisation

- La blogosphère
- Atom Publishing Protocol (RFC 5023)
- Amazon S3 (Simple Storage Service)
- Google Data API

Les avantages

- Simplicité
 - Liens structurés et de façon universelle
- Stateless
 - consommation de mémoire inférieure
 - mise au point plus simple, moins de cas à traiter
- URI uniques
 - mise en cache possible donc meilleure montée en charge
- Proche de la philosophie d'HTTP
 - moins de dépendances à une implémentation particulière

Les inconvénients

■ Stateless

- Les données de session chez le client

■ Un nombre important de ressources

- La gestion de l'espace de nom des URI peut devenir encombrante

What is REST?

- Representational State Transfer
- Maps your CRUD actions to HTTP verbs

Action	Verb
Create	POST
Retrieve	GET
Update	PUT
Delete	DELETE

Why REST?

- Simple, both conceptually and programmatically
- Simpler and cleaner than SOAP

SOAP Example

POST /InStock HTTP/1.1

Host: www.example.org

Content-Type: application/soap+xml; charset=utf-8

Content-Length: nnn

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
    soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
    <soap:Body xmlns:m="http://www.example.org/stock">
        <m:GetStockPrice>
            <m:StockName>IBM</m:StockName>
        </m:GetStockPrice>
    </soap:Body>
</soap:Envelope>
```

REST Example

```
GET /stock/IBM HTTP/1.1  
Host: www.example.org  
Accept: application/xml
```

SOAP Example 2

POST /InStock HTTP/1.1

Host: www.example.org

Content-Type: application/soap+xml; charset=utf-8

Content-Length: nnn

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-
    envelope" soap:encodingStyle="http://www.w3.org/2001/12/soap-
        encoding">
    <soap:Body xmlns:m="http://www.example.org/stock">
        <m:BuyStock>
            <m:StockName>IBM</m:StockName>
            <m:Quantity>50</m:Quantity>
        </m:BuyStock>
    </soap:Body>
</soap:Envelope>
```

REST Example 2

POST /order HTTP/1.1

Host: www.example.org

Content-Type: application/xml; charset=utf-8

```
<?xml version="1.0"?>
<order>
    <StockName>IBM</StockName>
    <Quantity>50</Quantity>
</order>
```

Single Resource Summary

- Create
 - POST /resourceName
- Retrieve
 - GET /resourceName/resourcId
- Update
 - PUT /resourceName/resourcId
- Delete
 - DELETE /resourceName/resourcId

Making it Easy: JSR 311

- JAX-RS: The Java API for RESTful Web Services
- No XML Configuration!
 - Simple annotations to quickly put together some rest based web services.
- Can do automatic serialization (both XML and JSON + many others)

Example Application w/ JaxRS

5. JAX-RS

- **JSR-311:** “JAX-RS: The Java API for RESTful Web Services”
- Introduced in Java SE 5: annotation-based
- Objective: simplify the development and deployment of web service clients and endpoints.
- Part of J2EE 6
- No configuration is necessary to start using JAX-RS

5. Making it Easy: JSR 311

- JAX-RS: The Java API for RESTful Web Services
- No XML Configuration!
 - Simple annotations to quickly put together some rest based web services.
- Can do automatic serialization (both XML and JSON + many others)

5. JAX-RS implementations

■ Implementations

- Apache, CFX from Apache
- **Jersey, from Sun/Oracle (J2EE 6 reference implementation)**
- RESTeasy, from Jboss
- Restlet, created by Jerome Louvel, a pioneer in REST frameworks.
- ...

5. JAX-RS API

• Specifications

- Annotations to aid in mapping a resource class (a Plain Old Java Object) as a web resource.
 - **@Path** specifies the relative path for a resource class or method.
 - **@GET, @PUT, @POST, @DELETE** and **@HEAD** specify the HTTP request type of a resource.
 - **@Produces** specifies the response MIME media types.
 - **@Consumes** specifies the accepted request media types.
- Annotations to method parameters to pull information out of the request. All the **@*Param** annotations take a key of some form which is used to look up the value required.
 - **@PathParam** binds the parameter to a path segment.
 - **@QueryParam** binds the parameter to the value of an HTTP query parameter.
 - **@MatrixParam** binds the parameter to the value of an HTTP matrix parameter.
 - **@HeaderParam** binds the parameter to an HTTP header value.
 - **@CookieParam** binds the parameter to a cookie value.
 - **@FormParam** binds the parameter to a form value.
 - **@DefaultValue** specifies a default value for the above bindings when the key is not found.
 - **@Context** returns the entire context of the object. Ex.(@Context HttpServletRequest request)

5. JAX-RS API

```
@Path("/helloworld")
public class HelloWorldResource {

    // The Java method will process HTTP GET requests
    @GET
    // The Java method will produce content identified by the MIME Media
    // type "text/plain"
    @Produces("text/plain")
    public String getClichedMessage() {
        // Return some cliched textual content
        return "Hello World";
    }
}
```

5. REST Web-services

■ Example

- base URI for the web service: *http://example.com/resources/*

Resource	GET	PUT	POST	DELETE
Collection URI, such as <i>http://example.com/resources/</i>	List the URIs and perhaps other details of the collection's members.	Replace the entire collection with another collection.	Create a new entry in the collection. The new entry's URL is assigned automatically and is usually returned by the operation.	Delete the entire collection.
Element URI, such as <i>http://example.com/resources/item17</i>	Retrieve a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	Replace the addressed member of the collection, or if it doesn't exist, create it.	Not generally used. Treat the addressed member as a collection in its own right and create a new entry in it.	Delete the addressed member of the collection.

5. SOAP Example

POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
    soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
    <soap:Body xmlns:m="http://www.example.org/stock">
        <m:GetStockPrice>
            <m:StockName>IBM</m:StockName>
        </m:GetStockPrice>
    </soap:Body>
</soap:Envelope>
```

5. REST Example

```
GET /stock/IBM HTTP/1.1  
Host: www.example.org  
Accept: application/xml
```

5. SOAP Example 2

POST /InStock HTTP/1.1

Host: www.example.org

Content-Type: application/soap+xml; charset=utf-8

Content-Length: nnn

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-
    envelope" soap:encodingStyle="http://www.w3.org/2001/12/soap-
        encoding">
    <soap:Body xmlns:m="http://www.example.org/stock">
        <m:BuyStock>
            <m:StockName>IBM</m:StockName>
            <m:Quantity>50</m:Quantity>
        </m:BuyStock>
    </soap:Body>
</soap:Envelope>
```

REST Example 2

POST /order HTTP/1.1

Host: www.example.org

Content-Type: application/xml; charset=utf-8

```
<?xml version="1.0"?>
<order>
    <StockName>IBM</StockName>
    <Quantity>50</Quantity>
</order>
```

Synthèse

- Une architecture orientée ressources
- Du vieux pour faire du neuf
- Simplicité et interopérabilité