
S.O.A.
(architecture
orientée service)
-
concepts et
éléments
*(Services web , ESB,
orchestration, cloud)*

Table des matières

I - Apport et enjeux de l'architecture SOA.....	4
1. SOA (présentation).....	4
2. SOA au niveau du Middle office.....	6
3. SOA et Intégration.....	7
4. Paradigme "tout est service".....	8
5. Gains de l'approche SOA et enjeux.....	9
6. Des composants aux services.....	9
7. Repères SOA (architecture , technologie, ...).....	11
II - WEB-Services.....	13
1. Deux grands types de WS (REST et SOAP).....	13
2. Protocole SOAP.....	14
3. Eléments du protocole SOAP.....	15
4. WSDL (Web Service Description Language).....	17
5. Détails sur WSDL.....	19
6. Utilité d'un annuaire de "service WEB".....	22
7. Annuaires UDDI (présentation).....	24
8. Annuaires publics et annuaires privés.....	24
9. taxonomies (au sein des annuaires).....	25
10. Web Services "R.E.S.T.".....	27
11. Gestion de la sécurité / Services Web.....	31
III - EAI et ESB.....	36
1. EAI et ESB.....	36
2. EAI (Enterprise Application Integration).....	41
3. ESB (Enterprise Service Bus).....	44
4. Notion de "moteur de services".....	49
5. Catégories d'ESB.....	50
6. SOA et mode asynchrone.....	53
7. Mule Esb (présentation).....	55
8. Présentation de l'ESB "ServiceMix".....	56
IV - Orchestration de services.....	59
1. Notion d'orchestration et traçabilité BPM.....	59
2. BPEL (présentation).....	61
3. BPEL en mode asynchrone.....	67

4. BPMN.....	69
5. jBPM et activiti (présentation).....	71
6. UserTask.....	73
V - Le "Cloud Computing".....	74
1. Le "Cloud Computing" (présentation).....	74
VI - Annexe - Api JAX-WS (pour SOAP) et CXF.....	79
1. Présentation de JAX-WS.....	79
2. Mise en oeuvre de JAX-WS (coté serveur).....	79
3. Tests via SOAPUI.....	83
Utilisation de JAX-WS coté client.....	84
4. Redéfinition de l'URL d'un service à invoquer.....	85
5. Client JAX-WS sans wsimport et directement basé sur l'interface java.....	86
6. CXF.....	87
VII - Annexe – WS REST en java.....	91
1. API java pour REST (JAX-RS).....	91

I - Apport et enjeux de l'architecture SOA

1. SOA (présentation)

1.1. Agilité visée dans l'architecture SOA

Architecture **SOA** (*orientée services*)

SOA:

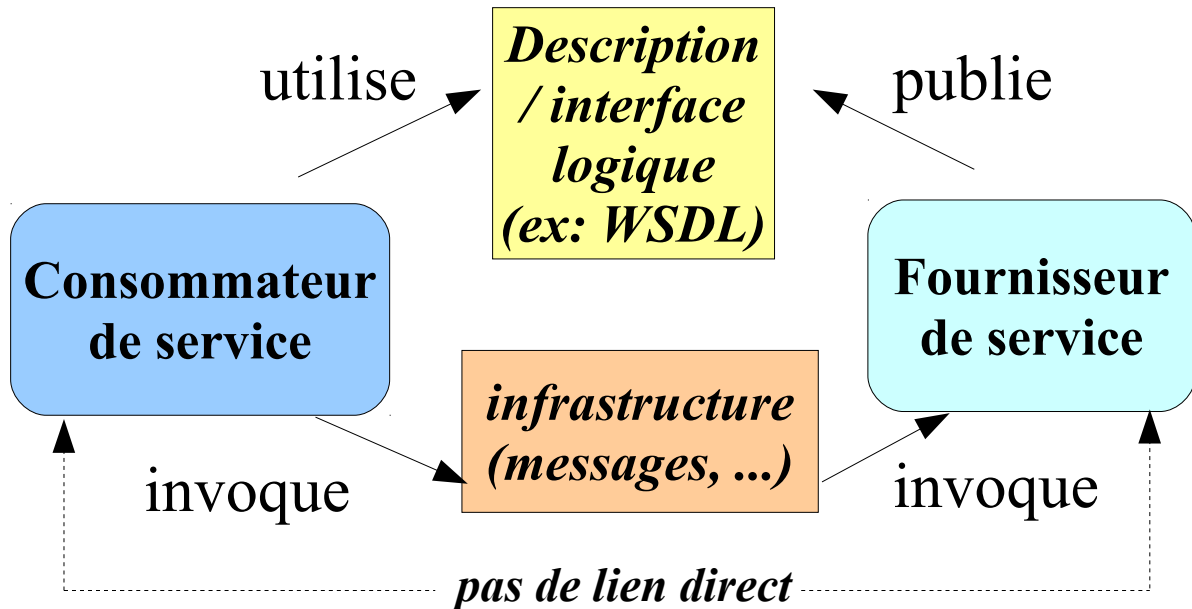
- **faible couplage entre les différents services**
- **découplage entre invocation et implémentation (avec éventuels intermédiaires [adaptateur,...])**
- **fait ressortir l'aspect logique (services rendus)**
==> *manifestations:*
 - * *infrastructure & organisationnel* ==> **ESB**
 - * *fonctionnel / orga.* ==> **BPM / BPEL**, pivot , urba.
 - * *technique* ==> **service web** (*interface fonctionnelle + port logique d'invocation dans .WSDL et url vers point d'accès à une certaine implémentation*)

Une **architecture "SOA"** vise essentiellement à mettre en œuvre un **S.I. Agile** :

- souple
 - modulaire
- facilement reconfigurable

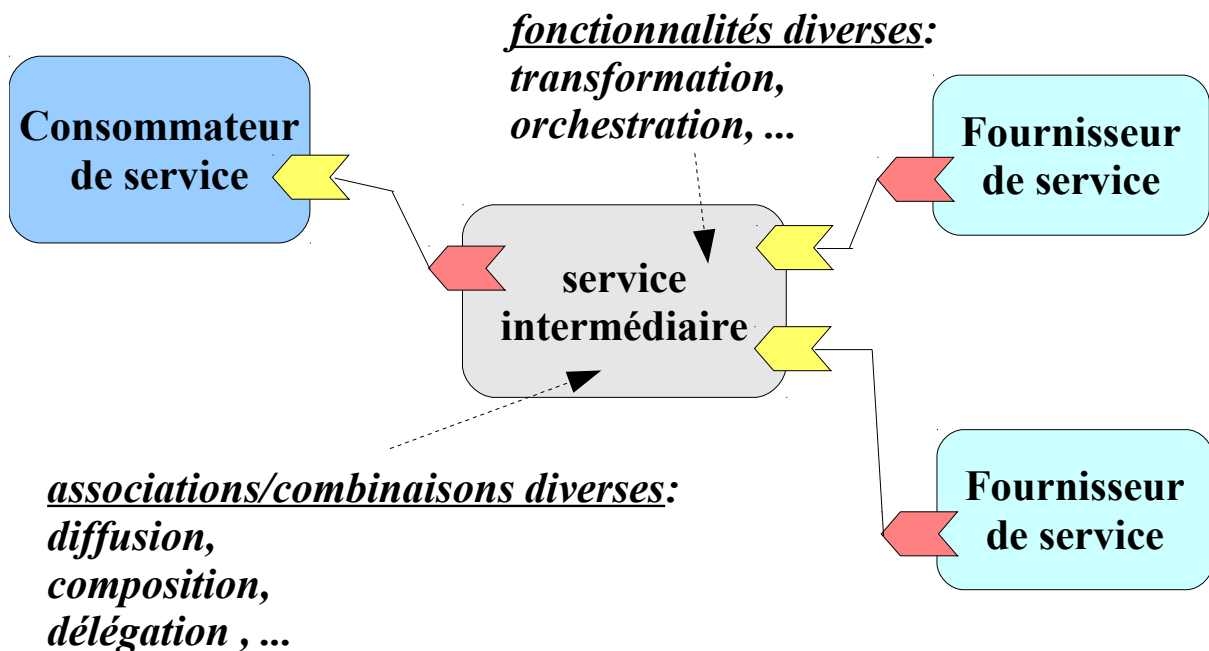
1.2. Découplages entres invocations et implémentations

SOA : Combinaison de services sur le mode "fournisseurs/consommateurs faiblement couplés"

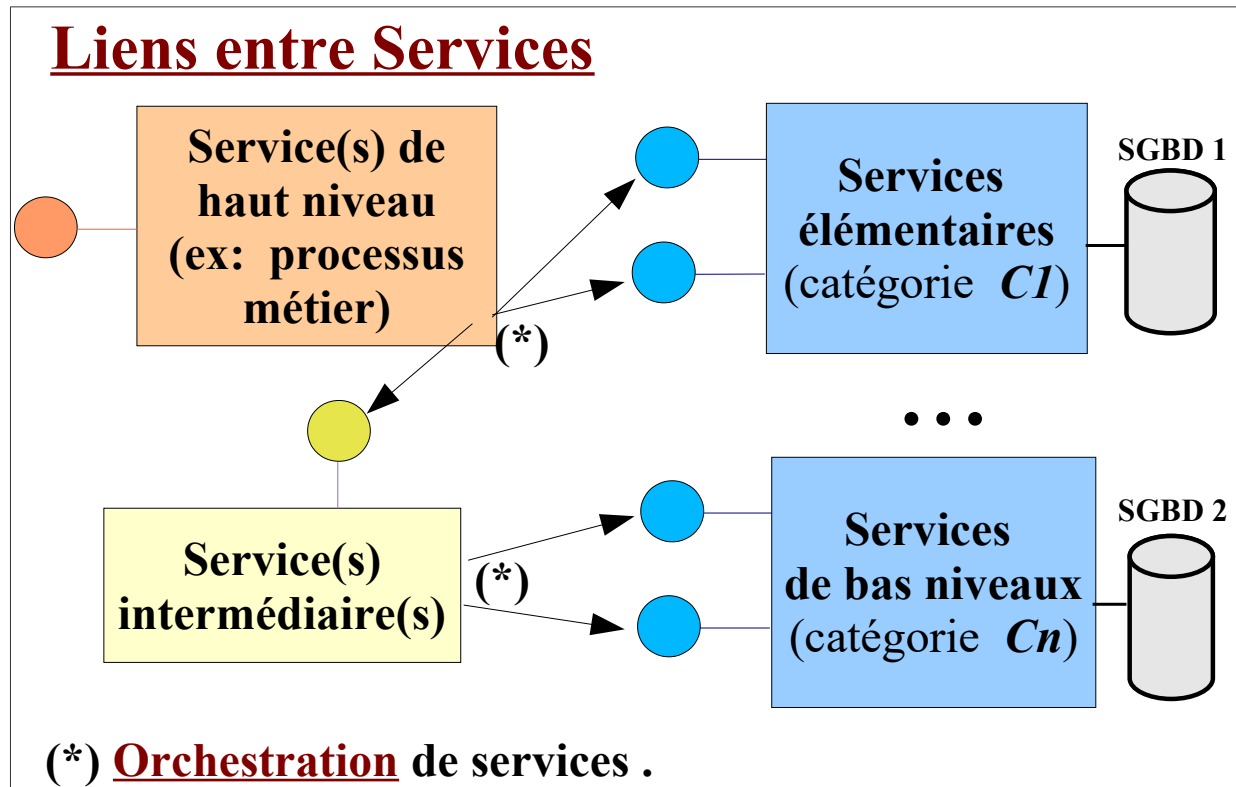


1.3. Consommation et offre de services

SOA : Services intermédiaires = fournisseurs et consommateurs.



1.4. Différents niveaux de services



2. SOA au niveau du Middle office

Beaucoup d'éléments de l'architecture SOA (*ESB, adaptateurs, processus BPEL, ...*) sont utilisés sur la partie "**Middle Office**" d'un SI d'une assez grande entreprise .

Autrement dit ,

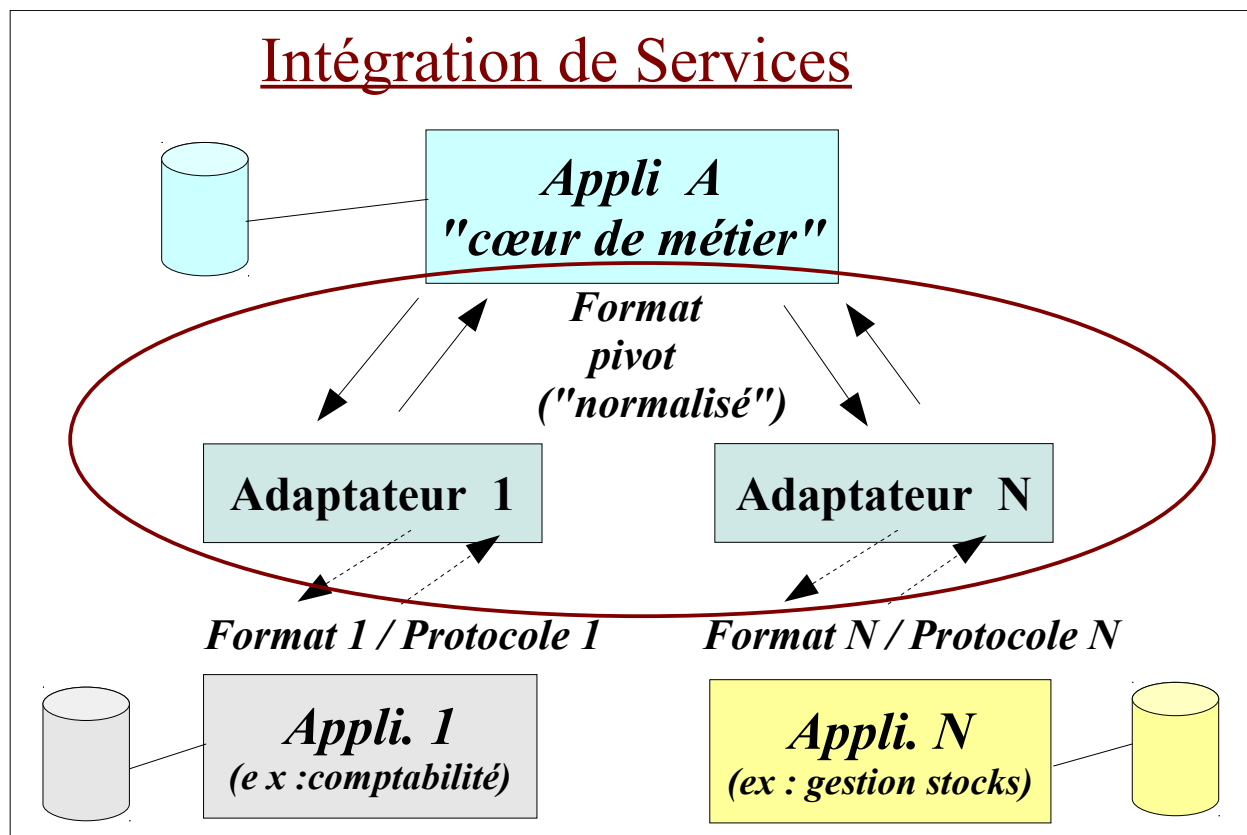
- Mise à part les web services REST, SOA n'est pas souvent directement lié à la partie IHM/web (c'est le rôle de la partie front-office)
- SOA n'est que rarement lié aux SGBDR (c'est le rôle de la partie back-office)
- **SOA est avant tout à voir comme un système fonctionnel .**

3. SOA et Intégration

L'architecture SOA est souvent utilisée pour **intégrer** dans un **SI spécifique au cœur de métier** des **applications existantes (non développées en interne mais achetées)** qui **rendent des services transverses ou génériques** (ex: Compta , gestion de Stock, ...).

Pour faire communiquer entre elles des applications provenant de différents éditeurs , on a besoin de tout un tas d'éléments **intermédiaires** fournis par l'**infrastructure SOA** (*ESB* , *Adaptateurs*, ...).

Dans le cas d'une application "clef en main" achetée auprès d'un éditeur de logiciels, on ne maîtrise absolument pas le format exact des services offerts (liste des méthodes , structures des données en entrées et en sorties des appels). Via des paramétrages de transformations "ad-hoc" effectuées au niveau d'un ESB, il sera (en général) tout de même possible d'invoquer cette application depuis d'autres applications internes au SI de l'entreprise.



Une ancienne norme SOA du monde Java s'appelait JBI (Java Business **I**ntegration)

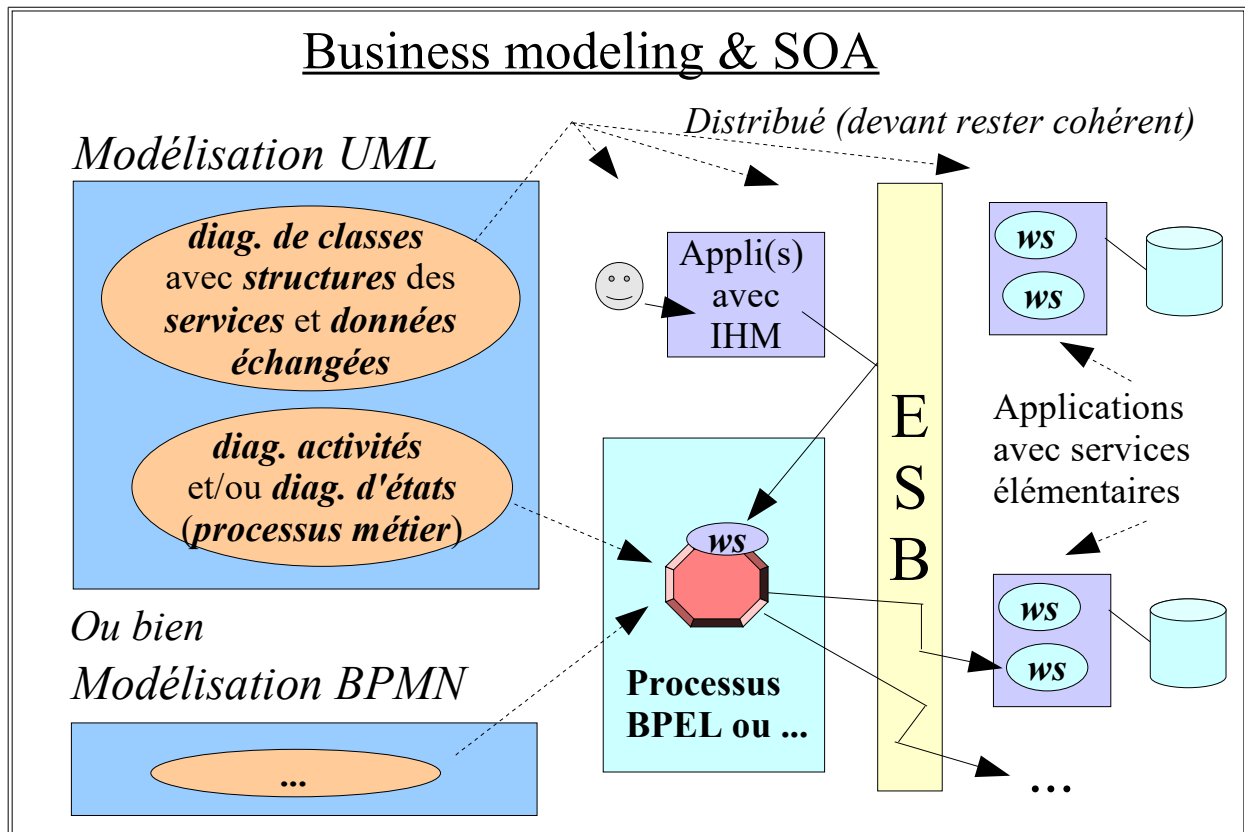
Attention, attention :

Les "Web-Services", ESB et "Processus BPM" ne constituent que l'*infrastructure technique*.

Un vrai projet SOA d'entreprise doit absolument comporter en outre un volet "**MODELISATION**" (avec UML et/ou BPMN) qui est vraiment **fondamental** !!!

SOA c'est un peu "**programmer/configurer le SI à grande échelle**".

Vue la grande portée de SOA, l'aspect modélisation est très important .



4. Paradigme "tout est service"

Au sein d'une architecture SOA , on peut (dans une première approche conceptuelle) considérer que presque tous les éléments d'un système informatique sont des services :

- services de persistance (lectures & mises à jour dans une base de données ou ...)
- services d'habilitation (vérifier les privilèges d'accès , ...)
- services de vérifications , contrôles ,
- services métiers
-

Mise à part quelques contraintes techniques dont il faut tenir compte (transactions courtes proches des données , performances correctes , sécurité ,) , la plupart des services (ou sous services) peuvent se mettre en œuvre de manières très diverses :

- prise en charge directe par l'entreprise ou bien sous traitance (SAAS ,) .
- service local , proche ou lointain (géographiquement) .
- service programmé sur mesure ou bien "acheté + adapté"
-

==> Il convient donc d'effectuer une première phase de modélisation SOA en prenant du recul et en ne se focalisant que sur des considérations purement sémantiques/fonctionnels .

Beaucoup d'autres aspects ("organisationnel" , "logiciel" , "géographique" ,) sont idéalement à modéliser comme des aspects séparés (que l'on règle par paramétrages ou via des implémentations dédiées) .

5. Gains de l'approche SOA et enjeux

5.1. Principaux apports de l'approche SOA :

- **interopérabilité** (java , .net , c++ , php , ...)
- **flexibilité/agilité** (relocalisations et/ou reconfigurations possibles des services via intermédiaire de type ESB)
- **partage de traitements métiers** (règles métiers , services d'entreprise , ...)
- **partage de données** (accessibles via les services) et donc moins de besoin(s) de "re-saisie" ou de "copie/réplication" .
- **urbanisation plus aisée** (chaque partie peut évoluer à son rythme tant que certains contrats fonctionnels sont respectés en utilisant s'il le faut des adaptateurs).
- ...

Ces apports sont tout de même **conditionnés par** :

- le besoin quasi-impératif d'une **bonne modélisation** SOA (à caractère *transverse*)
- l'**utilisation maîtrisée de technologies SOA** (WS,ESB,...) fiables et performantes.

5.2. Enjeux pour l'entreprise :

- **Potentielles économies d'échelle** (si partage effectif de services mutualisés).
- **Meilleur réactivité** (du fait d'un SI plus souple/flexible/agile) .
- **Meilleurs communications externes** (B2C , B2B) via des services publics visibles à l'extérieur de l'entreprise.
- ...

Et selon la qualité de la modélisation et de la mise en œuvre :

- **Peut être une maintenance plus aisée**
(il est plus facile de maintenir ou faire évoluer plusieurs petites applications assez indépendantes et qui communiquent via SOA que de maintenir une énorme application monolithique).
- Des performances à peu près aussi bonnes (si ESB bien choisi et bien paramétré).

5.3. Enjeux stratégiques (liés au cloud-computing)

Le concept de **cloud-computing** (à dimensionnement flexible) est actuellement à la mode et considéré quelquefois comme un enjeu stratégique .

La mise en œuvre du "**cloud-computing**" s'appuie essentiellement sur :

- **SaaS** (software as a service)
- **Paas** (platform as a service)
- **Iaas** (infrastructure as a service)

6. Des composants aux services

Un **composant** est une brique de "code" (généralement orienté objet) qui **ne peut être réutilisé qu'au sein d'un environnement technique bien spécifique** .

Exemples :

- Un composant "EJB3" ne peut être utilisé qu'au sein d'un serveur d'application JEE5 (ou 6).
- Un composant "Spring" ne peut être utilisé qu'au sein d'une application Java utilisant le framework Spring.
- Un composant ".NET" (programmé en C#) ne peut être réutilisé qu'au sein une application

utilisant le framework ".NET" sur une machine Windows (de Microsoft).

- ...

Un **composant** se **ré-utilise** par **copie/intégration d'un code compilé** généralement *packagé sous forme de librairie ou de module* (ex : *xy.dll* , *xy.lib* , *xy.jar* , ...).

Un composant est ré-utilisable s'il respecte un contrat (interface) d'intégration technique.

Chaque application qui ré-utilise un composant prendra en charge une instance spécifique (dans une certaine version, avec un certain paramétrage,...).

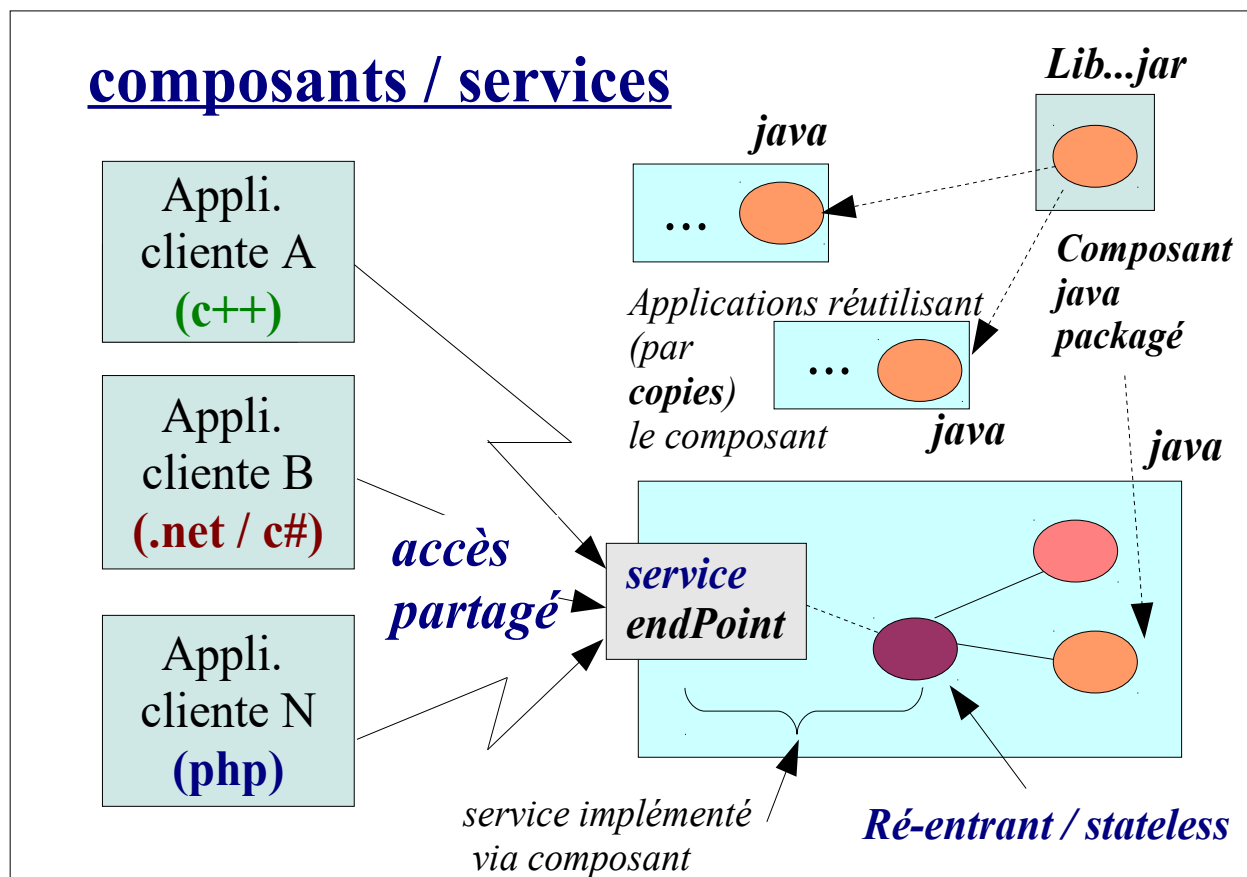
Un **service** est avant tout un **élément de traitement accessible et potentiellement partageable** .

Une fois publié (avec une URL connue et stable) , un service peut (si les contraintes de sécurité le permettent) être invoqué par une multitude d'applications "clientes" .

Des évolutions effectuées au niveau d'un service en place seront alors tout de suite vues au niveau de toutes les applications clientes qui l'invoquent.

Un **service** est généralement "**ré-entrant**" et se **ré-utilise** par "**partage d'accès fonctionnel**".

Un service est ré-utilisable s'il respecte un contrat (interface) fonctionnel (ex : WSDL) .



Lien entre composant et service (composant "service web") :

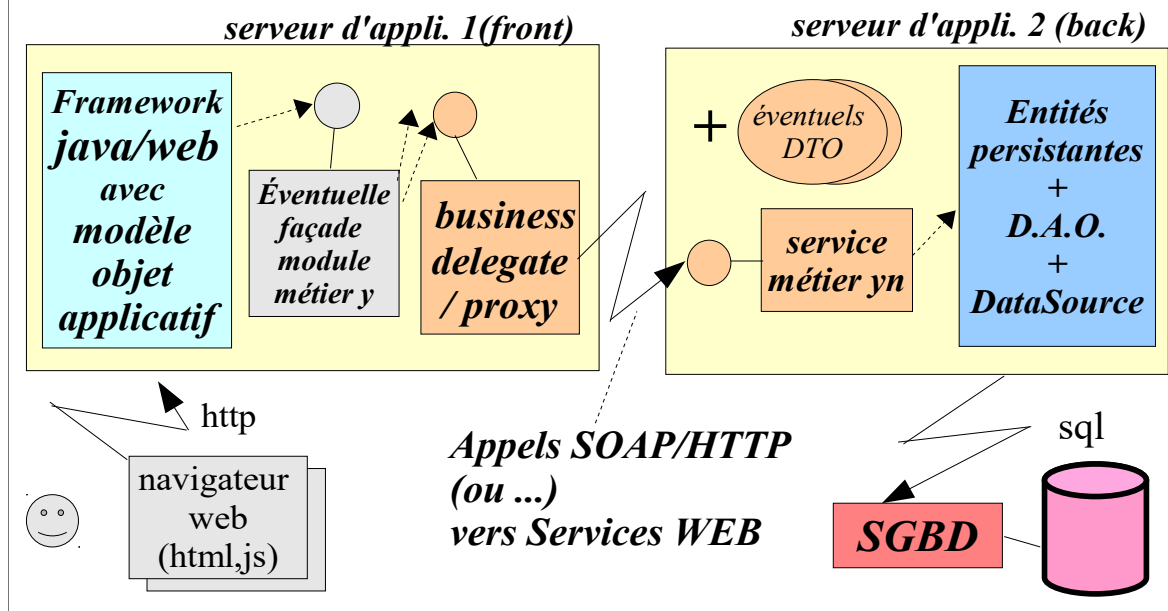
Dans beaucoup d'environnement technique (Java/JEE ,.NET , ...), un service WEB est mise en œuvre à partir de composant(s).

On pourra considérer qu'un service (publié au bout d'une URL) est une instance (accessible et partagée) des fonctionnalités offertes par un composant technique d'une application traditionnelle. Cette ouverture extérieure est généralement accomplie via la publication d'un point d'accès (*endpoint*) accessible vers les fonctionnalités d'un composant interne.

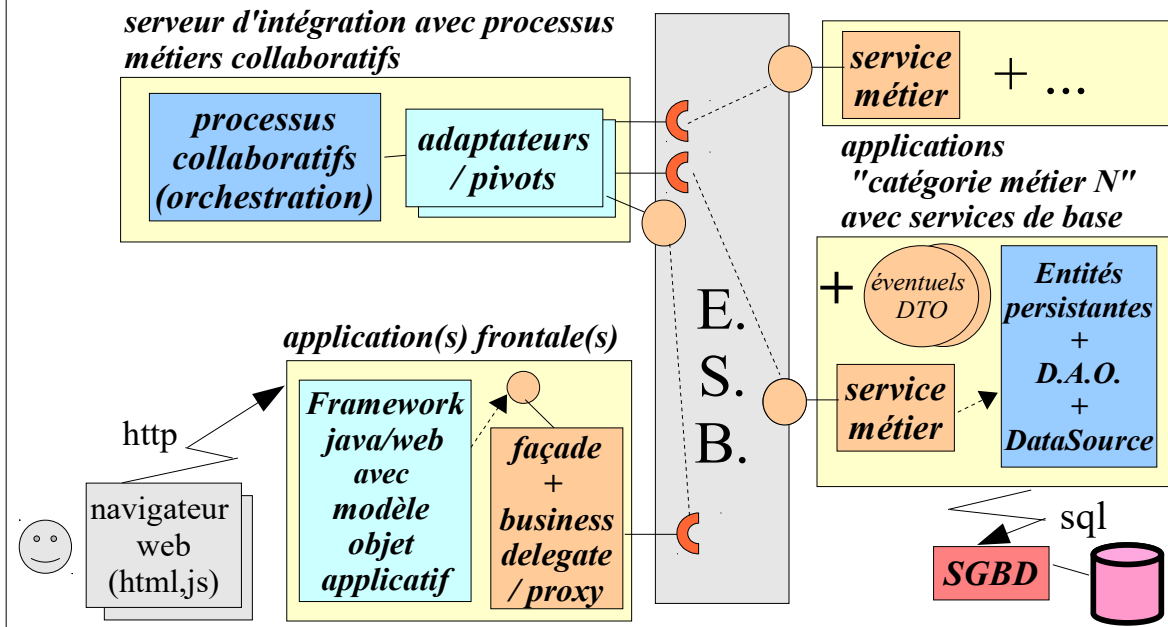
7. Repères SOA (architecture , technologie, ...)

7.1. Place de SOA dans une architecture n-tiers

Architecture 3-tiers **distribuée**: Application Web, services métiers sur **2 serveurs distincts**.

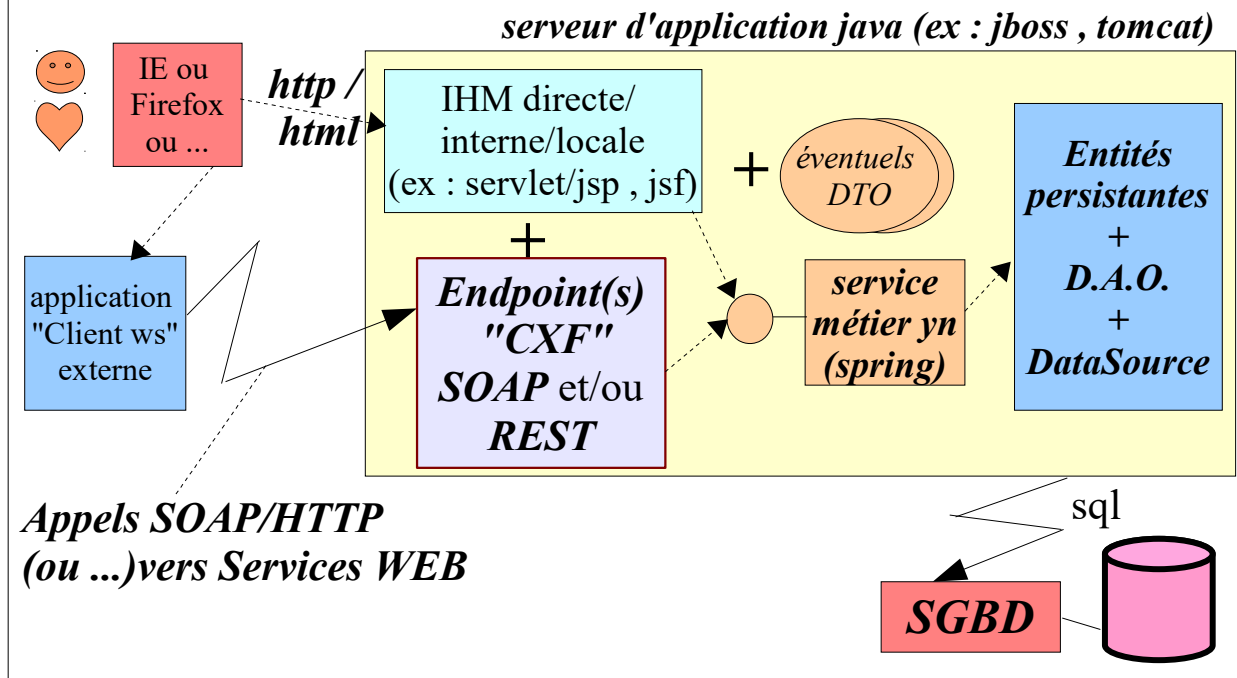


Architecture n-tiers **orientée services (S.O.A.)**: avec Bus "**E.S.B.**" et **processus collaboratifs**.



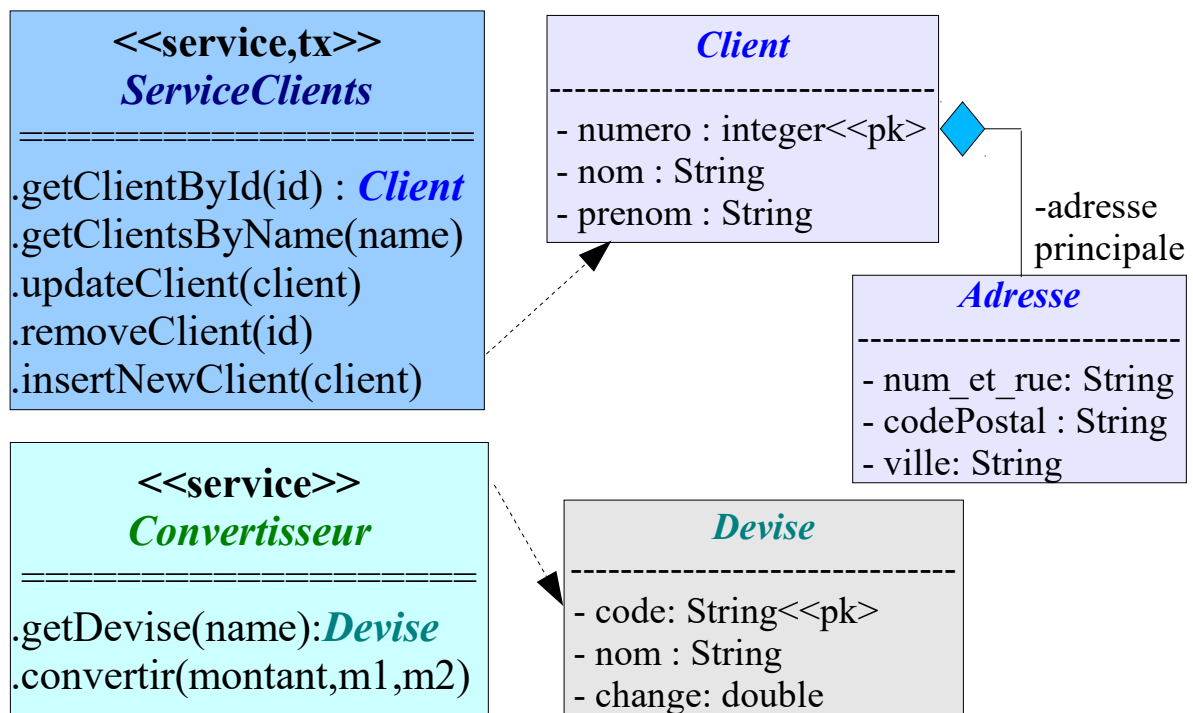
7.2. Exemple de mise en œuvre en java

WebService : Exemple de mise en œuvre avec Spring + CXF (en java) :



7.3. Modélisation essentielle (UML) d'un service web

Modélisation UML essentielle d'un Webservice (exemples)



II - WEB-Services

1. Deux grands types de WS (REST et SOAP)

2 grands types de services WEB: **SOAP/XML** et **REST**

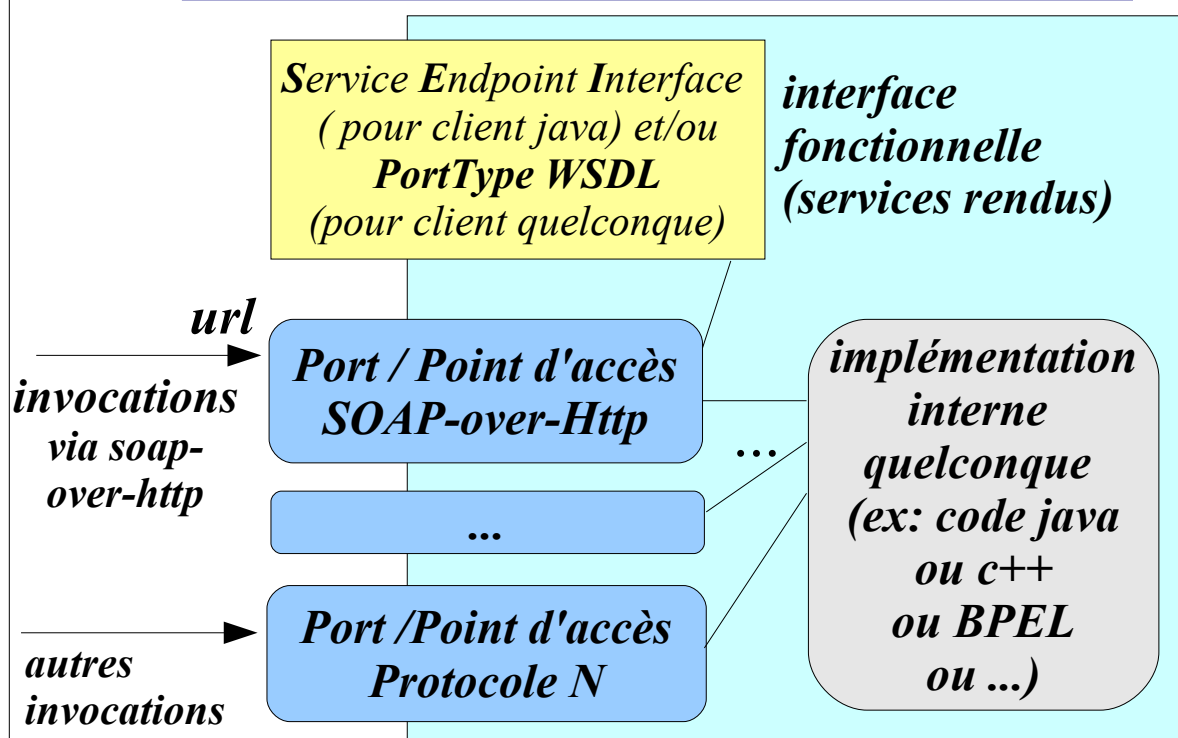
WS-* (SOAP / XML)

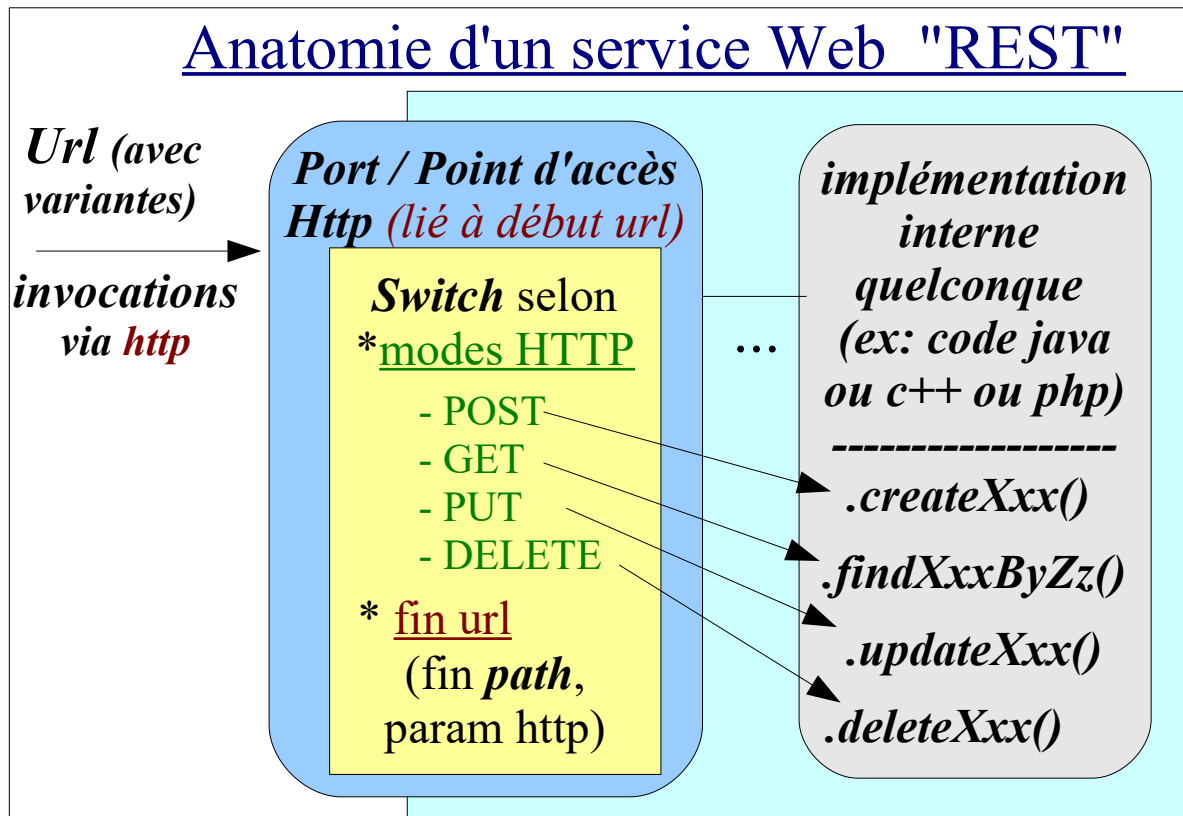
- "Payload" systématiquement en **XML** (sauf pièces attachées / HTTP)
- **Enveloppe SOAP** en XML (header facultatif pour extensions)
- Protocole de transport au choix (HTTP, JMS, ...)
- Sémantique quelconque, **description WSDL**
- Plutôt orienté SOA / proche "back office"

REST (HTTP)

- "Payload" au choix (XML, HTML, JSON, ...)
- Pas d'enveloppe imposée
- **Protocole de transport = toujours HTTP.**
- Sémantique "CRUD" (modes *http* PUT, GET, POST, DELETE)
- Plutôt orienté Web/Web2 et proche "front office"

Anatomie d'un service Web "SOAP"



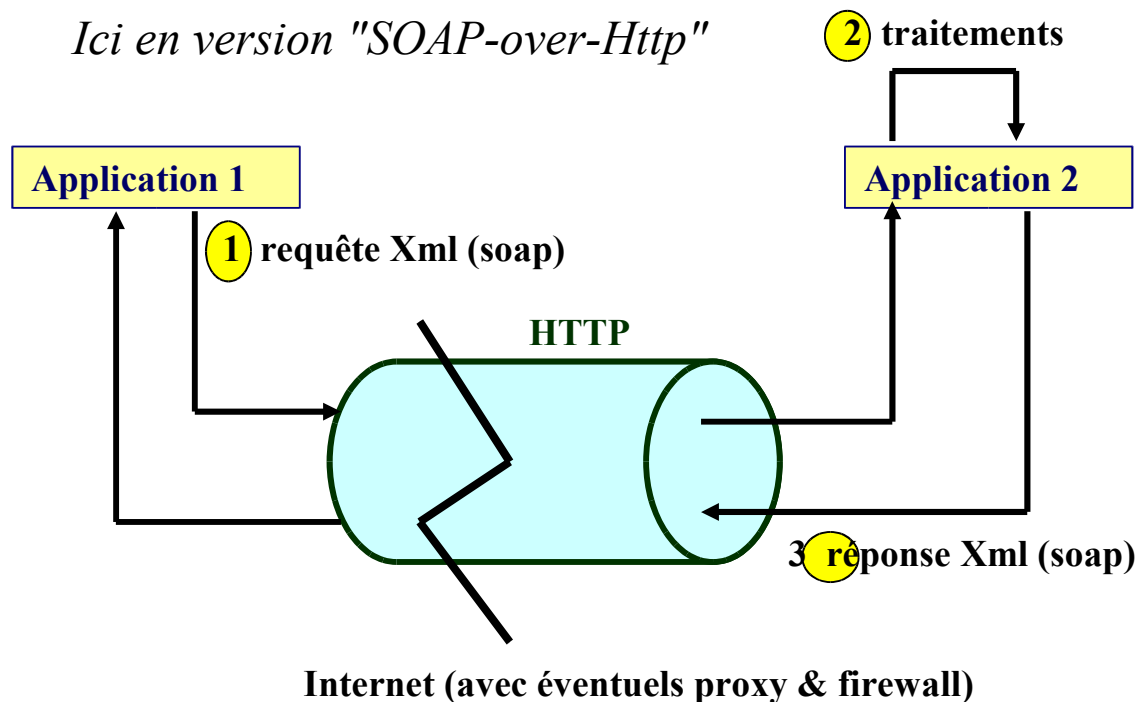


2. Protocole SOAP

2.1. SOAP on HTTP en mode synchrone (RPC)

SOAP (*Simple Object Access Protocol*)

Ici en version "SOAP-over-Http"



2.2. SOAP on JMS (ou SMTP) en mode asynchrone

L'enveloppe SOAP véhiculée est la même qu'avec SOAP over HTTP.
Seul le protocole de transfert est variable ==> SMTP ,

Un mode asynchrone avec de véritable "files d'attentes" ou ("mail box") peut quelquefois est très pratique.

3. Éléments du protocole SOAP

Bien que le sigle **S.O.A.P.** sous entende principalement l'accès distant à un objet de façon à y invoquer des fonctions, le protocole **SOAP** est avant tout basé sur le concept d'**envoi de messages** (message de requête, message de réponse).

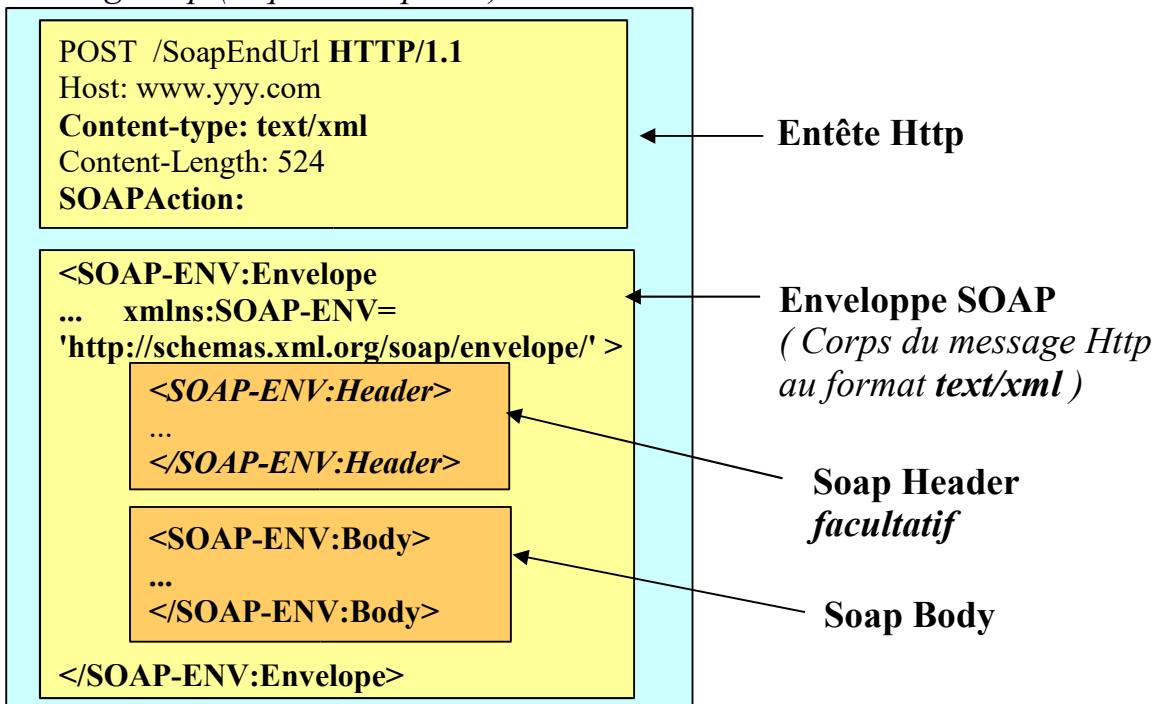
Ces messages peuvent tout aussi bien comporter des appels de fonctions (avec des paramètres valués) que comporter des messages textuels (éventuellement balisés en xml) et eux-même éventuellement accompagnés de pièces jointes.

Le **protocole SOAP** est constitué de **3 grandes parties (aspects complémentaires)**:

- L' **enveloppe SOAP** précisant le **contenu** du message avec ses parties facultative (header) et obligatoire (body) .
- L' **ENCodage SOAP** (sérialisation et dé-sérialisation des types de données)
NB: l'encodage SOAP normalisé au sein de la version 1.1 et correspondant au mode "rpc/encoded" est petit à petit supplanté par l'encodage "literal" .
- La représentation des appels de fonctions (**SOAP-RPC**) et des réponses (ou erreurs).

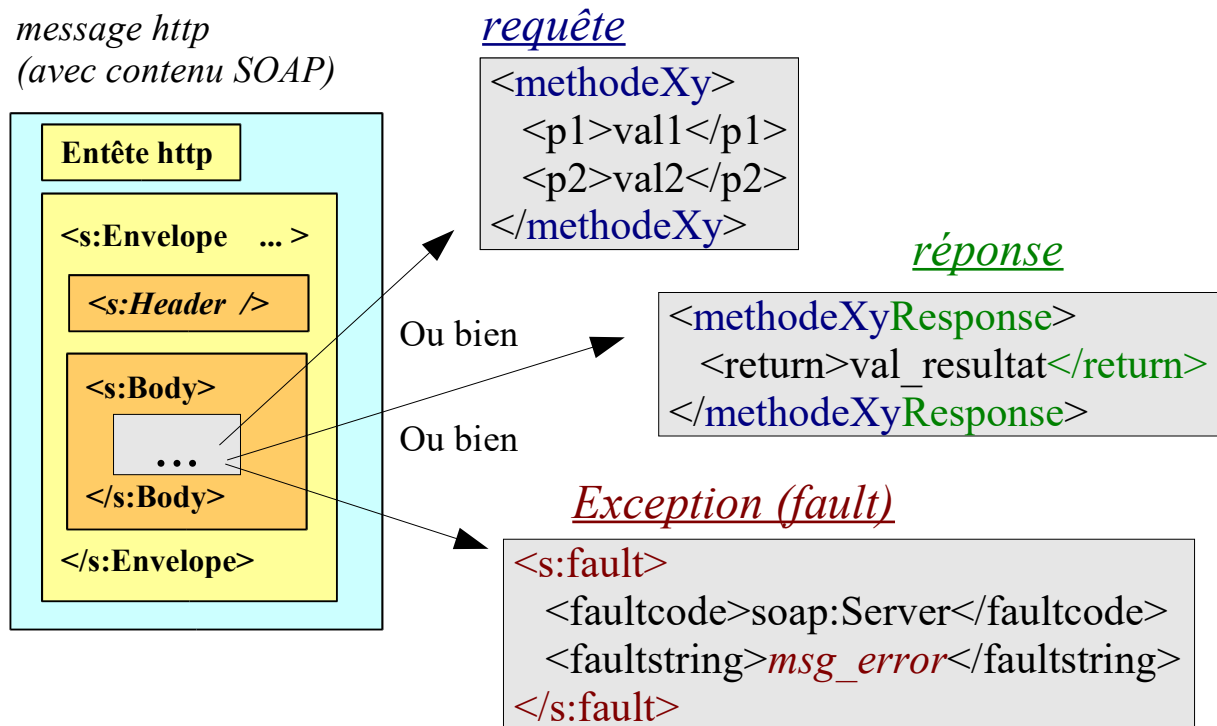
Enveloppe SOAP véhiculée via HTTP

message http (requête / réponse)

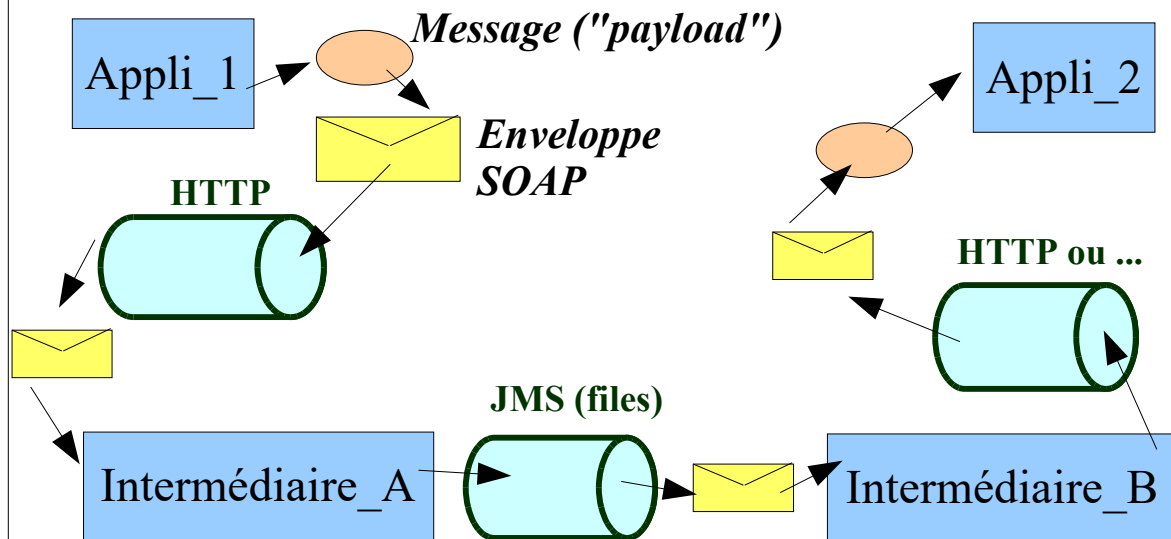


Contenu du corps de l'enveloppe "SOAP"

*message http
(avec contenu SOAP)*



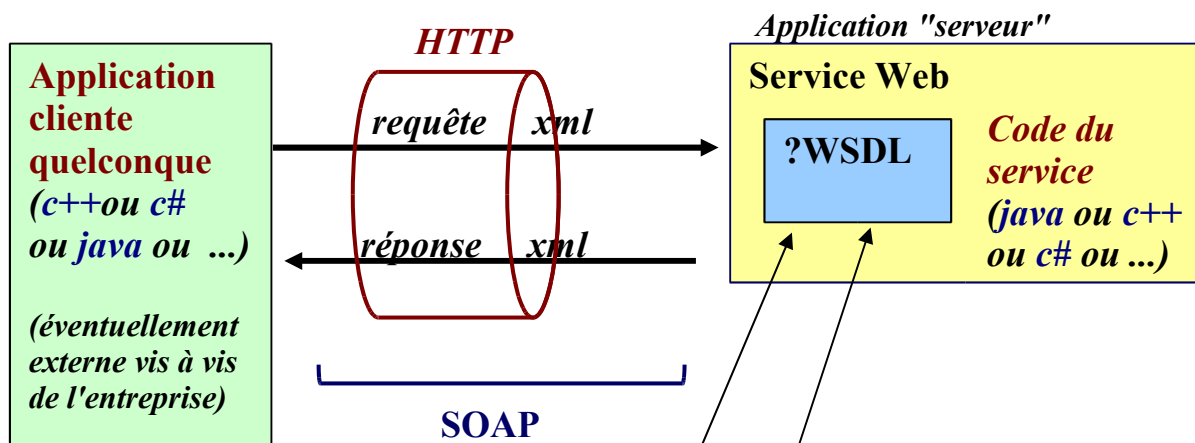
SOAP: "Payload", enveloppe et transports



Analogies: Lettre , enveloppe , acheminement (voiture + train + avion + ...)
Charge utile , conteneur , transport (train + bateau + camion + ...)

4. WSDL (Web Service Description Language)

Service Web (description WSDL)



Fichier de description **généré automatiquement côté serveur**, puis analysé par un programme du genre **Wsd2Java** pour générer un **"proxy"** côté client.

Description XML indépendante des langages de programmation (c++ ou java ou c# ou ...)

WSDL est un langage standard basé sur Xml qui a pour objectif de **décrire précisément les prototypes des fonctions activables au niveau d'un certain service web.**

Code coté serveur du *service web* en C++ (gSoap)

- génération du fichier WSDL via soapcpp2

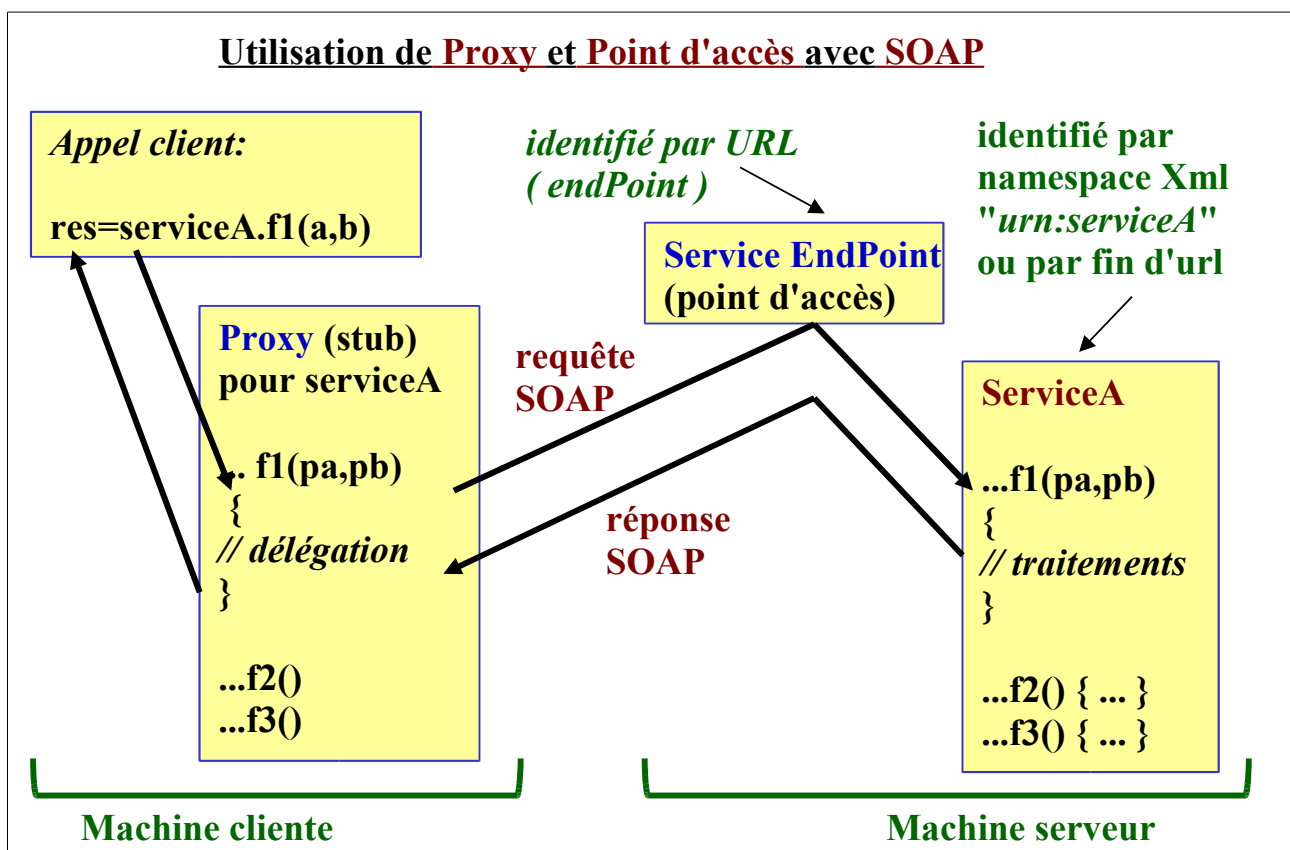
WSDL

- génération via **WSDL2Java** (ou **wsimport**)
d'un *proxy* pour client **Java**

Quelques définitions (WSDL):

message	paramètre(s) en entrée ou bien en sortie d'une opération
portType	Interface (ensemble d'opérations abstraites)
operation	couple (message d'entrée, message de retour) = méthode
binding	associer un portType à un protocole (SOAP, COBBA, DCOM) et à un manespace(un même URI qualifiant généralement tout un service web)
service	service Web (avec url) comportant un ensemble de port(s) = endpoint(s)

Utilisation de **Proxy** et **Point d'accès** avec **SOAP**



4.1. Versions et évolutions de SOAP/WSDL

Différentes versions (et évolution)

... , **SOAP 1.1** (xmlns="<http://schemas.xml.org/soap/envelope/>")
SOAP 1.2 (xmlns="http://www.w3.org/2003/05/soap-envelope")

Attention: SOAP 1.2 n'est pas pris en charge par toutes les technologies "WebService" et la version 1.1 suffit en général.

... , **WSDL 1.1** (xmlns="<http://schemas.xmlsoap.org/wsdl/>")
WSDL 2 (xmlns="<http://www.w3.org/ns/wsdl/>")

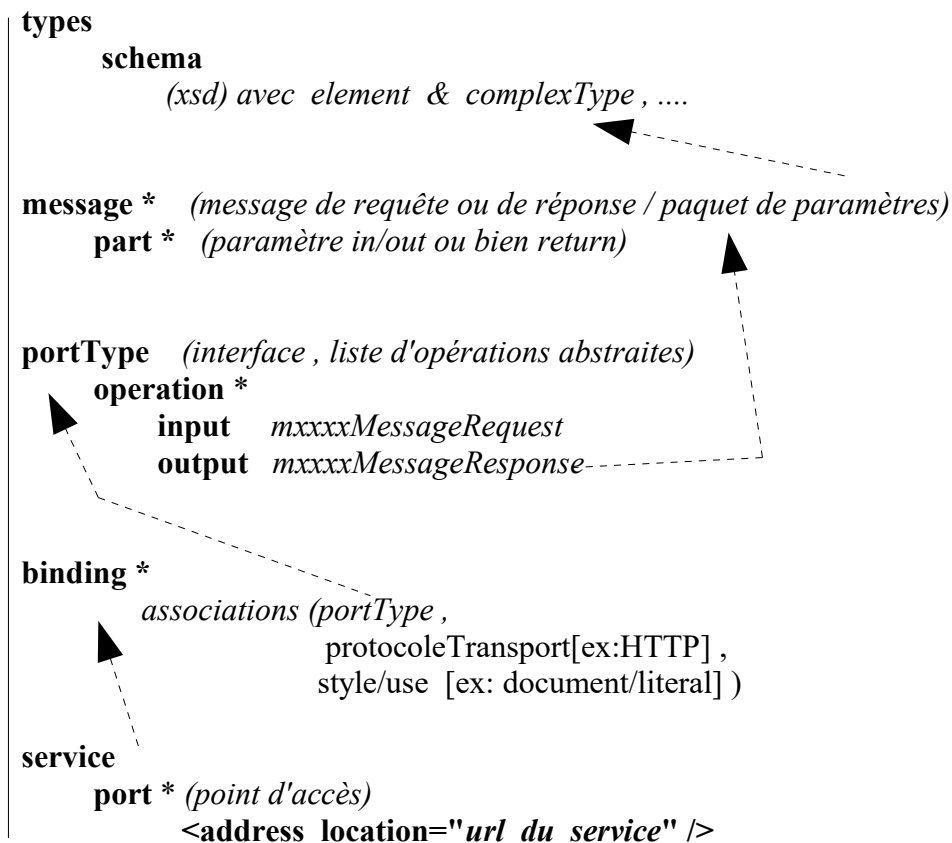
Attention: WSDL 2 n'est pas pris en charge par toutes les technologies "WebService" et la version 1.1 suffit en général. Cependant, certaines extensions WSDL2 (MEP) sont utilisées par certains produits (ex: ESB)

Pour SOAP, **style/use** = ~~rpc/encoded~~ (complexe et "has been")
 puis **rpc/literal** (plus performant, plus simple)
 puis **document/literal** (avec schéma "xsd" pour valider)

5. Détails sur WSDL

5.1. Structure générale d'un fichier WSDL

definitions



NB: Le contenu de la partie `<xsd:schema>` (sous la branche "wsdl:types") peut éventuellement correspondre à un sous fichier annexe "*xxx.xsd*" relié au fichier "*xxx.wsdl*" via un "*xsd:import*".

5.2. Exemple de fichier WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://calcul"
xmlns:apachesoap="http://xml.apache.org/xml-soap" xmlns:impl="http://calcul"
xmlns:intf="http://calcul" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.4 Built on Apr 22, 2006 (06:55:48 PDT)-->

<wsdl:types>
  <schema elementFormDefault="qualified" targetNamespace="http://calcul"
xmlns="http://www.w3.org/2001/XMLSchema">
    <element name="addition">
      <complexType>
        <sequence>
          <element name="a" type="xsd:int"/>
          <element name="b" type="xsd:int"/>
        </sequence>
      </complexType>
    </element>
    <element name="additionResponse">
      <complexType>
        <sequence>
          <element name="additionReturn" type="xsd:int"/>
        </sequence>
      </complexType>
    </element>
  </schema>
</wsdl:types>

<wsdl:message name="additionResponse"
  <wsdl:part element="impl:additionResponse" name="parameters"/>
</wsdl:message>

<wsdl:message name="additionRequest">
  <wsdl:part element="impl:addition" name="parameters"/>
</wsdl:message>

<wsdl:portType name="Calculator">
  <wsdl:operation name="addition">
    <wsdl:input message="impl:additionRequest" name="additionRequest"/>
    <wsdl:output message="impl:additionResponse" name="additionResponse"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="CalculatorSoapBinding" type="impl:Calculator">
  <wsdlsoap:binding style="document"
```

```

        transport="http://schemas.xmlsoap.org/soap/http"/>
<wsdl:operation name="addition">
  <wsdlsoap:operation soapAction=""/>

  <wsdl:input name="additionRequest">
    <wsdlsoap:body use="literal"/>
  </wsdl:input>

  <wsdl:output name="additionResponse">
    <wsdlsoap:body use="literal"/>
  </wsdl:output>
</wsdl:operation>
</wsdl:binding>

<wsdl:service name="CalculatorService">
  <wsdl:port binding="impl:CalculatorSoapBinding" name="Calculator">
    <wsdlsoap:address
      location="http://localhost:8080/axis2-webapp/services/Calculator"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>

```

5.3. WDSL abstraits et concrets

Structure **WSDL** (*abstrait* ou *concret*)

WSDL abstrait (*interface*)

```

<definitions ...>
  <types>
    <schema>
      <complexType .../> ...
      <element .../> ...
    </schema>
  </types>
  <message>...</message>
  <message>...</message>
  <portType>
    <operation ...>
      <input .../> <output .../>
    </operation>
    <operation>...</operation>
  </portType>
</definitions>

```

WSDL concret/complet (*impl*)

```

<definitions>
  <types>
    <schema ... />
  </types>
  <message>...</message> ...
  <portType>
    <operation .../> ...
  </portType>
  <binding>
    ... HTTP/SOAP style="document"
    ... use="literal" ...
  </binding>
  <service>
    <port><address location="...url..." />
  </port> </service>
</definitions>

```

Partie abstraite

Structure et sémantique WSDL (partie abstraite)

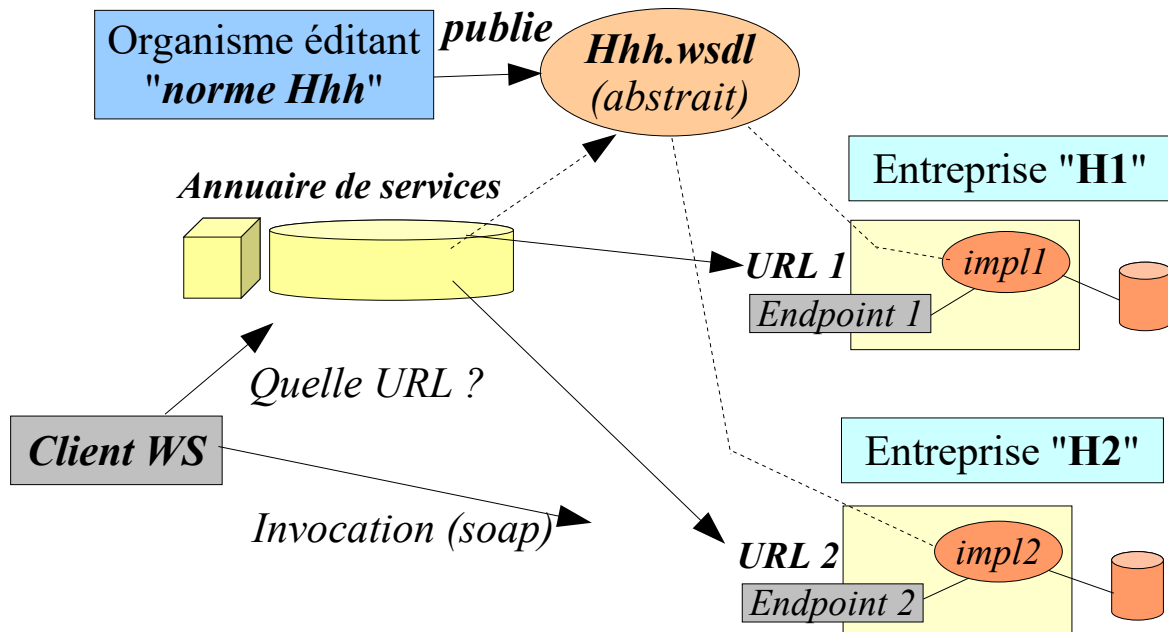
- Un *même "WSDL abstrait"* (idéalement issu d'une norme) pourra être physiquement implémenté par *plusieurs services distincts* (avec des *points d'accès (et URL) différents*).
- Un *portType* représente (en XML) l'*interface fonctionnelle* qui sera accessible depuis un futur "port" (*alias "endPoint"*)
- Chaque opération d'un "portType" est associée à des *messages* (structures des *requêtes[input]* et *réponses[output]*).
- En mode "*document/literal*", ces *messages* sont définis comme des *références* vers des "*element*"s (*xml/xsd*) qui ont la structure XML précise définie dans des "*complexType*".

Structure et sémantique WSDL (partie concrète)

- Un fichier "*WSDL concret*" comporte toute la partie abstraite précédente plus tous les paramétrages nécessaires pour pouvoir invoquer *un point d'accès précis sur le réseau*.
- Un *port* (*alias "endPoint"*) permet d'atteindre une *implémentation* physique d'un service publié sur un serveur. La principale information rattachée est l'**URL** permettant d'invoquer le service via SOAP.
- La partie "**binding**" (référéncée par un "port") permet d'*associer/relier* entre eux les différents éléments suivants:
 - * *interface abstraite (portType)* et ses opérations
 - * *protocole de transport (HTTP + détails ou)*
 - * *point d'accès (port)*

6. Utilité d'un annuaire de "service WEB"

Un service fonctionnel (avec WSDL abstrait) peut avoir **plusieurs implémentations** avec URL(s) à découvrir via un **annuaire**.



Un **annuaire de services** est une sorte de **base de données d'URL** avec des **critères de recherches**.

En général, dans une logique SOA, l'interrogation et la mise à jour de l'annuaire s'effectuent via des services WEB techniques.

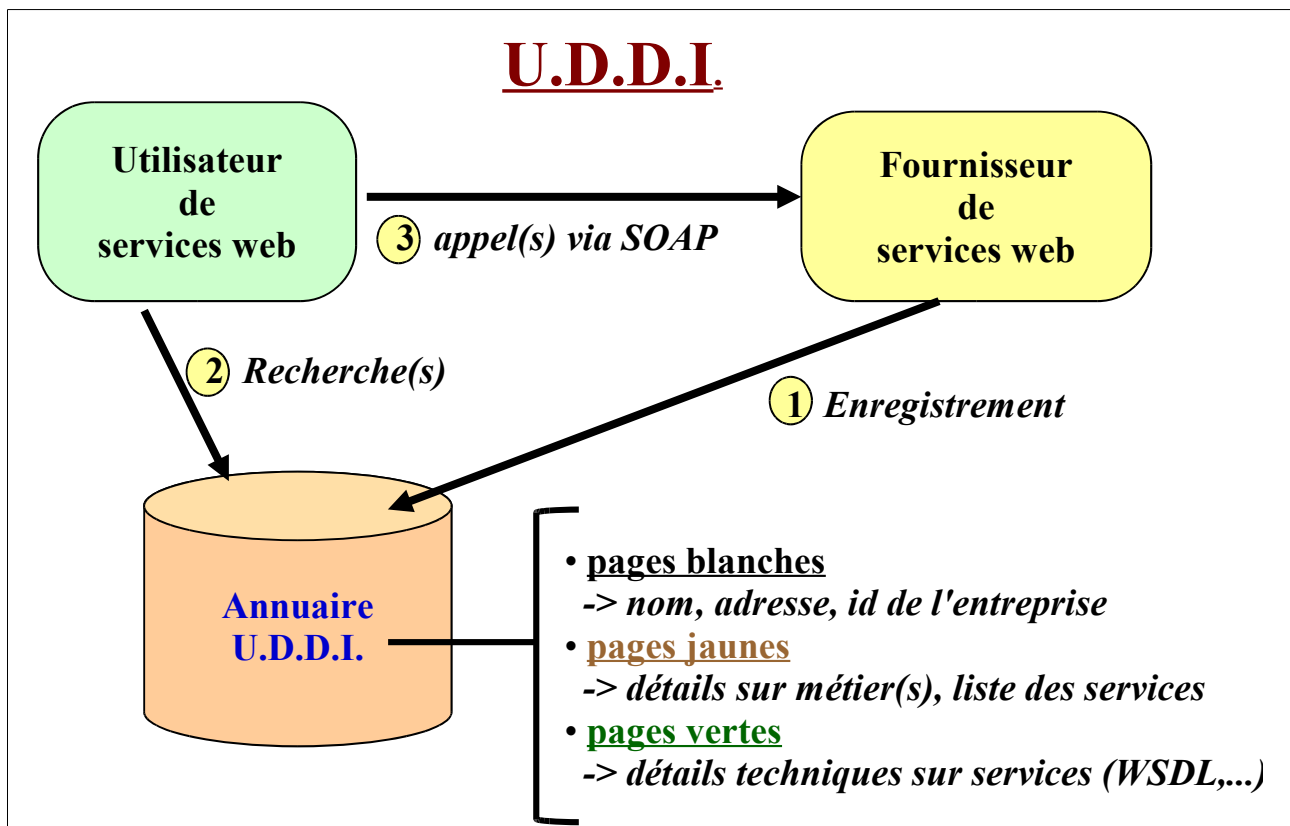
La norme **UDDI** vise à structurer la structure des annuaires de services et définit des services techniques pour la publication et l'interrogation.

Cependant, cette norme (déjà assez ancienne) est très complexe et n'est souvent utilisée qu'au sein de grandes entreprises multi-nationales (là où il faut tenir compte de plusieurs langues humaines au niveau des critères de recherche) .

Une PME aura tout intérêt à utiliser un annuaire plus simple que UDDI.

7. Annuaire UDDI (présentation)

UDDI signifie *Universal Description, Discovery & Integration*



Universal Description, Discovery & Integration (<http://www.uddi.org>)

UDDI est un **service d'annuaire basé sur Xml** essentiellement prévu pour **référencer des services "web"**. Le noyau du projet UDDI est l'**UDDI Business Registry**, annuaire contenant des informations de trois types, décrites au format XML :

- **Pages blanches** : noms, adresses, contacts, identifiants,... des entreprises enregistrées. Ces informations sont décrites dans des entités de type **Business Entity**. Cette description inclut des informations de catégorisation permettant de faire des recherches spécifiques dépendant du métier de l'entreprise ;
- **Pages jaunes** : *détails sur le métier de l'entreprise, les services qu'elle propose*. Ces informations sont décrites dans des entités de type **Business Service** ;
- **Pages vertes** : *informations techniques sur les services proposés*. Les pages vertes incluent des références vers les spécifications des services Web, et les détails nécessaires à l'utilisation de ces services : interfaces implémentées, information sur les contacts pour un processus particulier, description du processus en plusieurs langages, catégorisation des processus, pointeurs vers les spécifications décrivant chaque API. Ces informations sont décrites dans deux documents : un *Binding Template*, et un *Technology Model (tModel)*.

8. Annuaire publics et annuaire privés

8.1. Principaux annuaires publics

NB: La plupart des annuaires UDDI publics (mise en place par IBM et Microsoft) ne sont plus maintenus.

Raisons prétextées:

- Technologie au point , tests terminés
- Ce n'est pas aux éditeurs de logiciels (IBM ou Microsoft , ...) de maintenir des annuaires publics (ce n'est pas leurs métiers)

Conséquences:

La technologie UDDI à un niveau privé perd un peu de son intérêt mais il faut accepter cet état de fait (tant qu'un organisme international indépendant ne prendra pas à sa charge un annuaire UDDI véritablement public).

8.2. Quelques implémentations d'annuaires UDDI

IBM/WebSphere/AppServer/InstallableApps/**uddi.ear** à installer dans **WebSphere 6**

JUDDI (<http://ws.apache.org/juddi>) ==> **juddi.ear** (ou **juddi.war**) = produit open source à installer dans n'importe quel conteneur Web récent (ex: Tomcat 5) ou serveur J2EE (ex: JBoss) .

NB: L'application J2EE correspondant à un annuaire UDDI nécessite généralement la mise en place d'une base de données relationnelle ainsi qu'un pool de connexions JDBC. ==> Lire la documentation de JUDDI ou autre pour connaître la procédure d'installation.

9. taxonomies (au sein des annuaires)

Il existe plusieurs sortes de taxonomies (UNSPSC ,) et de nouvelles peuvent éventuellement apparaître. Une entrée de taxonomie comporte les 3 éléments suivants:

- l'**identifiant (au format UUID) de la taxonomie** (lié à un organisme et une version)
- le **nom** de l'entrée = *valeur en clair* (ex: "France" , "Accounting Software" ,)
- la **valeur** de l'entrée = *valeur codée* (ex: "Fr" , "43.23.16.01")

Le nom est pratique pour un affichage et une compréhension humaine.

La valeur est pratique et bien souvent nécessaire pour effectuer une recherche précise.

NB: Certaines taxonomies ont un attribut "Checked" à "true". Ceci signifie que la valeur sera vérifiée .

9.1. UNSPSC

The **United Nations Standard Products and Services Code** is a hierarchical convention that is

used to classify all products and services.

Exemple :

Hierarchy Category Number and NameSegment:

43 *Information Technology Broadcasting and Telecommunications Communications Devices and Accessories*

Family:

20 *Components for information technology or broadcasting or telecommunications Computer Equipment and Accessories*

Class:

15 *Computers Computer accessories*

Commodity:

01 *Computer switch boxes Docking stations*

Business Function:

14 *Retail*

===== > **43.20.15.01.14**

Autre exemple: 43.23.16.01 , Accounting Software

9.2. ISO 3166 Geographic Taxonomy

<http://www.uddi.org/taxonomies/iso3166-1999-utf8.txt>

exemple : valeur (codée) , nom (en clair)

ex1 (région): **FR-J** , Ile-De-France

ex2 (département): **FR-75** , Paris

ex3 (pays): **FR** , France

10. Web Services "R.E.S.T."

REST = style d'architecture (conventions)

REST est l'acronyme de **R**epresentational **S**tate **T**ransfert.

C'est un **style d'architecture** qui a été décrit par *Roy Thomas Fielding* dans sa thèse «*Architectural Styles and the Design of Network-based Software Architectures*».

L'information de base, dans une architecture REST, est appelée **ressource**. Toute information (à sémantique stable) qui peut être nommée est une ressource: un article, une photo, une personne, un service ou n'importe quel concept.

Une ressource est identifiée par un **identificateur de ressource**. Sur le web ces identificateurs sont les **URI** (Uniform Resource Identifier).

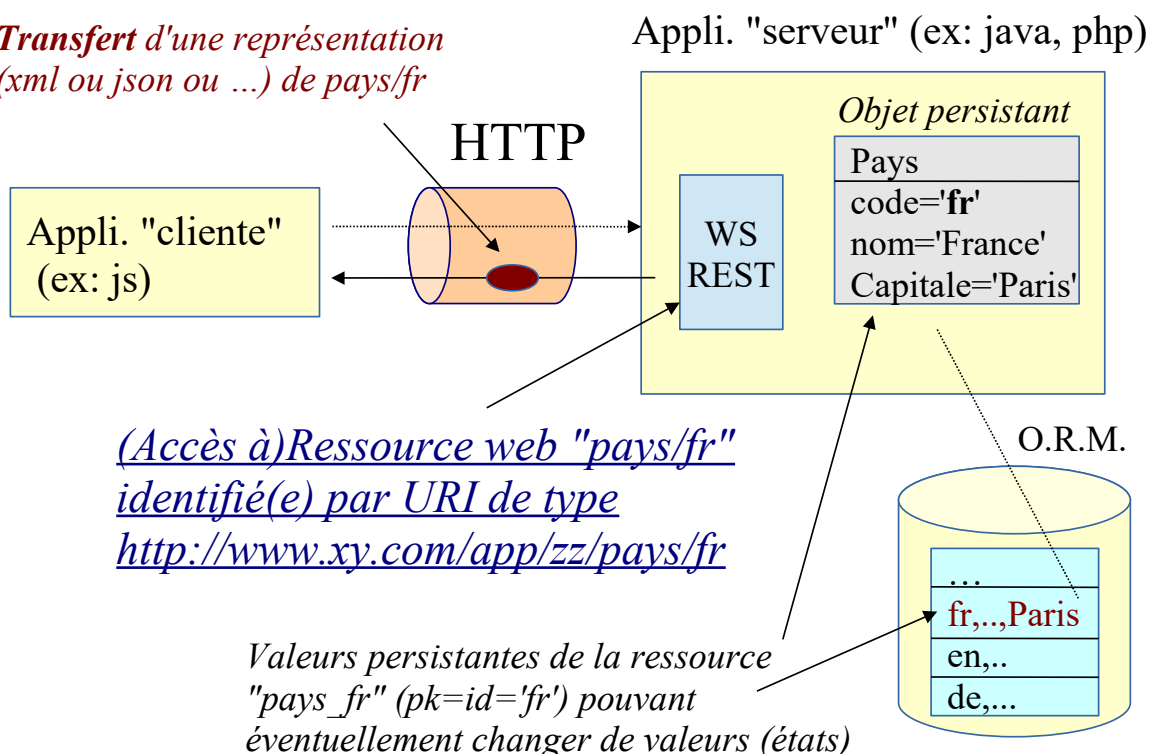
NB: dans la plupart des cas, une ressource REST correspond indirectement à un enregistrement en base (avec la *clef primaire* comme partie finale de l'uri "identifiant").

Les composants de l'architecture REST manipulent ces ressources en **transférant à travers le réseau** (via HTTP) des **représentations de ces ressources**.

Sur le web, on trouve aujourd'hui le plus souvent des représentations au format **HTML, XML ou JSON**.

REST : transferts de représentations de ressources

*Transfert d'une représentation
(xml ou json ou ...) de pays/fr*



REST et principaux formats (xml,json)

Une invocation d'URL de service REST peut être accompagnée de données (en entrée ou en sortie) pouvant prendre des formats quelconques :

text/plain , text/html , application/xml , application/json , ...

Dans le cas d'une lecture/recherche d'informations , le format du résultat retourné pourra (selon les cas) être :

- **imposé (en dur) par le code du service REST .**
- **au choix (xml , json) et précisé par une partie de l'url**
- **au choix (xml , json) et précisé par le champ "Accept :" de l'entête HTTP de la requête. (exemple: Accept: application/json) .**

Dans tous les cas, la réponse HTTP devra avoir son format précisé via le champ habituel **Content-Type: application/json** de l'entête.

Format JSON (JSON = *JavaScript Object Notation*)

Les 2 principales caractéristiques de JSON sont :

- Le principe de clé / valeur (map)
- L'organisation des données sous forme de tableau

Les types de données valables sont :

- tableau
- objet
- chaîne de caractères
- valeur numérique (entier, double)
- booléen (true/false)
- null

```
[
  {
    "nom": "article a",
    "prix": 3.05,
    "disponible": false,
    "descriptif": "article1"
  },
  {
    "nom": "article b",
    "prix": 13.05,
    "disponible": true,
    "descriptif": null
  }
]
```

une liste d'articles

une personne

```
{
  "nom": "xxxx",
  "prenom": "yyyy",
  "age": 25
}
```

REST et méthodes HTTP (verbes)

Les **méthodes HTTP** sont utilisées pour indiquer la **sémantique des actions demandées** :

- **GET** : **lecture/recherche** d'information
- **POST** : **envoi** d'information
- **PUT** : **mise à jour** d'information
- **DELETE** : **suppression** d'information

Par exemple, pour récupérer la liste des adhérents d'un club, on peut effectuer une requête de type **GET** vers la ressource <http://monsite.com/adherents>

Pour obtenir que les adhérents ayant plus de 20 ans, la requête devient <http://monsite.com/adherents?ageMinimum=20>

Pour supprimer numéro 4, on peut employer une requête de type **DELETE** telle que <http://monsite.com/adherents/4>

Pour envoyer des informations, on utilise **POST** ou **PUT** en passant les informations dans le corps (invisible) du message HTTP avec comme URL celle de la ressource web que l'on veut créer ou mettre à jour.

Exemple concret de service REST : "Elevation API"

L'entreprise "**Google**" fournit gratuitement certains services WEB de type REST. "**Elevation API**" est un service REST de Google qui renvoie l'altitude d'un point de la planète selon ses coordonnées (latitude ,longitude) .

La documentation complète se trouve au bout de l'URL suivante :

<https://developers.google.com/maps/documentation/elevation/?hl=fr>

Sachant que les coordonnées du Mont blanc sont :

Lat/Lon : 45.8325 N / 6.86417 E (GPS : 32T 334120 5077656)

Les invocations suivantes (du service web rest "api/elevation")

<http://maps.googleapis.com/maps/api/elevation/json?locations=45.8325,6.86417>

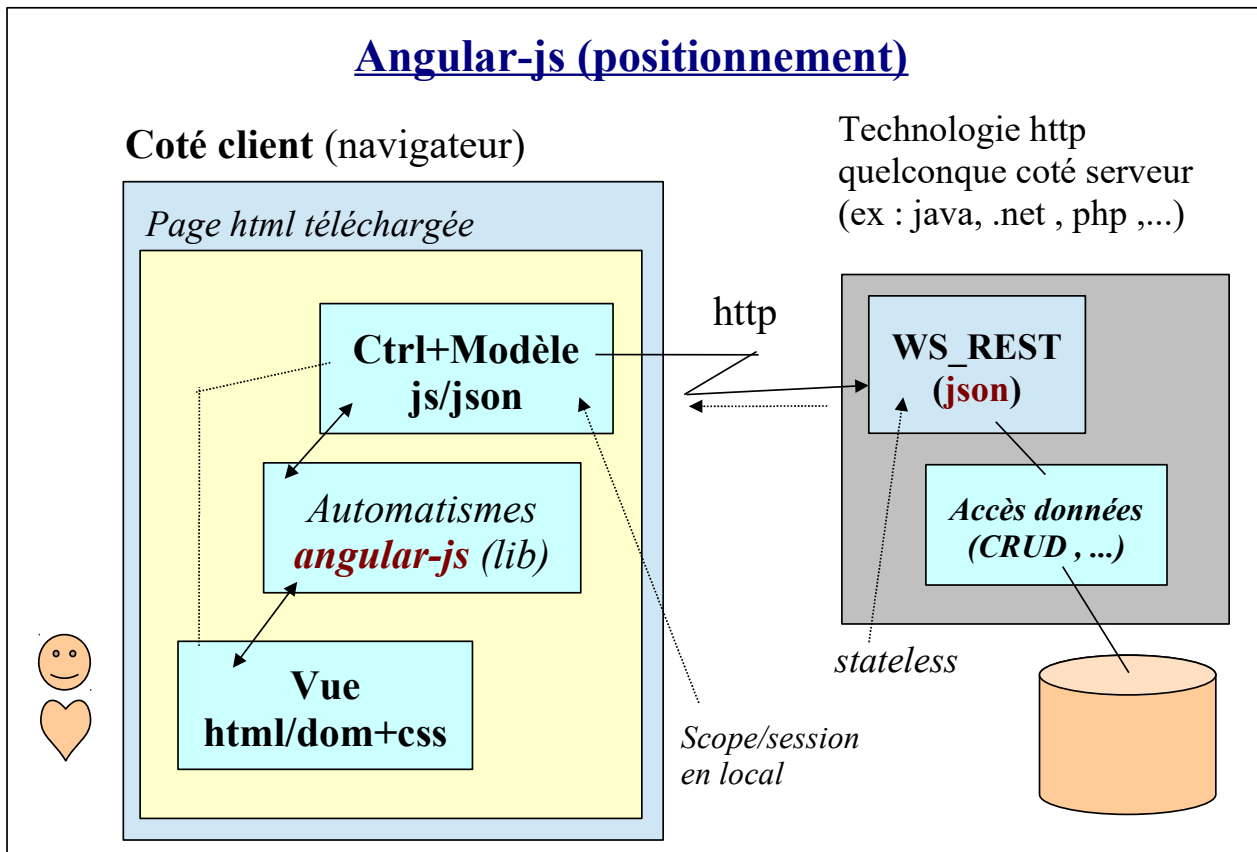
<http://maps.googleapis.com/maps/api/elevation/xml?locations=45.8325,6.86417>

donne les résultats suivants "json" ou "xml":

```
{ "results" : [
  {
    "elevation" : 4766.466796875,
    "location" : {
      "lat" : 45.8325,
      "lng" : 6.86417
    },
    "resolution" : 152.7032318115234
  }
], "status" : "OK"
}
```

```
?xml version="1.0" encoding="UTF-8"?>
<ElevationResponse>
  <status>OK</status>
  <result>
    <location>
      <lat>45.8325000</lat>
      <lng>6.8641700</lng>
    </location>
    <elevation>4766.4667969</elevation>
    <resolution>152.7032318</resolution>
  </result>
</ElevationResponse>
```

Angular-js (positionnement)



Conventions "angular-js" sur URL des ressources REST

Type requêtes	HTTP Method	URL ressource(s) distante(s)	Request body	Réponse JSON
Recherche multiple	GET	.../products .../products?crit1=v1&crit2=v2	vide	Liste/tableau d'objets
Recherche par id	GET	.../products/idRes (avec idRes=1,...)	vide	Objet JSON
Ajout (seul)	POST		Objet JSON	Objet JSON avec id quelquefois calculé (incr)
Mise à jour (seule)	PUT	.../products/idRes	Objet JSON	Statut ou ...
SaveOr Update	POST	.../products/idRes	Objet JSON	Objet JSON modifié (id)
suppression	DELETE	.../products/idRes	vide	Statut ou ...
Autres/product-action/opXy/...

11. Gestion de la sécurité / Services Web

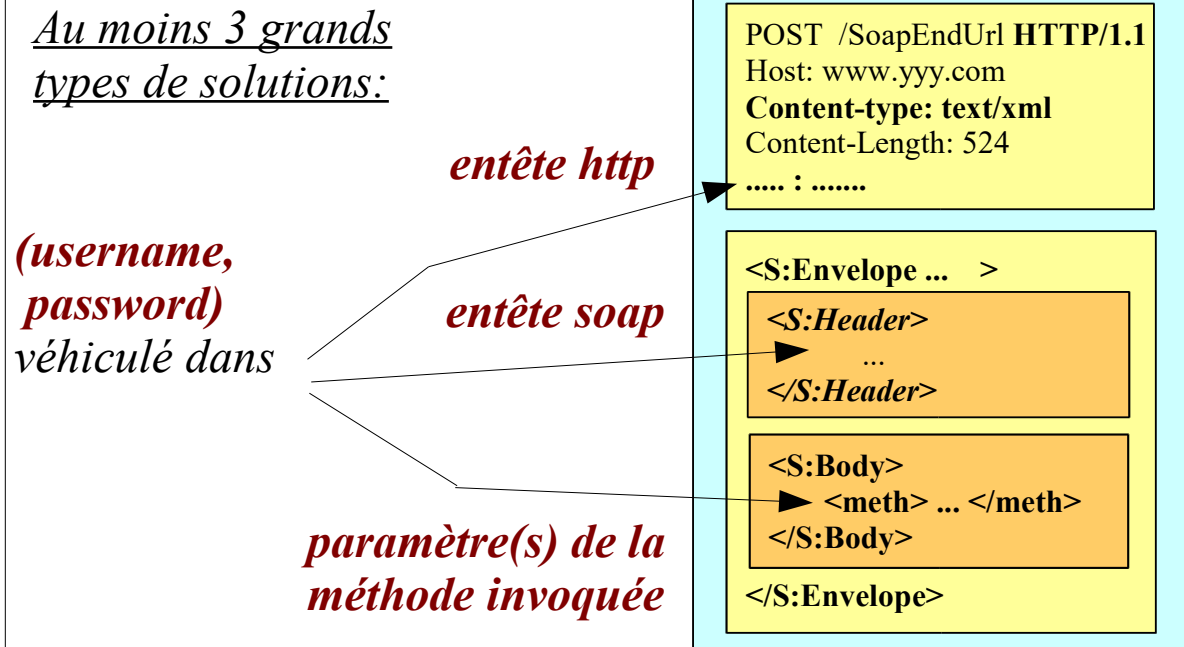
11.1. Éléments de sécurité nécessaires sur les services WEB

Sécurité liée aux services "WEB"

- ***Authentification du client*** via (*username,password*) ou autre pour contrôler les accès aux services
--> "*infos auth*" dans ***entête Http*** ou ***entête SOAP*** ou
- ***Cryptage / confidentialité***
--> via ***HTTPS/SSL*** et/ou cryptage XML/SOAP
- ***Signature électronique des messages*** (certificats)
- ***Intégrité des messages (vérif. non interception/modification)***
--> via *signature d'une empreinte d'un message*
- ...

11.2. différents types d'authentications (WS)

Authentification du client invoquant un service WEB



11.3. authentification élémentaire / fonctionnelle via "jeton"

La technique élémentaire d'authentification associée aux services WEB correspond à celle qui est utilisée au niveau de UDDI à savoir:

- 1) appel d'une méthode de type
 String sessToken = **getSessionToken**(String userName,String password)
 qui renvoie un jeton de sécurité lié à une session après avoir vérifié la validité du couple (username,password) d'une façon ou d'une autre (ex: via ldap).
- 2) passage du jeton de sécurité "sessToken" en tant qu'argument supplémentaire de toutes les autres méthodes du service Web : **methodeXy**(String sessToken, String param2,...) ;
 Une simple vérification de la validité du jeton servira à décider si le service accepte ou pas de répondre à la requête formulée.

Variantes classiques:

- repasser le jeton de sécurité/session au travers d'un argument plus large généralement appelé "contexte" (au sens "contexte technique" et non métier). Ce contexte pourra ainsi véhiculer d'autres informations techniques jugées pertinentes (ex: transactions , référence xy, ...).
- utiliser en plus **SOAP over HTTPS** pour crypter si besoin certaines informations échangées (ex: [username, password] ou plus).

---> gros inconvénient de la méthode "élémentaire/fonctionnelle" : chaque méthode appelée et

sécurisée comporte au moins un paramètre supplémentaire .

Autrement dit , la sécurité se voit dans les signatures des méthodes à invoquer.

11.4. Authentification véhiculée via l'entête (header) "HTTP"

En passant les informations d'authentification (username,password) ou ... dans l'entête HTTP on bénéficie des avantages suivants:

- authentification bien séparée des aspects fonctionnels
- réutilisation d'une fonctionnalité standard d'HTTP "Authentification basique HTTP" (username/password) véhiculés de façon un peu crypté dans un champ de l'entête HTTP.
- paramétrage/codage assez simple à faire via des intercepteurs ou API des technologies "Web Services" (*ex: intercepteur "cxf" , interface "BindingProvider" de JAX-WS*)

Pour encore plus de sécurité , on peut utiliser conjointement SSL/HTTPS .

Cette méthode est très bien dans la plupart des cas "ordinaires" (non pointus).

Limitation:

Dans certains cas pointus , l'enveloppe SOAP est relayée par différents intermédiaires (ex: ESB ,) . Certains maillons du transport peuvent être effectués par d'autres protocoles que HTTP et les informations de l'entête HTTP risquent alors de ne pas bien être retransmises jusqu'à la destination prévue.

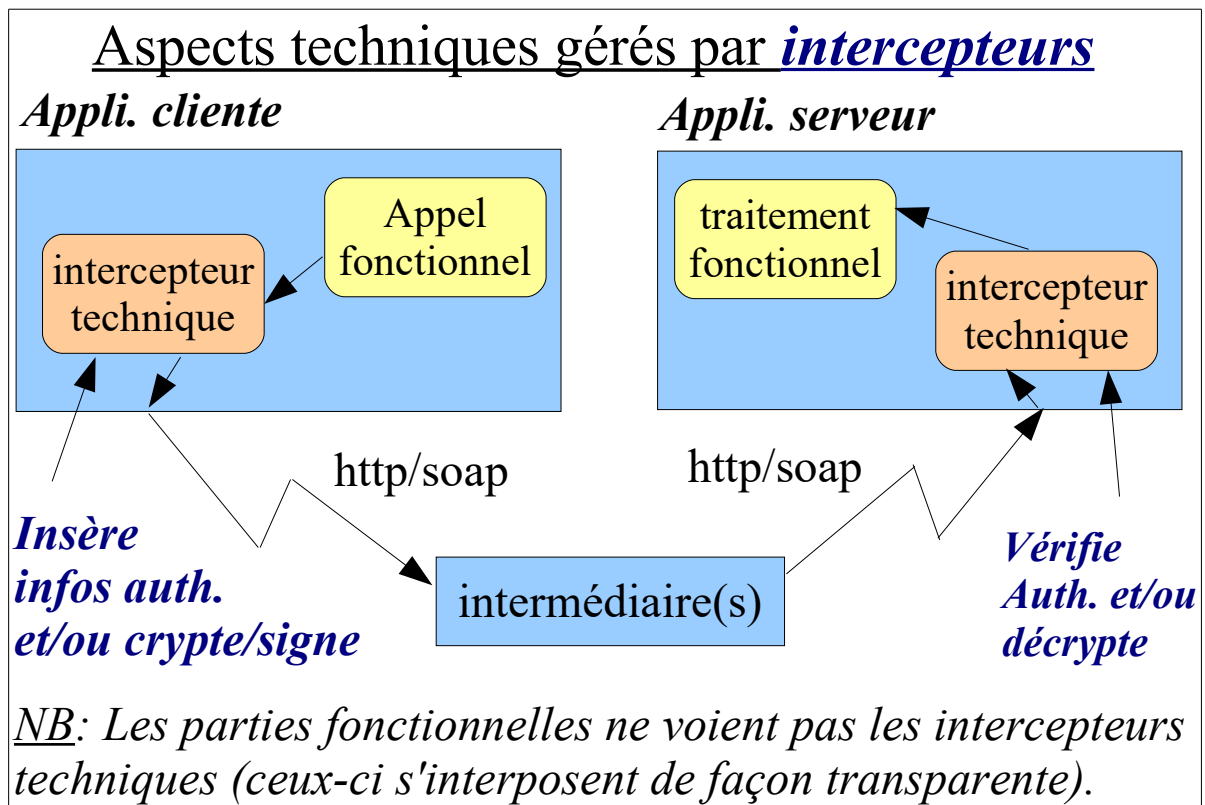
11.5. Authentification véhiculée via l'entête (header) "soap"

Avantage: fonctionne même si plusieurs protocoles de transport (HTTP puis JMS puis ...) plein d'options permettant une authentification forte (multicritères , très sécurisée)

Inconvénients :

- API et/ou intercepteur(s) un peu plus complexe(s)
- Les technologies "client" et "serveur" doivent toutes les deux être compatibles sur la gestion des entêtes SOAP .

11.6. Aspects techniques gérés par intercepteurs



11.7. Problématique (sécurité avancée)

Bien que permettant globalement une interaction entre 2 parties (client et serveur), la technologie des services WEB (XML sur HTTP) fait intervenir de multiples intermédiaires à travers le réseau internet.

Le contexte de sécurité SOAP (idéalement géré de bout en bout mais pas toujours pour des raisons de coûts) doit être suffisamment perfectionné pour supporter des éventuelles attaques ou menaces (interception des messages, altération de ceux-ci, ...).

D'autre part, tout service n'est pas forcément gratuit et un accord préalable (abonnement, ...) peut déboucher sur la génération d'un compte utilisateur (username, password) permettant d'accorder finement des droits d'accès au service Web.

Recommandation du ws-i en terme de sécurité:

==> utiliser **HTTPS** (avec SSL 3.0 ou TLS 1.0) et non pas HTTP pour véhiculer l'enveloppe SOAP. Cette recommandation est toutefois nuancée et insiste sur le compromis **sécurité/sur-coûts** qui doit être évalué au cas par cas.

Dans certains cas SSL (prévu pour un mode point à point) ne suffit pas.

Il faut alors utiliser quelques unes des technologies présentées ci-après.

11.8. Web Service Security (WS-S)

Développé par l'organisme OASIS, **WS-S** est une **extension de SOAP** permettant de contrôler de bout en bout les éléments suivants sur certains messages:

- intégrité (via *Xml-Signature* et *Security token* (ex: *certificat X-509* ou *ticket kerberos* encapsulés dans des éléments XML))
- confidentialité (via *Xml-Encryption* + *Security token*)
- authentication

NB:

WS-S indique comment représenter en XML des jetons de sécurité mais n'impose pas un type précis de certificat .

Ces jetons (certificats) servent essentiellement à s'assurer que les messages SOAP n'ont pas été interceptés ni modifiés.

Un peu à la façon de SSL , WS-S agit de façon transparente en ajoutant automatiquement des éléments de sécurité sur les messages "SOAP" .

Exemple (partiel):

```
<SOAP:Envelope xmlns:SOAP='http://schemas.xmlsoap.org/soap/envelope/'>
  <SOAP:Header>
    <wsse:Security
      xmlns:wsse='http://schemas.xmlsoap.org/ws/2002/07/secext'>
      <Signature xmlns='http://www.w3.org/2000/09/xmldsig#'>
        ...defined below...
      </Signature>
    </wsse:Security>
  </SOAP:Header>
  <SOAP:Body id='Body'>
    ...message body...
  </SOAP:Body>
</SOAP:Envelope>
```

11.9. Aspects avancés liés à la sécurité

==> Approfondir si besoin WS-Security (Xml-Signature, Xml-encryption, ...) pour les cas "pointus" .

Les API sophistiquées (java , .net) telles que CXF prennent généralement en sorte WS-Security.

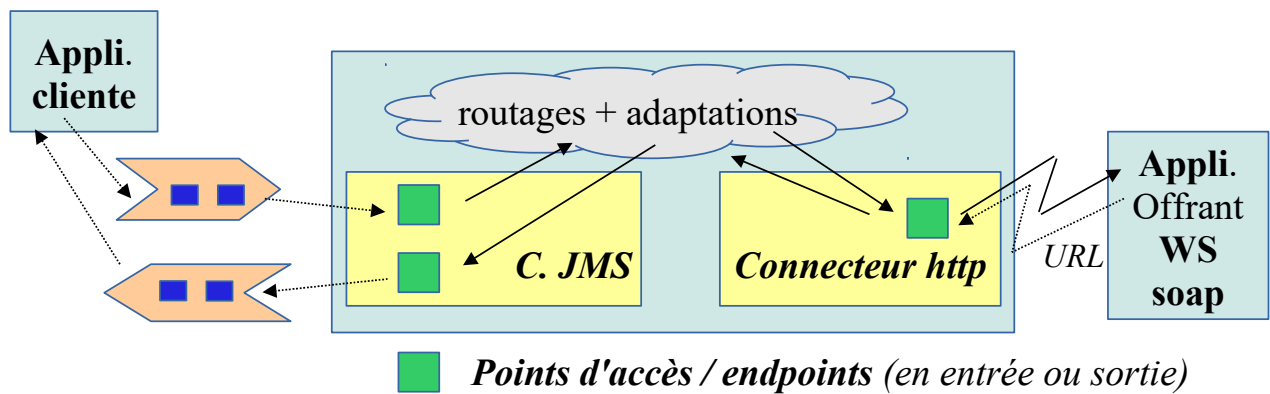
Il existe également un lien avec **Security assertion markup language (SAML)** (sorte de "SSO / Single Sign On" en version Web) .

III - EAI et ESB

1. EAI et ESB

E.S.B. = Enterprise Service Bus :

- Un **ESB** est essentiellement à voir comme un serveur d'accès à des services distants .
- Une application cliente peut se connecter à l'ESB selon un **grand choix de protocoles** (avec **points d'accès** gérés par connecteurs).
- Les services fonctionnels sont souvent rendus par d'autres applications en arrière plan et combinés ou enrichis par l'ESB.



Principales fonctionnalités d'un ESB

- **Routage de messages** (*selon URL, éventuellement conditionné*)

Changement de protocoles

(*http:// , file:// , jms , smtp , rmi , ...*)

- **Transformation de format de message**
(*xml , json , ...*)

- **Adaptations fonctionnelles** (*traductions, mapping entre Api , xslt , ...*)

- **Enrichissements techniques**
(*logs , sécurité , monitoring , ...*)

- **Combinaison/orchestration de services**

- **Éventuels services internes** (*java , ...*)

- **Supervision** (*PKI , SAM , BAM*)

essentiel

*éventuelle
intégration
de bpm ou bpmn2*

*potentiellement
serveur d'applications
(comme jee)*

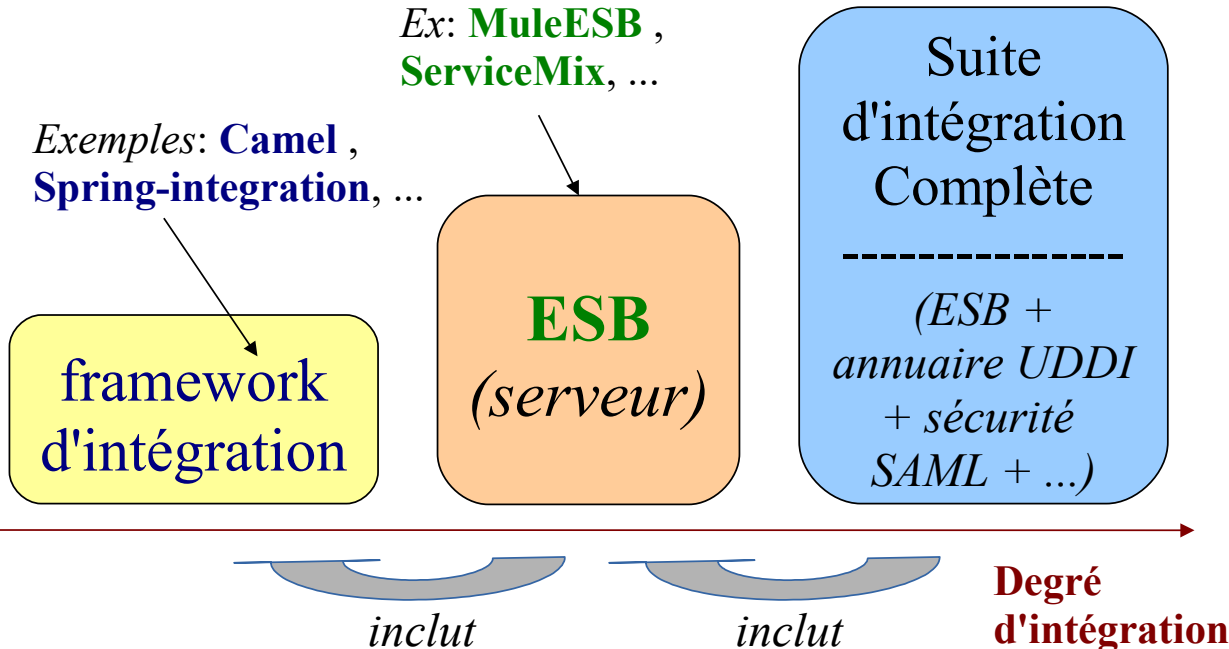
ESB et framework d'intégration

Taille / complexité / prix

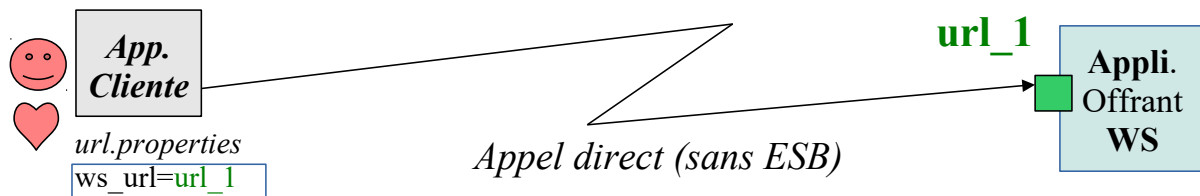
Ex: suite *WebMethod*, ...

Exemples: **Camel**,
Spring-integration, ...

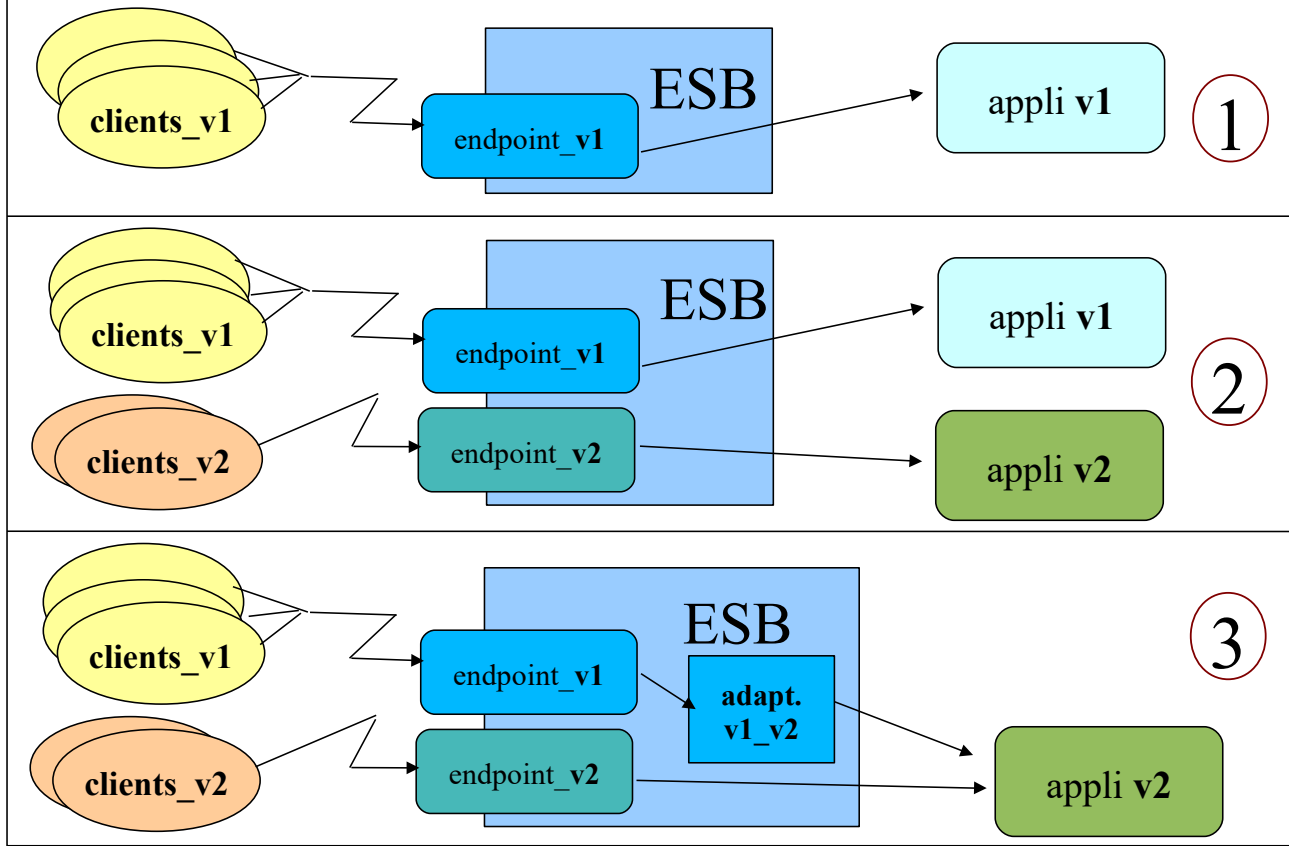
Ex: **MuleESB**,
ServiceMix, ...



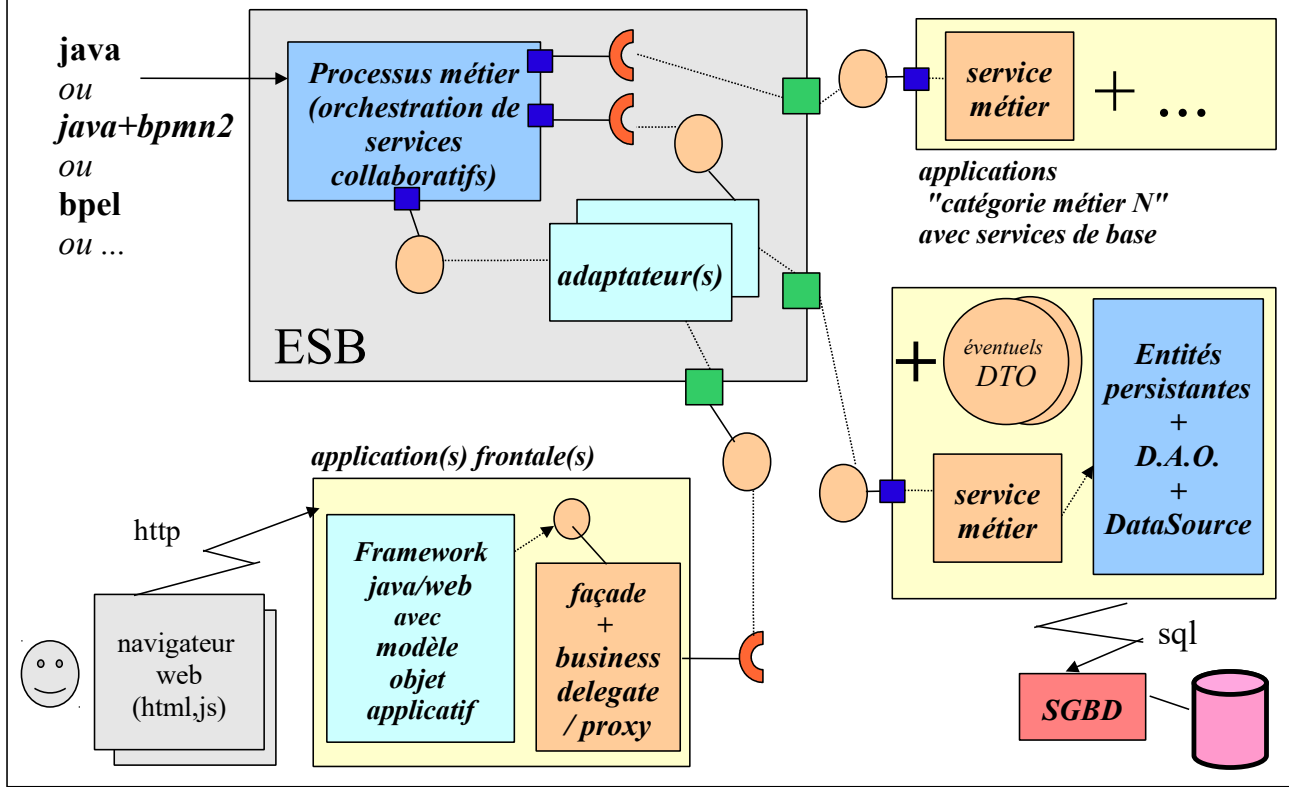
Trois variantes fonctionnellement identiques :



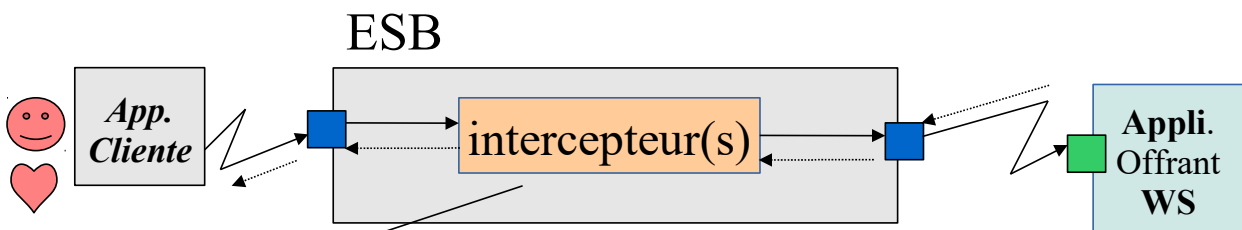
Exemple d'évolution de versions en douceur



ESB hébergeant éventuellement des services d'orchestration au sein d'une architecture SOA :



Intercepteurs, PKI, SAM et BAM



Production potentielle de **mesures/statistiques** *selon paramétrages*
(KPI= Key Performance Indicator)

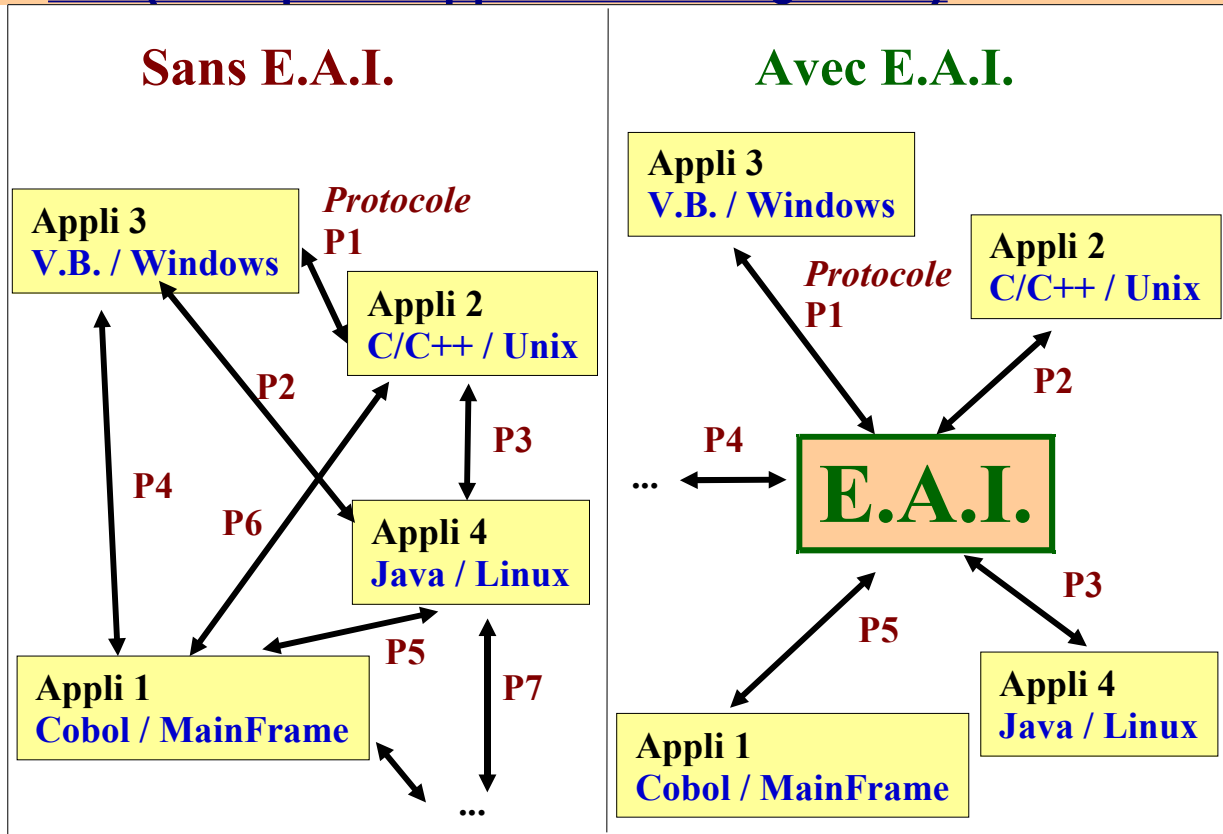
SAM = **S**ystem **A**ctivity **M**onitoring
(supervision technique/opérationnelle)

BAM = **B**usiness **A**ctivity **M**onitoring
(supervision fonctionnelle/métier)

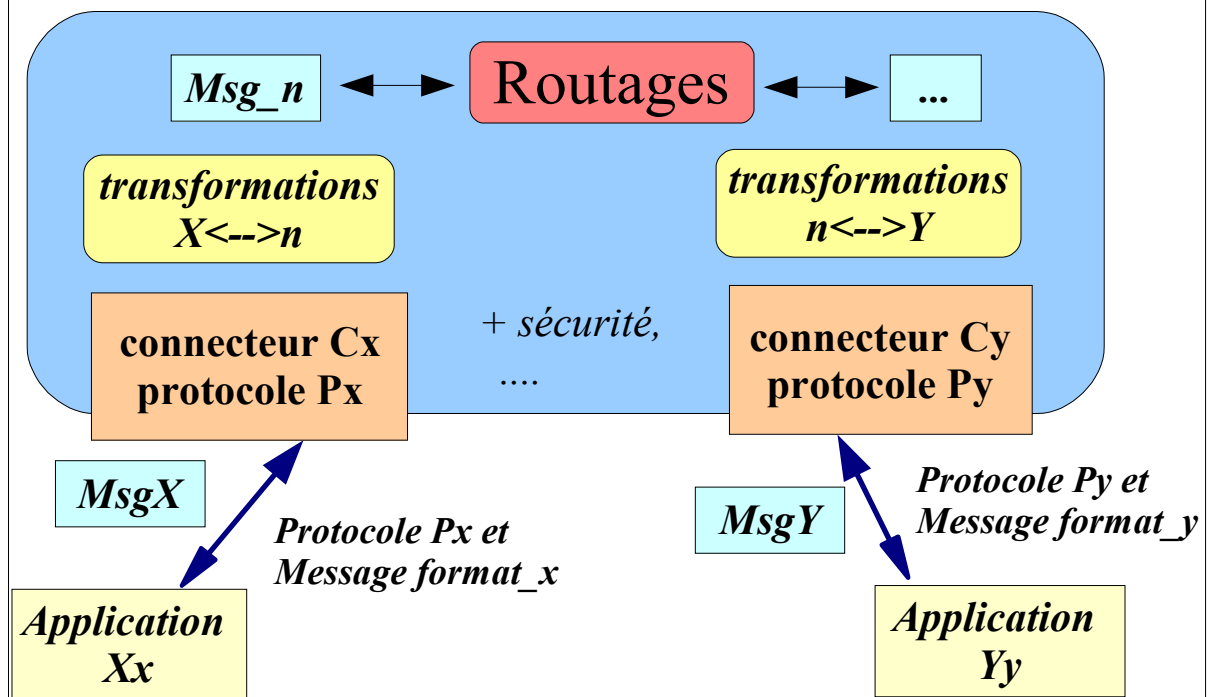
À bien doser !

(trop de mesures/stats peuvent sensiblement ralentir les flux dans l'ESB) .

2. EAI (Enterprise Application Integration)



Fonctionnalités d'un EAI:

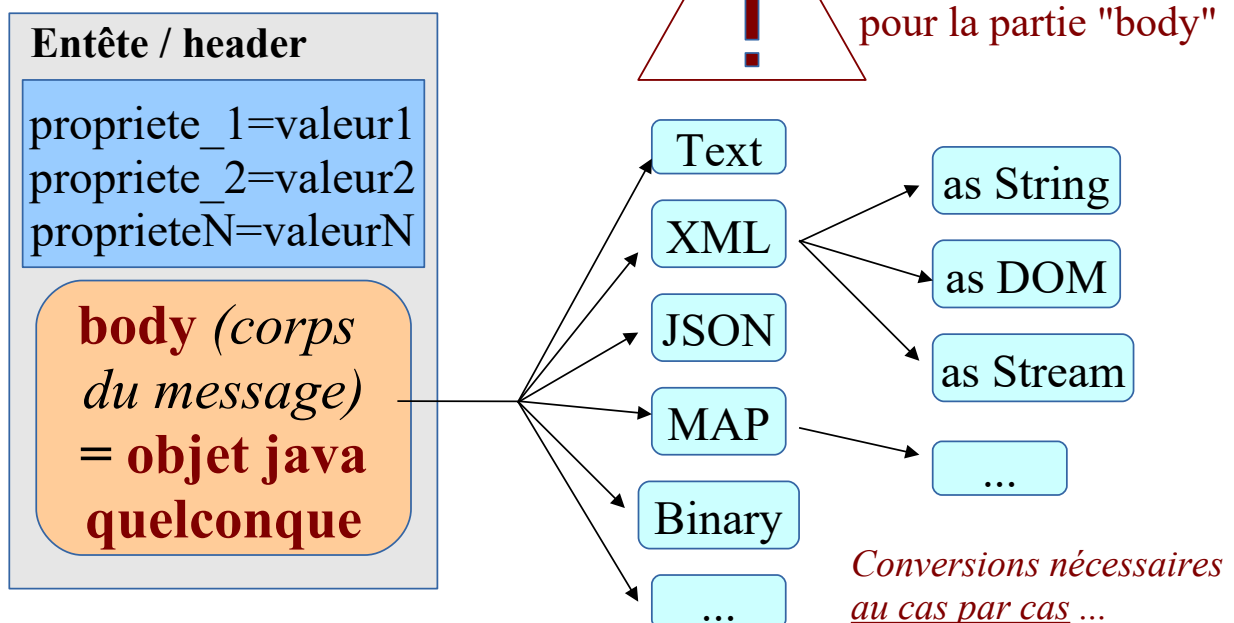


Transformations techniques internes à l'EAI/ESB

- * Si au cas par cas : potentiellement **$O(n^2)$**
transformations directes
[complexe , performant]
- * Si format interne "*neutre*" ou "*normalisé*"
 $(x \leftrightarrow n \leftrightarrow y)$
 $\rightarrow 2*n$ doubles transformations
[simple , moins performant]

Cas fréquent des ESB basés sur java :

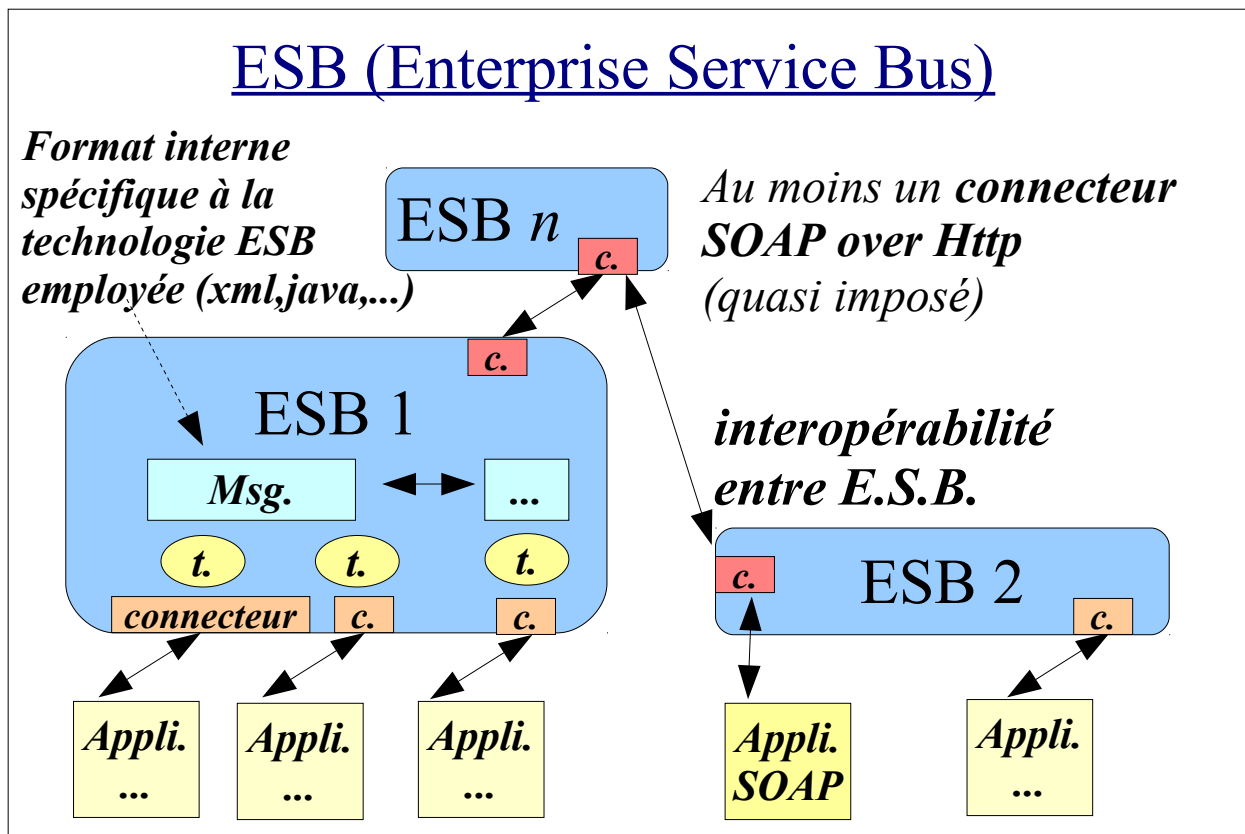
Structure d'un **message** interne géré par l'ESB :



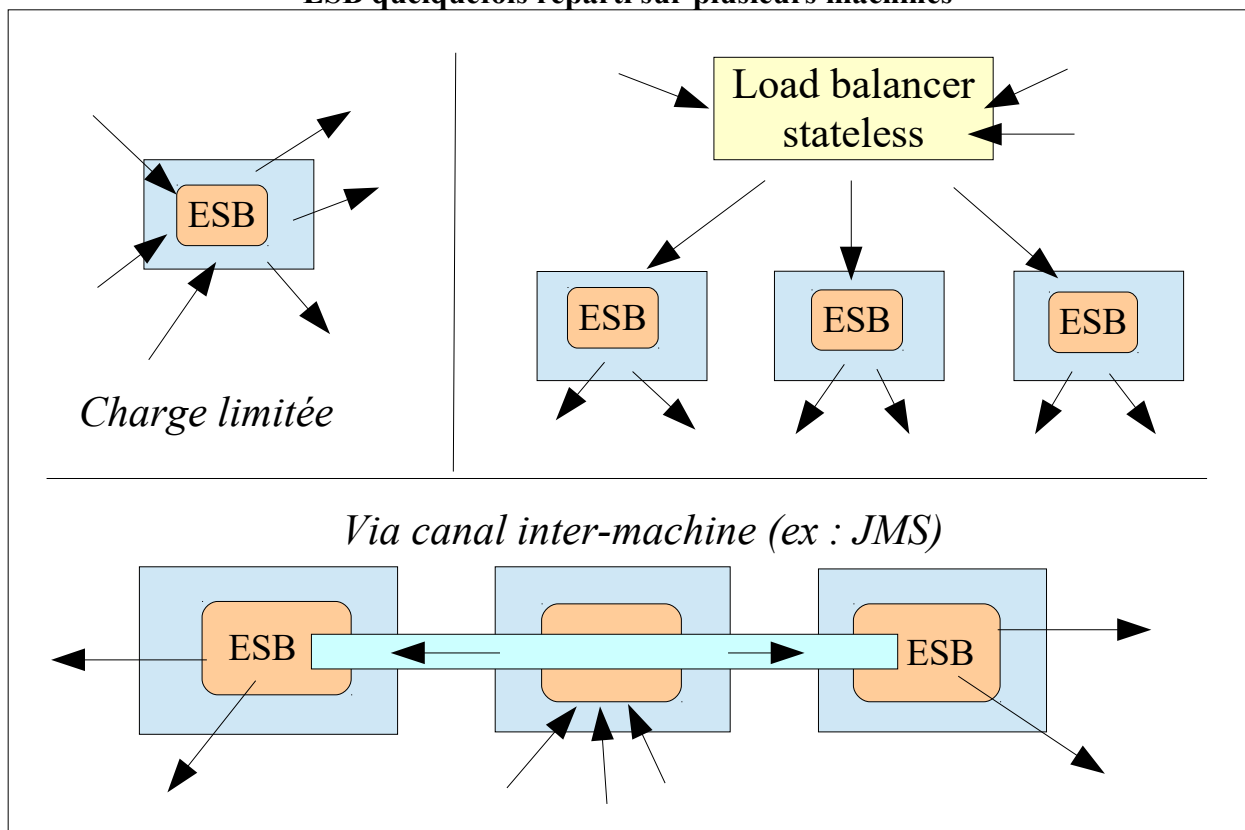
Grands traits des premiers EAI:

- Xml quelquefois utilisé pour le format des messages intermédiaires (re-transformations pratiques).
- EAI = concept
==> différentes implémentations
assez propriétaires (ex: **Tibco** , **WebMethod**, ...) et pas directement interopérables.
- Attention aux performances (beaucoup de trafic de messages + traitements CPU liés aux transformations , ...).

3. ESB (Enterprise Service Bus)



ESB quelquefois réparti sur plusieurs machines



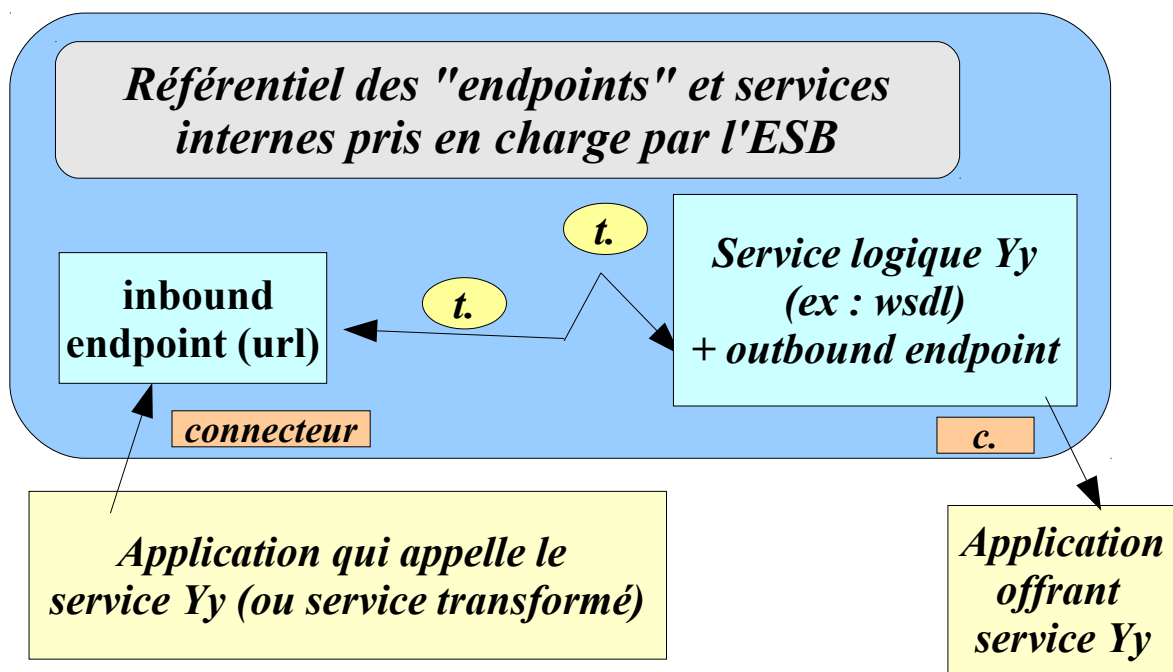
Configuration requise au sein d'un ESB

Un ESB offre généralement une infrastructure à base de connecteurs et transformateurs paramétrables .

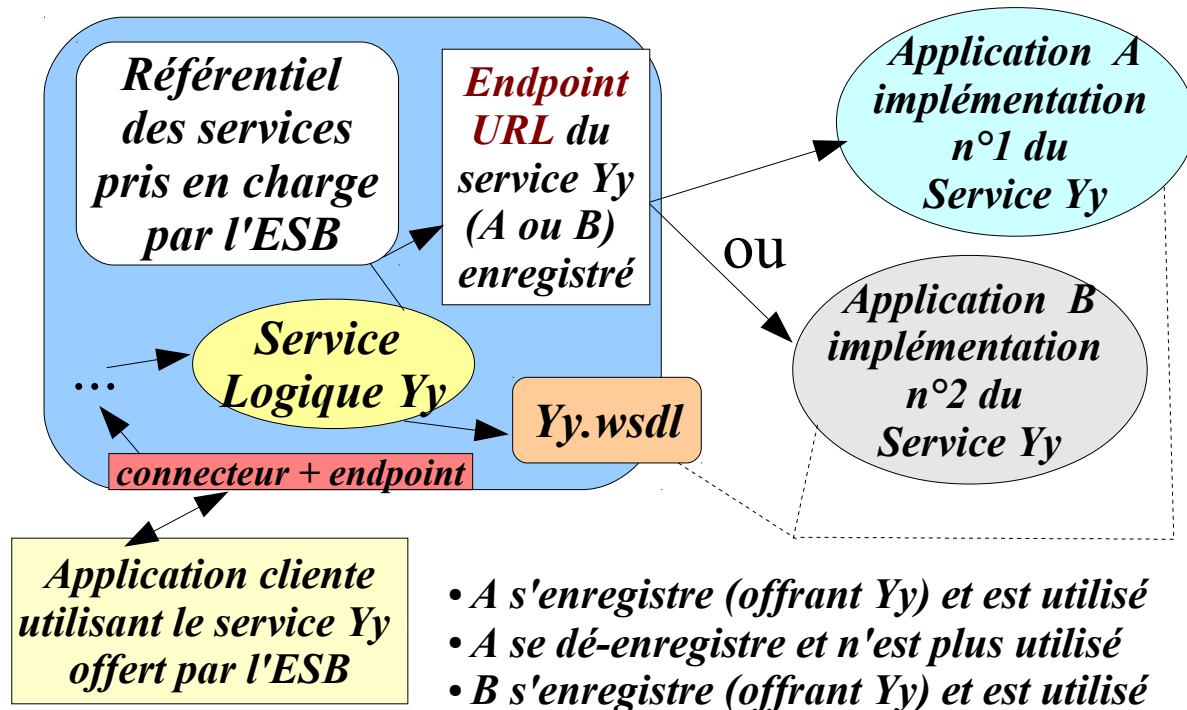
On a généralement besoin de configurer :

- * des *points d'entrées et de sorties (endpoints)* avec des *URLs* et protocoles particuliers.
- * des *transformations* et *routages internes* au cas par cas (selon les besoins)
- * des *définitions logiques des services rendus ou accessibles* (ex : fichier **wsdl** , code java annoté , ...)

ESB: paramétrage & intégration



ESB: basculement transparent d'implémentation



Valeurs ajoutées couramment configurées au sein d'un ESB

Médiation de service :

Service intermédiaire pouvant ajouter des contrôles de sécurité, des mesures statistiques, des ajustements de formats, des relocalisations transparentes, ...

Orchestrations/com combinaisons diverses:

Service interne à l'ESB qui configure plusieurs appels vers d'autres services (internes ou externes) et qui agrège/combine quelquefois les résultats.
(ex : composition , diffusion, comparaisons/filtrages, duplication/re-transfert, ...)

ESB: éléments libres & imposés (propriétaires & standards)*Tout ESB se doit de:*

- * Offrir un accès interne dans un format local unifié (java, xml/soap/wsdl , ou ...) aux services (internes et externes) enregistrés.
- * Permettre à une application externe de se connecter (au moins via (SOAP, WSDL)) à un service pris en charge par l'ESB (indirectement via un connecteur)

Chaque ESB est libre de:

- * se configurer à sa façon (scripts, console , fichiers de configuration, ...).
- * de gérer à sa guise les enregistrements / déploiements de services.

3.1. Principaux ESB

Liste (non exhaustive) de quelques ESB :

ESB des grandes marques (avec prix quelquefois élevé mais un support généralement sérieux) :

<i>ESB</i>	<i>Editeur</i>	<i>Caractéristiques</i>
Tibco ESB	Tibco	Un des pionniers (anciennement EAI)
WebMethod	Software AG	Autre pionnier (anciennement EAI). Les versions récentes ont été beaucoup améliorées.
OSB (<i>Oracle Service Bus</i>)	Oracle	ESB très complet (<i>attention : pas mal de différences entre certaines versions</i>). <i>Architecture très propriétaire.</i>
WebSphere ESB	IBM	ESB très complet (utilisant norme SCA)
BizTalk	Microsoft	Lien avec norme WCF de .net
Talend ESB	Talend (éditeur ETL)	Suite SOA complète (en interne basé sur composants "open-source")

ESB "Open source" ou mixte (version "community" et version "payante") :

<i>ESB</i>	<i>Editeur</i>	<i>Caractéristiques</i>
<i>OpenESB "has been" ServiceMix [3.x , 4.3] Petals</i>	SUN, Apache, OW2 (éditeur de Jonas)	JB1 et "has been"
ServiceMix >= 4.4	Apache	OSGi (quasiment plus JB1) , plus d'intégration possible de ODE/BPEL mais intégration possible de activiti5
FuseESB	FuseSource puis reprise par Jboss/Red-Hat	Version améliorée de ServiceMix (avec support , documentation plus précise,)
Mule ESB	MuleSoft	Bon ESB assez populaire relativement léger. Il existe une version payante/améliorée avec plus de connecteurs. Accompagné de l'IDE "MuleStudio" .
WSO2	WSO2	Basé sur OSGi (comme ServiceMix et FuseESB). Suite SOA assez complète
<i>Spring Intégration et Camel</i>	Spring , Apache	Framework d'intégration (API) pour intégration légère.

NB : L'ancienne norme JB1 (de SUN) est tombée à l'eau car trop complexe.

4. Notion de "moteur de services"

Utilités des "moteurs de services (java/bpel,...)"

* **transformations des formats des messages fonctionnels**

(ex : *addition(a,b)* <---> *add(x,y)* ,
chaîne_adresse <----> *adresse xml(rue + codePostal + ville)*)

techniquement , ce type de transformation peut être effectué via **XSLT** (via "interpréteur xslt") ou via **java** (avec dozer ou ...).

* **orchestration de services**

Les technologies "**BPEL**" et "**java/jBpm**" nécessitent des moteurs d'exécutions (interpréteurs) spécifiques pour faire fonctionner des services collaboratifs (*invokant des services élémentaires selon un algorithme/processus convenu*)

Moteurs de services (exécution, intégration)

Moteur de Services BPEL

*Service n
(script BPEL
interprété)*

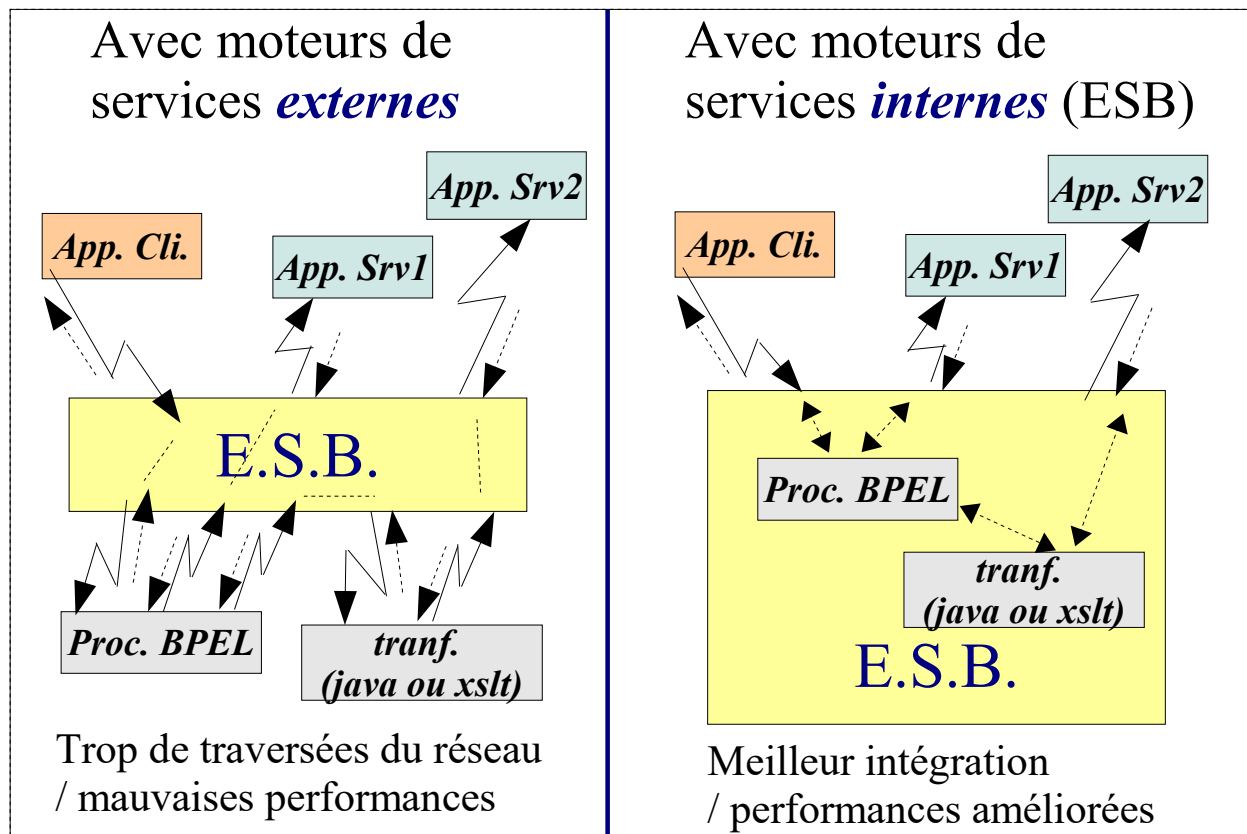
Moteur de Services Java (Serveur d'applications)

*Service Yy
(annotations jax-ws
interprétées)*

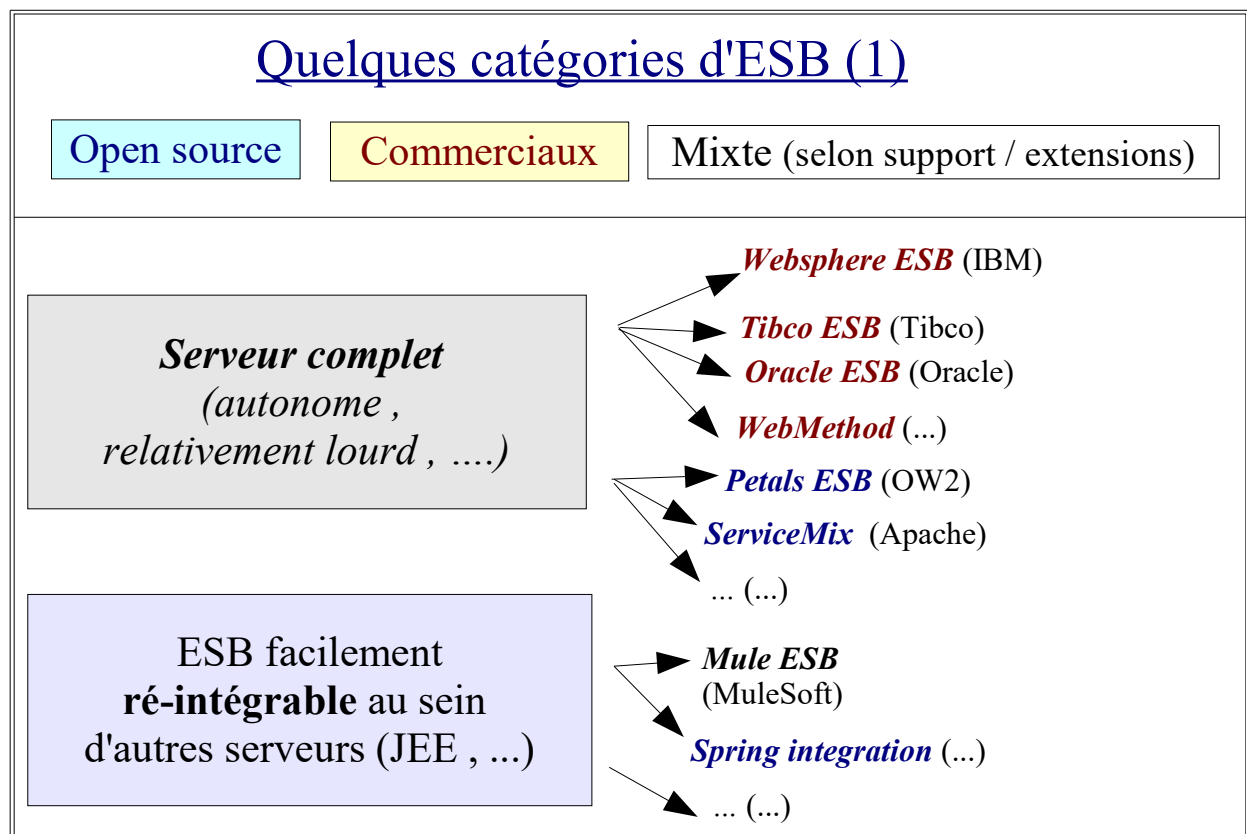
* **Définition logique
WSDL indépendante
du déploiement.**

* ***l'URL directe du point
d'accès au service est liée
au moteur de services
dans lequel le service est
déployé et interprété.***

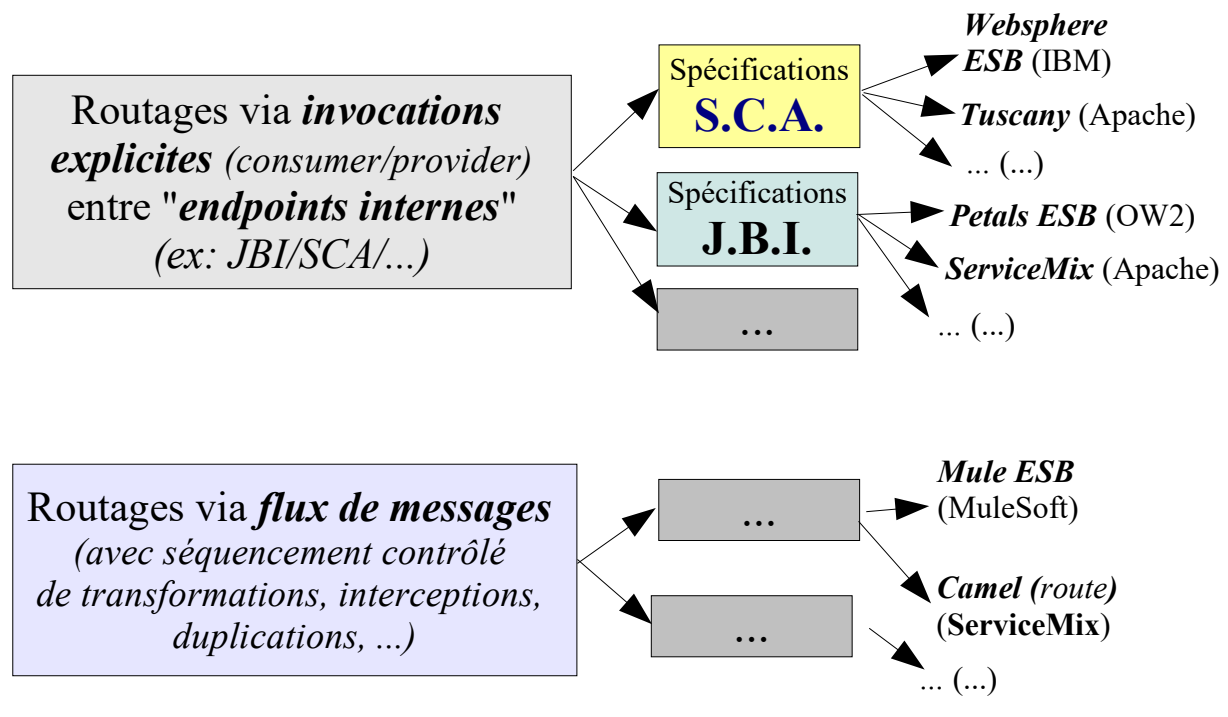
* ***Application "Moteur de
services" liée à l'ESB ?
(interne , externe ou ... ?)***



5. Catégories d'ESB

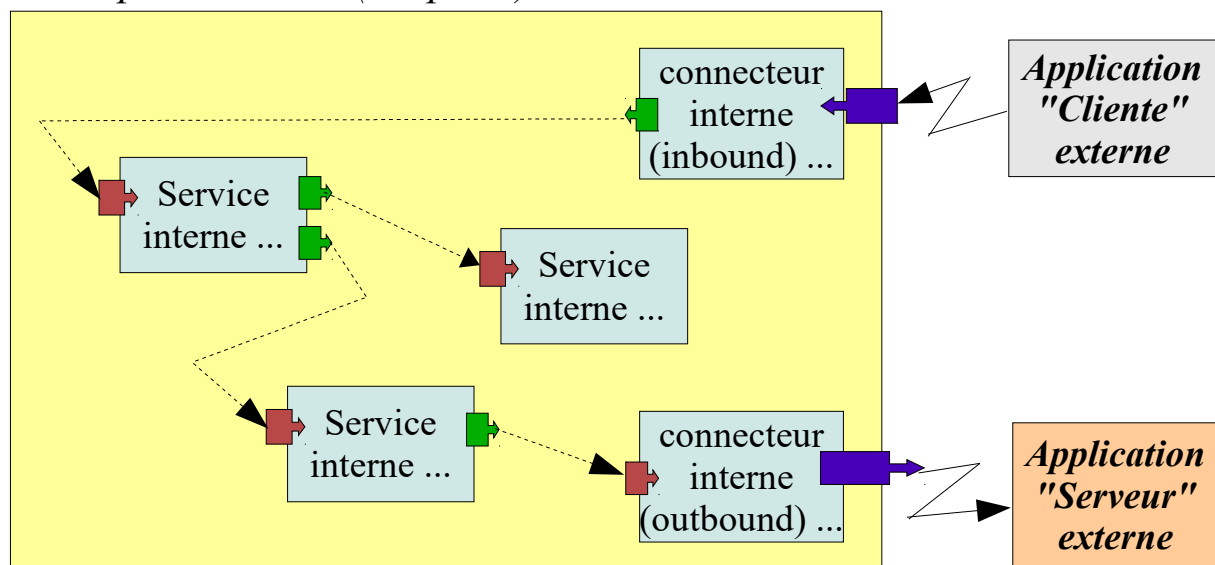


Quelques catégories d'ESB (2)



"Endpoints" internes à un ESB et connexions

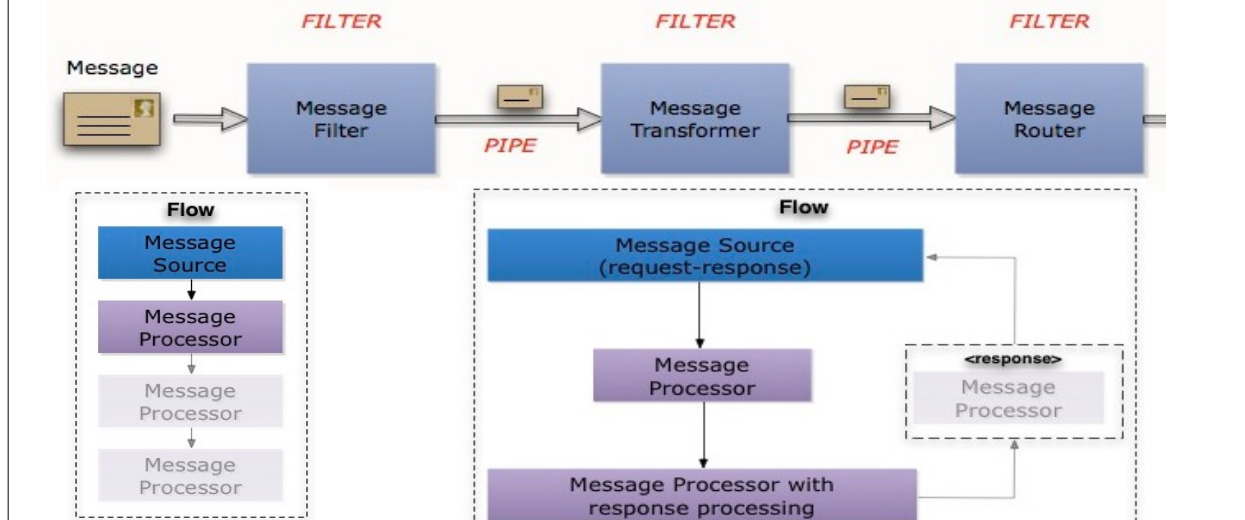
■ = point d'accès (endpoint) interne à l'esb



ESB (de type SCA, JBI ou ...)

ESB configuré(s) via des flux/flots de messages

Au lieu d'expliciter des connexions entre "endPoints internes", *certaines ESB (ex : MuleESB) se configurent en paramétrant une **suite (séquentielle) de traitements à appliquer sur des flux de messages** qui entrent dans l'ESB au niveau d'un point d'accès précis (url , ...)* .



ESB = microcosme , écosystème

*Chaque type d'ESB (SCA, JBI , Mule, ...) peut être vu comme une sorte de **microcosme (ou écosystème)** à l'intérieur duquel seront pris en charge des assemblages de services internes.*

La configuration exacte d'un service interne dépend énormément de l'ESB hôte .

D'un ESB à un autre, des fonctionnalités identiques peuvent se configurer de manières très différentes.

Attention : connecteurs quelquefois très limités et paramétrages quelquefois complexes (au sein de certains ESB) !!!!

6. SOA et mode asynchrone

Liens entre "appel de fonction" et document

Une structure XML de type

```
<commande>
  <numero>1</numero>
  <adresse>...</adresse>
  ...
</commande>
```

peut aussi bien représenter :

- * une requête vers une opération d'un service web
- * un document transmis à traiter/analyser

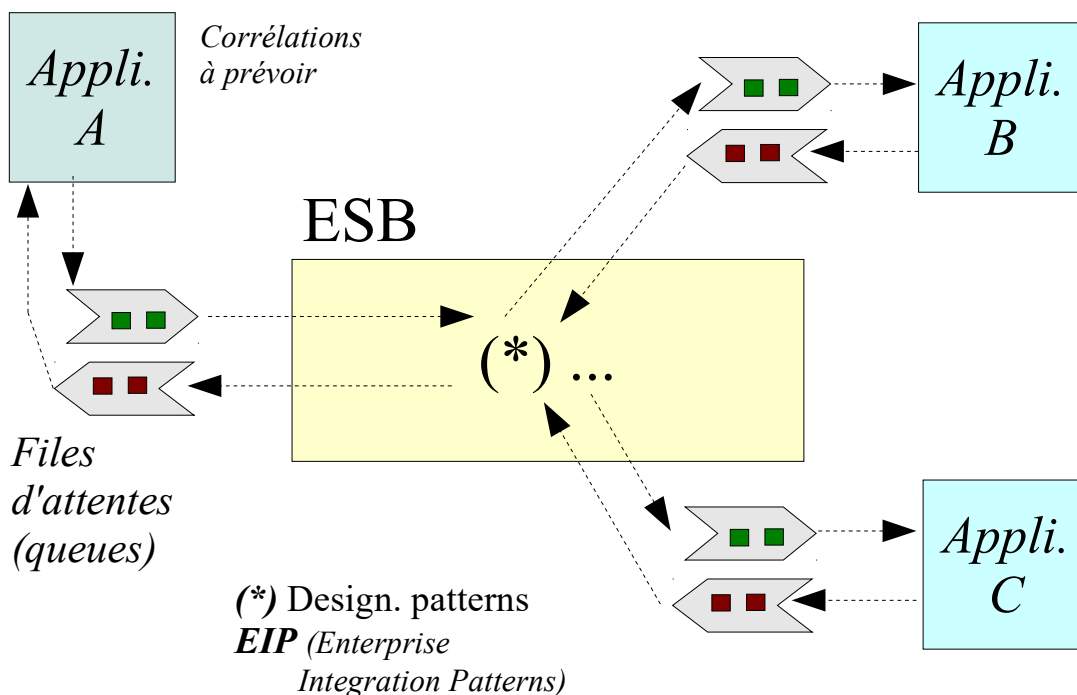
Cette structure peut être véhiculée par :

- * une enveloppe SOAP (elle même véhiculée par HTTP ou ...)
- * un message JMS (déposé dans une file d'attente)
- * ...

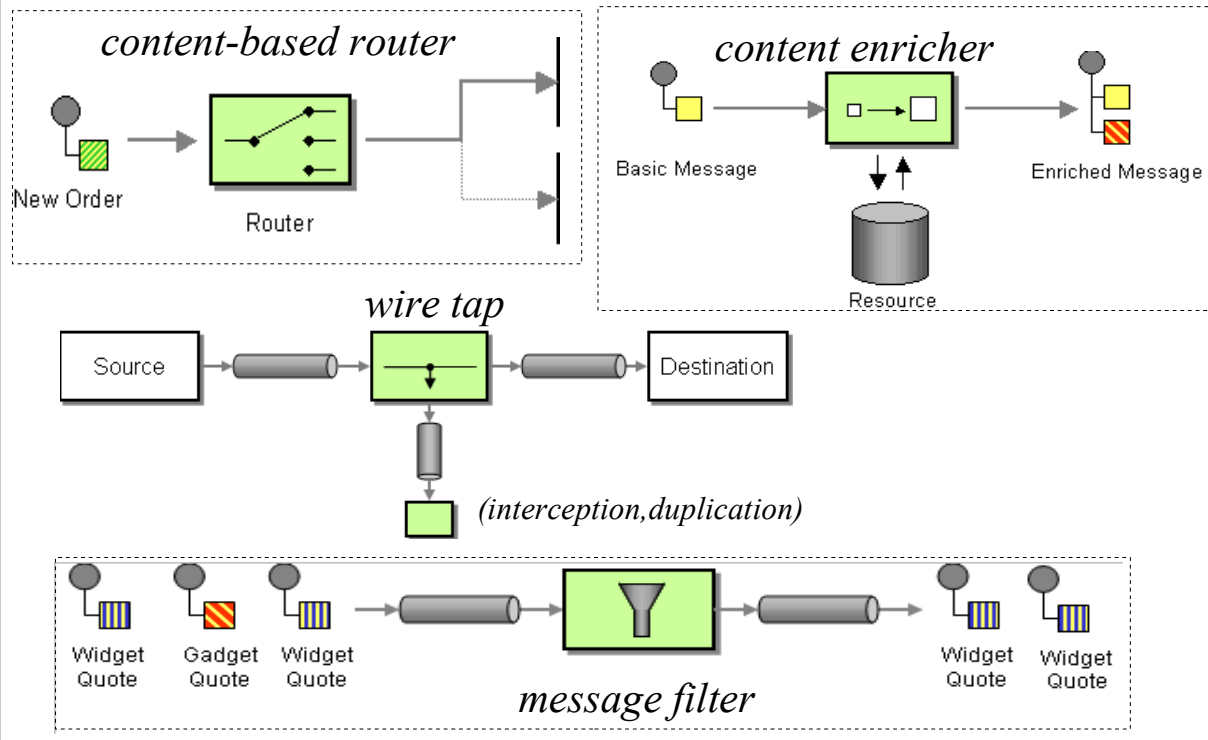
==> passerelles techniques envisageables entre :

- * synchrones et asynchrones
- * document et RPC (Remote Procedure Call) .

SOA en mode asynchrone



Quelques "design patterns" EIP (*Enterprise Integration Patterns*)



7. Mule Esb (présentation)

Mule ESB est un **ESB Open source** développé en **Java** par l'entreprise "... (MuleSoft) ...".

Cet **ESB** est volontairement basé sur une **architecture légère** (non JBI) et se configure avec des **fichiers xml** qui ressemblent à ceux de Spring.

Le concept central de Mule ESB est la notion de "**flow**" : **séquence** de *routages* , *transformations* et *services internes* .

Existant depuis longtemps et étant simple à utiliser/configurer, **Mule ESB est l'ESB Open source java le plus populaire (le plus utilisé)** .

Mule ESB peut éventuellement être intégré dans une application Java/web (et donc fonctionner au sein de Tomcat ou Jetty) .

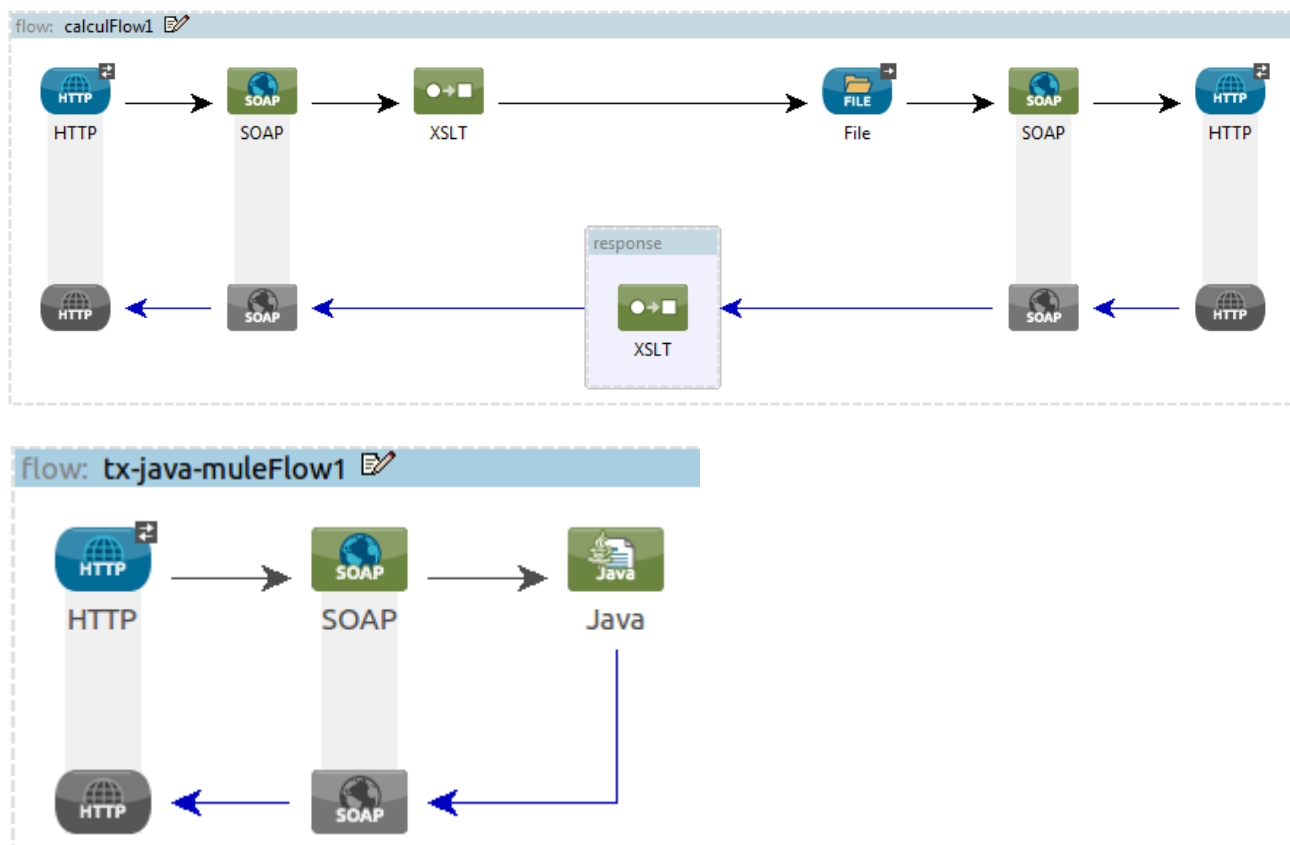
Mule ESB peut être lancé en tant que serveur autonome (standalone) .

Mule ESB existe en deux grandes versions/variantes :

- version "**CE : Community Edition**" gratuite et sans fonctionnalité pointue , sans support
- version "**EE : Enterprise Edition**" payante avec support et fonctionnalités avancées.

Le développement d'une application "Mule ESB" peut s'effectuer avec l'IDE "**MuleStudio**" (Eclipse avec plugins spécifiques) . Bien que l'icône représente une "**tête de Mule**" , cet outil fonctionne tout de même.

Les performances de Mule ESB sont assez bonnes . **On peut "charger la mule" !!!!**



8. Présentation de l'ESB "ServiceMix"

8.1. Principales fonctionnalités et spécificités

ServiceMix est un **ESB Open source** codé en **java** .

Il est actuellement à considérer comme le deuxième Bon ESB open source java (après MuleESB qui est déjà populaire depuis quelques années).

ServiceMix est un ESB de type "serveur autonome complet" et n'est pour l'instant pas prévu pour être intégrable dans Tomcat (contrairement à Mule ESB qui est plus léger) .

8.2. Variantes

Esb "**ServiceMix**" de l'éditeur "**Apache**" = version de base (gratuite , sans support).

FuseESB → version améliorée avec support / services / documentation enrichie , ...

8.3. Evolutions

Les premières versions au point de **ServiceMix (3.x)** étaient à l'origine entièrement basées sur les spécifications **JB1** .

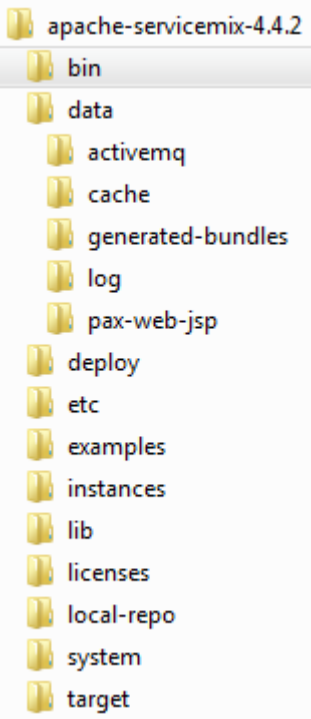
A partir de la **version 4** , la structure de l'ESB **ServiceMix** a été entièrement refondue sur un **cœur OSGi**. ServiceMix ≥ 4 s'appuie en interne sur le container OSGI "**Karaf**" de la marque "Apache"

Les spécifications OSGI en version 4.2 on introduit le déploiement de bundles OSGi basés sur les spécifications "**Blueprint**" (dérivées de **Spring DynamicModule**) .

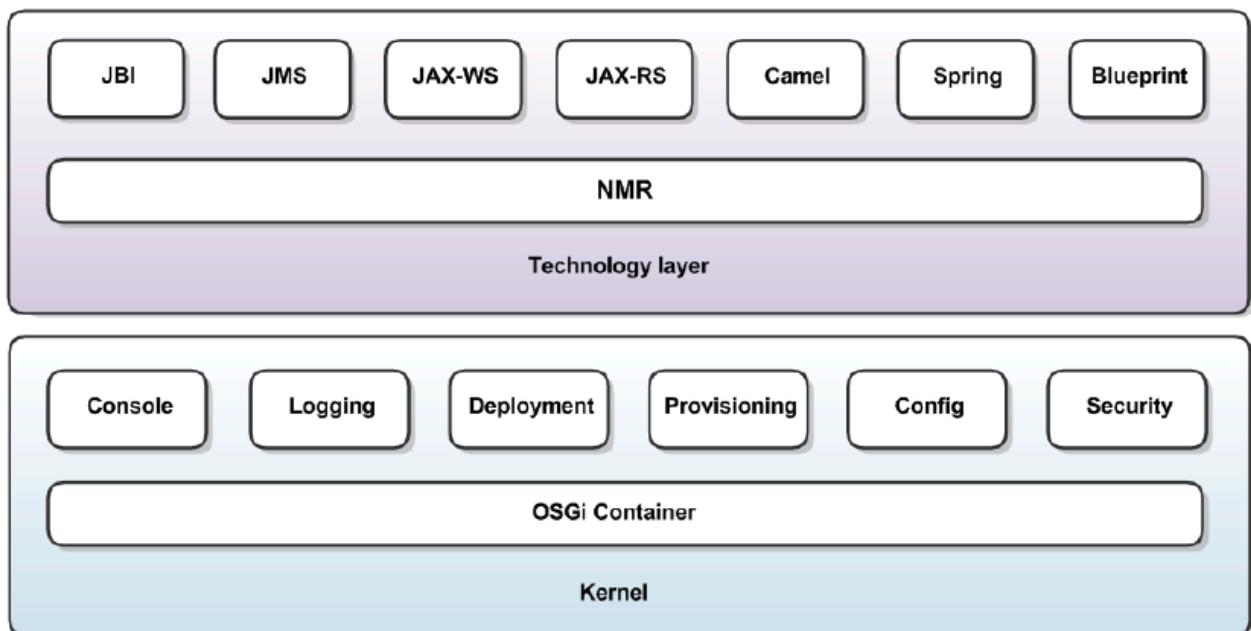
Ceci permet de déployer facilement des configurations complètes basées sur des fichiers de type "**Spring**". Une configuration "**Blueprint + cxf + camel**" pour **servicemix 4** est alors quasiment aussi **simple** qu'une configuration "MuleESB" (et bien plus simple que ce qui était auparavant attendu par la norme JBI)

<i>Versions</i>	<i>Technologies</i>	<i>Caractéristiques</i>
3.x	JB1 (pas osgi)	Époque où l'on croyait en l'avenir de JBI
4.1 , 4.2 , 4.3	JB1 sur Osgi ou Osgi sans JBI (ODE-BPEL intégré via JBI)	Transition "jbi" / "osgi"
4.4	Plus Osgi que JBI (ODE-BPEL pas intégré)	À fond Osgi ("jbi" considéré "has been")
5.x (attendue en 2013?)	Osgi avec support de NMR et ODE-BPEL	À priori reprise de la partie "NMR" de JBI (sans toute la lourdeur de JBI).

8.4. Arborescence de servicemix 4

 <ul style="list-style-type: none"> apache-servicemix-4.4.2 <ul style="list-style-type: none"> bin data <ul style="list-style-type: none"> activemq cache generated-bundles log pax-web-jsp deploy etc examples instances lib licenses local-repo system target 	<p>Répertoire "bin" --> script ".sh" ou ".bat"</p> <p>Répertoire deploy --> où il faut déposer les "xxx_sa.zip"</p> <p>Répertoire "etc" --> configurations (ex: <i>org.ops4j.pax.logging.cfg</i> comporte le paramétrage des logs de servicemix "INFO" / "DEBUG" / ...)</p> <hr/> <p>démarrage de smx 4 ---> bin/servicemix[.bat] (en mode console/shell OSGi) ou bien bin/start[.bat] (en mode tâche de fond)</p> <p>arrêt de smx 4 ---> "osgi:shutdown" dans <i>la fenêtre (shell)</i> <i>"servicemix"</i> ou bien bin/stop[.bat] (si en mode tâche de fond)</p>
---	--

8.5. Structure de servicemix 4 (OSGi , NMR et JBI)



JBI n'est plus le cœur de servicemix 4, JBI est maintenant considéré comme une simple technologie optionnelle.

8.6. Servicemix 4 et OSGi

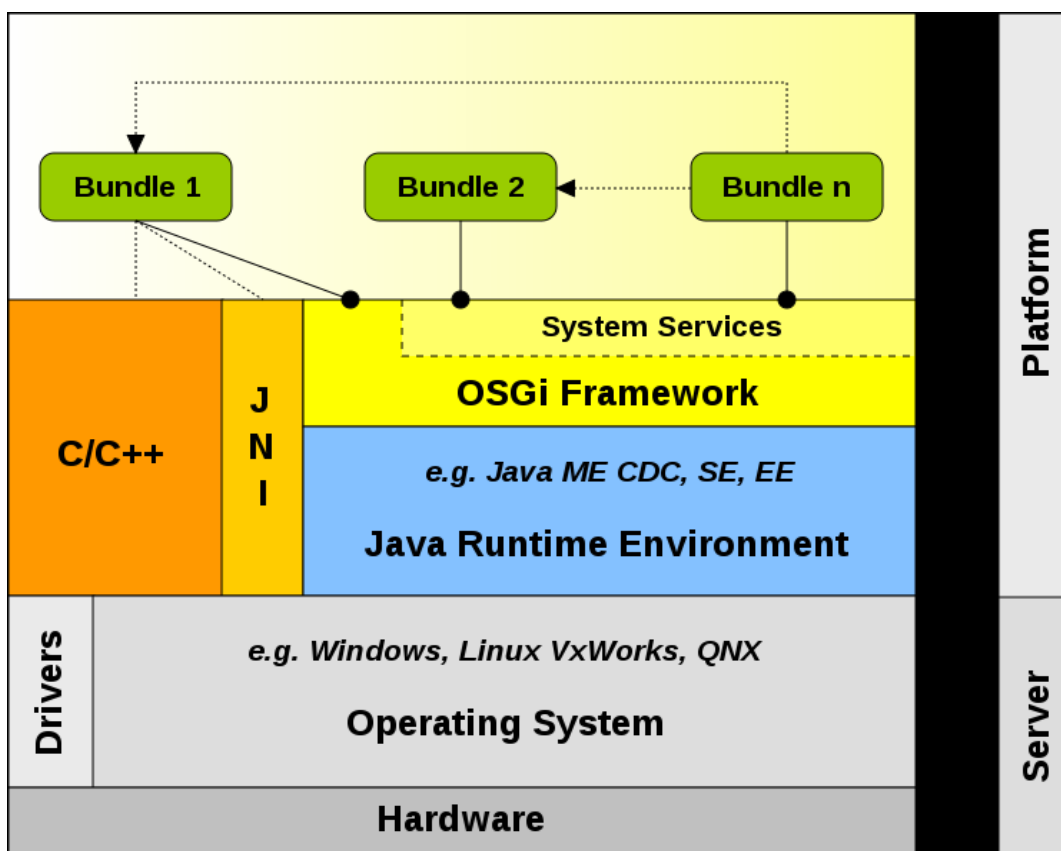
Principales commandes OSGi :

osgi:list affiche la *liste des bundles osgi déployés* avec leurs *numéros* et leurs *états* (install , active , ...)

```
[ 237] [Active] [Created] [ 60] blueprint-osgi-camelRoutes
<1.0.0.SNAPSHOT>
karaf@root>
```

osgi:shutdown arrête le serveur OSGi (servicemix).

osgi:start / **osgi:stop** / **osgi:uninstall** *<numero_bundle>* démarre , arrête ou bien désinstalle le bundle osgi identifié par le numéro renseigné .



Principaux apports de OSGi :

- Possibilité d'**arrêter / remplacer / redémarrer un (sous) module "à chaud"** (sans arrêter tout le processus d'exécution (serveur ou ...)).
- Possibilité de **gérer très finement les "classpath" de chacun des modules en permettant des réutilisations partielles contrôlées (via import/export)** .
⇒ Ceci permet **globalement de très bien optimiser la gestion de la mémoire** .
- Possibilité de faire cohabiter plusieurs versions d'une même librairie .
- **Mise en relation automatique (selon le modèle publication/souscription) des services offerts par un module avec les besoins des autres modules.**

IV - Orchestration de services

1. Notion d'orchestration et traçabilité BPM

1.1. Orchestration de services (ordonnancement de sous-services)

Le terme d'**orchestration de service** correspond à une *logique d'interaction de type "maître / subordonnés (promptement coopératifs ou pas)"*.

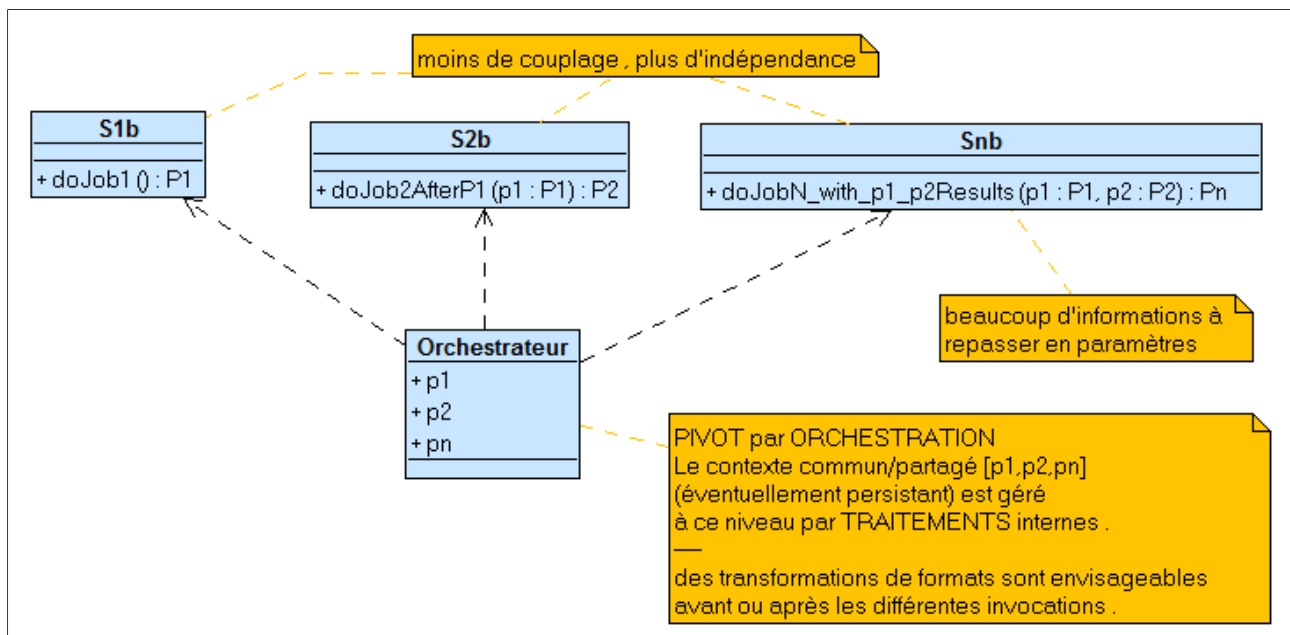
Un **service de haut niveau pilote des services de bas niveaux** en invoquant des opérations dans un ordre bien établi et/ou selon une logique conditionnelle .

L'orchestration de services peut se mettre en œuvre de différentes façons (plus ou moins complexes/élaborées) :

- **service web synchrone (ex : java/soap) invoquant immédiatement d'autres services synchrones** selon un ordre précis et une logique métier (éventuellement conditionnée) .
[peut se programmer avec uniquement la technologie "web-service"]
- **service web avec une branche asynchrone correspondant à un processus métier long et déclenchant d'autres services asynchrones** (ex : via files d'attentes JMS ou mail) .

[nécessite éventuellement une technologie évoluée dédiée à l'orchestration telle que BPEL ou bien activiti ou jbpn]

Pivot par orchestration :

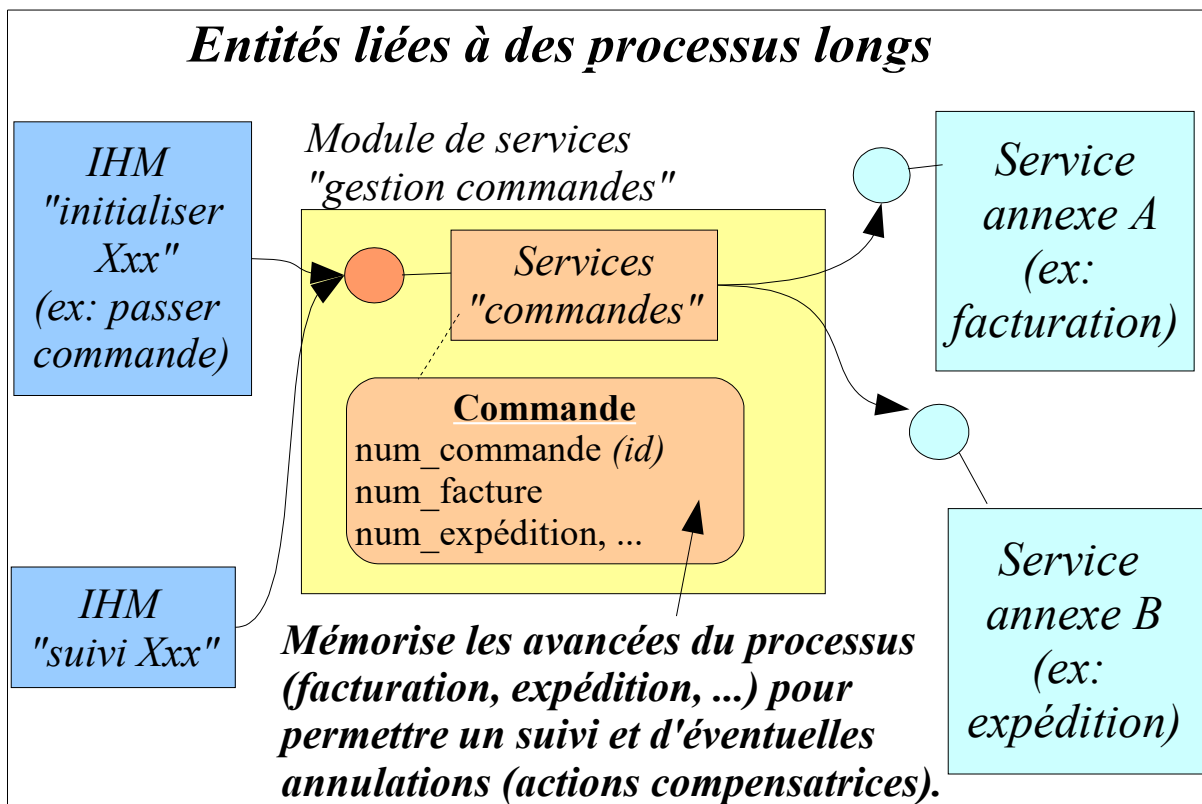


Remarque: entre deux invocations de sous-services , le service "orchestrateur" a sous besoin de convertir d'un format à un autre les données "résultats du sous-service-1" en données "paramètre-d-entrée-du-sous-service2" .

1.2. Traçabilité BPM

La logique métier liée à un processus long peut potentiellement être codée avec :

- une base de données spécifique (pour enregistrer l'état d'avancement du processus et toutes les données qui y sont liées)
- une multitude de petits services (qui ne gèrent chacun qu'une partie du processus global).



Un processus métier long peut également être mis en œuvre avec une technologie d'orchestration évoluée (telle que BPEL ou bien activiti/jbpm) .

L'exécution des différentes étapes sera alors pilotée/supervisée depuis la logique du processus.

Et au final, le **principal intérêt des technologies BPM** (BPMN , BPEL) réside dans la **traçabilité** résultante entre les **éléments de la modélisation** (BPMN) du processus et les **éléments du code exécutable**.

Autrement dit, sans technologie BPM/BPEL, il peut parfois être difficile de localiser les éléments du code logiciel à modifier si le processus métier évolue.

Par contre, avec une technologie d'orchestration BPM , on localise la mise à jour à effectuer immédiatement car l'exécution de code est directement bâtie sur le fil conducteur du processus.

2. BPEL (présentation)

BPEL (*Business Process Execution Language*)

BPEL (*BPEL4WS* renommé *WS-BPEL*) a été créé par le consortium OASIS et vise à encoder en Xml des services Web de haut niveau qui orchestrent ou pilotent des services Web de plus bas niveaux.

On parle généralement en terme de "*processus collaboratif*" pour désigner le type de services produit via BPEL .

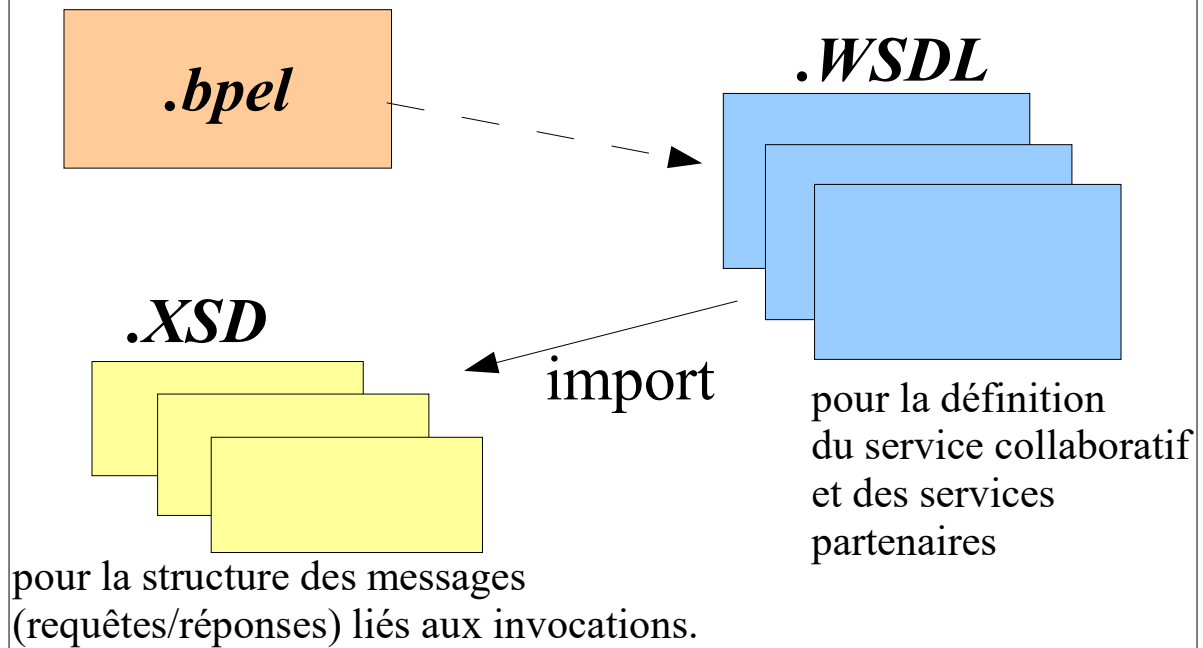
NB: Deux grandes versions :

- * BPEL 1.1 (2003) et BPEL 2.0 (2007)
- * En pratique quelques variantes d'interprétations selon le moteur d'exécution (ODE, Orchestra , ...)

Principales fonctionnalités de BPEL:

- **invoquer** des services externes (partenaires)
- gérer finement le *contenu des messages de requêtes et de réponses* avant et/ou après les différentes invocations.
- ajouter une logique métier de haut niveau (tests , vérifications, *copies et calculs de valeurs à retransmettre*).
- **orchestrer / séquencer** de façon adéquate les invocations(*synchrones ou asynchrones*) de services partenaires.

Technologies XML utilisées par BPEL:



Quelques implémentations : moteurs BPEL

- **ODE** (open source , apache)
à intégrer dans *Tomcat* ou *ServiceMix*
- **BPEL-SE** de OpenESB et GlassFish V2 (*SUN*)
- **Orchestra** (Open source, *OW2*)
- **BizTalk Server** (*Microsoft*)
- **WebSphere Process Server** (*IBM*)
- **Oracle BPEL Process Manager**
- **SAP Exchange Infrastructure**
- **ActiveVOS**
- ...

2.1. Structure BPEL

Structure générale d'un processus BPEL:

```
<?xml ... ?>
<process ...>
  <import ...="...wsdl"/>
  <import ...="...wsdl" />
  <partnerLinks>
    <partnerLink .../>
  </partnerLinks>
  <variables>
    <variable .../>
  </variables>
  ... instructions ...
</process>
```

*déclarations logiques
des services partenaires*
(types précis définis dans
wsdl , partenaire spécial =
le process bpel lui même)

déclarations de variables
(messages transmis , parties
recopiées ou calculées)
(types précis définis dans
schémas ".xsd" des ".wsdl")

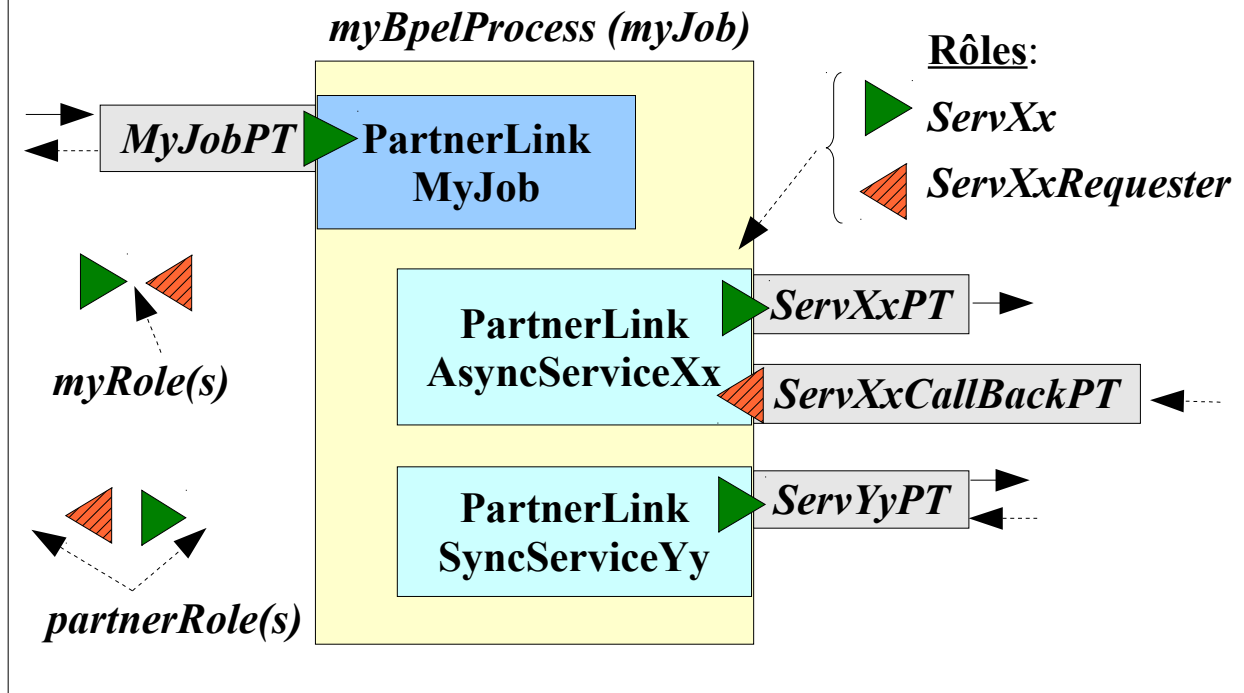
Séquence type d'un processus BPEL:

```
<process ...> ....
  <sequence>
    <receive partnerLink=".."
      operation="..." variable="..." />
    <assign>
      <copy> from ... to ... </copy>
    </assign>
    <invoke partnerLink=".."
      operation=".." inputVariable=
      "..." outputVariable="..." /> ...
    <reply partnerLink=".."
      operation="..." variable="..." />
  </process>
```

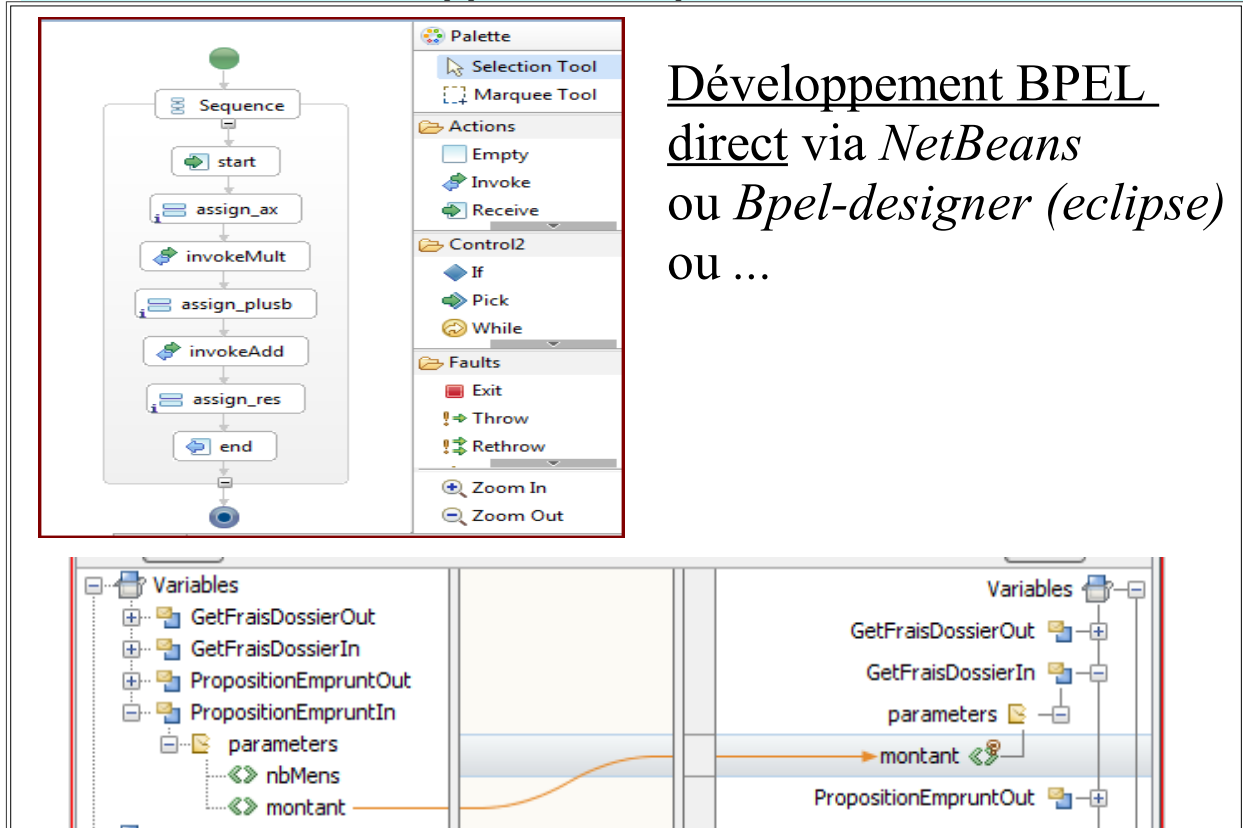
*réception d'une
requête et affectation
du message
dans une variable.*

*invocation d'une
opération
sur un service Web
partenaire*

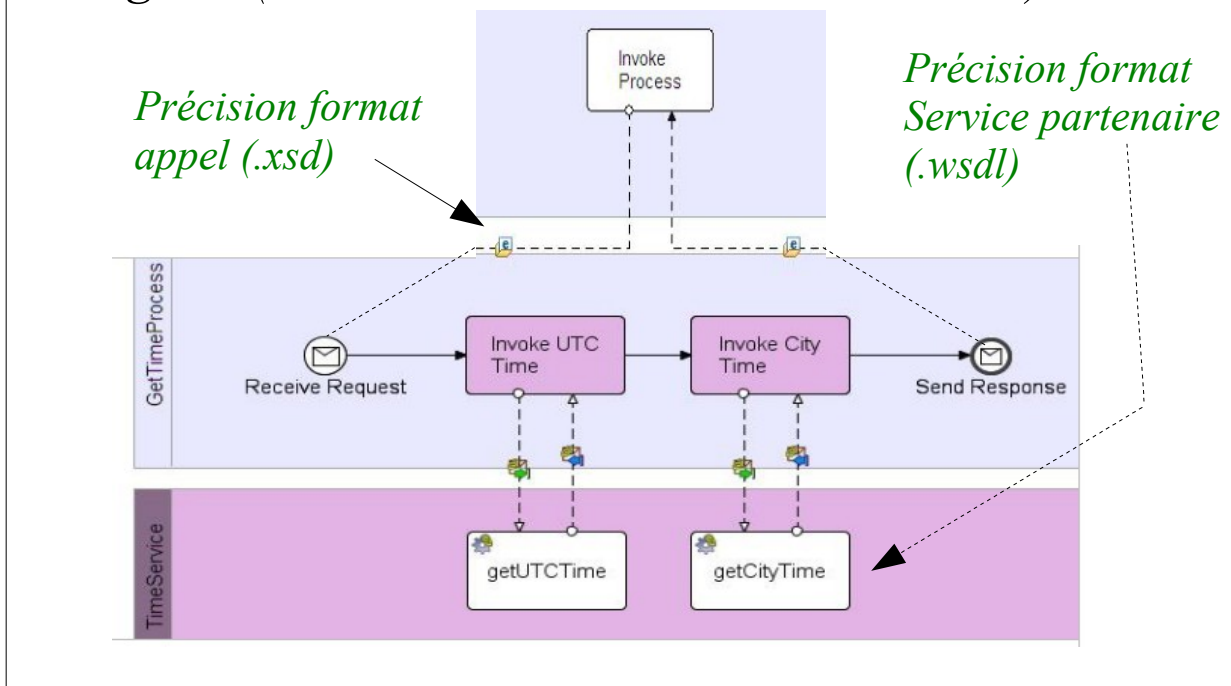
BPEL – PartnerLinks & PortType:



2.2. Vues sur le développement de processus BPEL

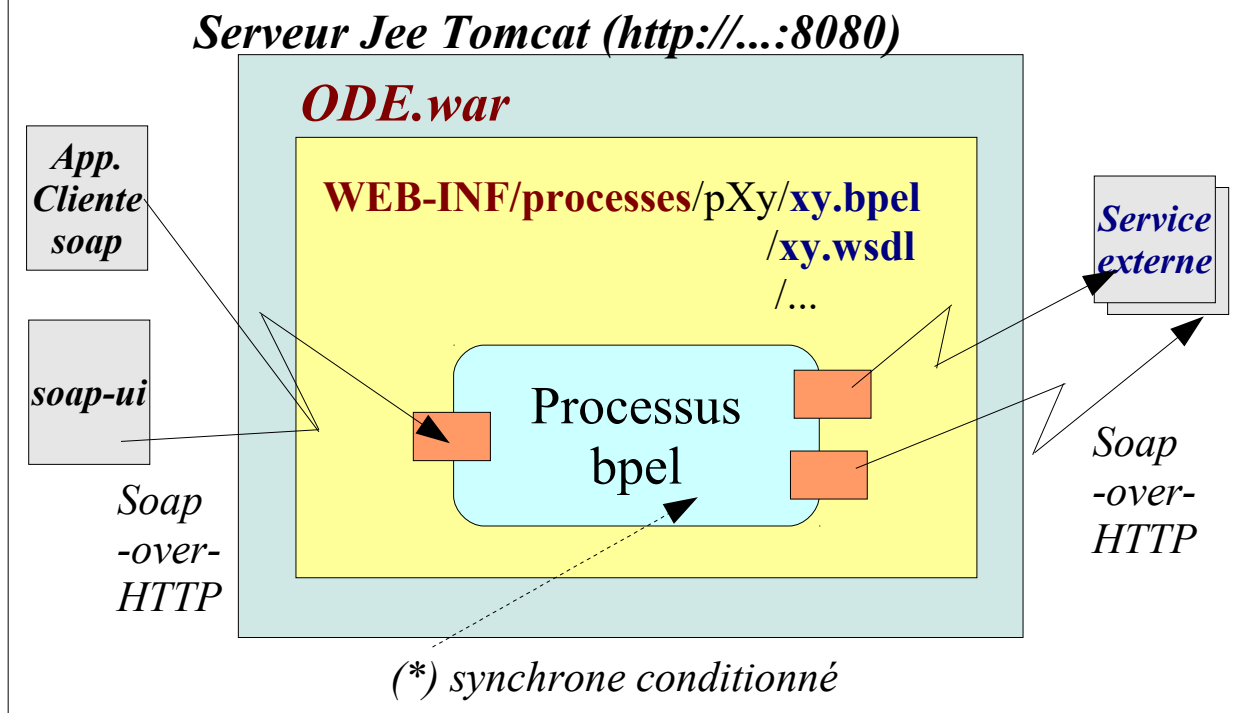


Développement BPEL indirect depuis *Intalio BPMNs Designer* (*modélisation BPMN → code BPEL*)



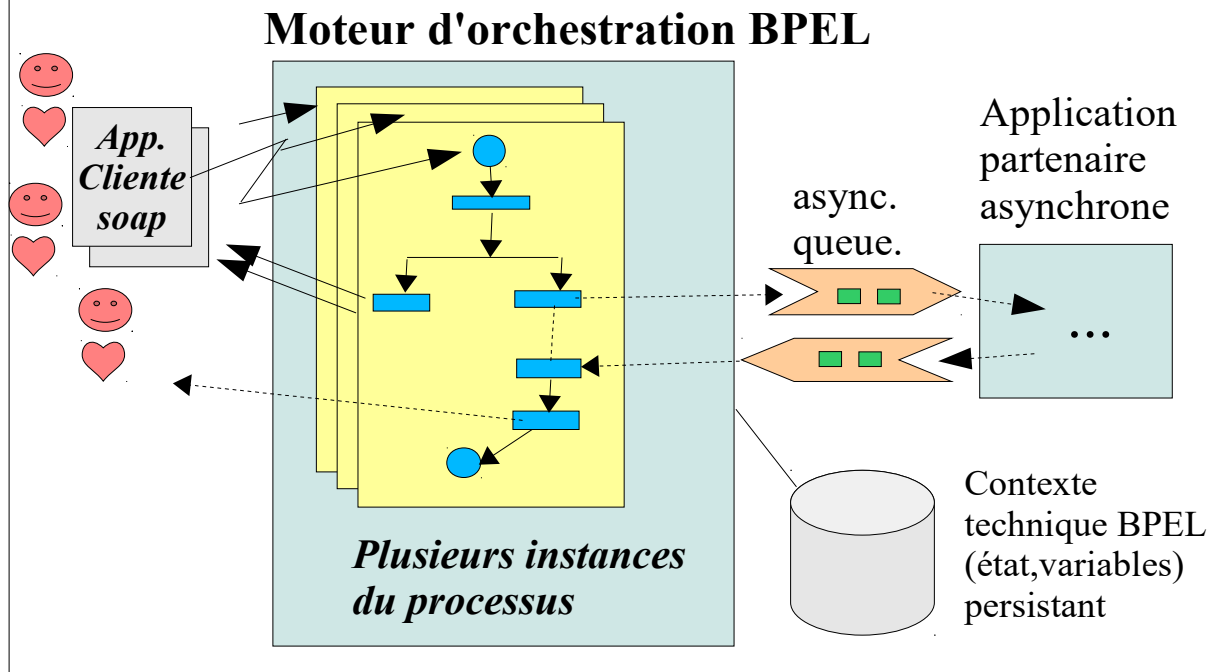
2.3. BPEL en mode synchrone et exemple d'intégration

Exemple d'intégration 1: BPEL synchrone via ODE dans Tomcat

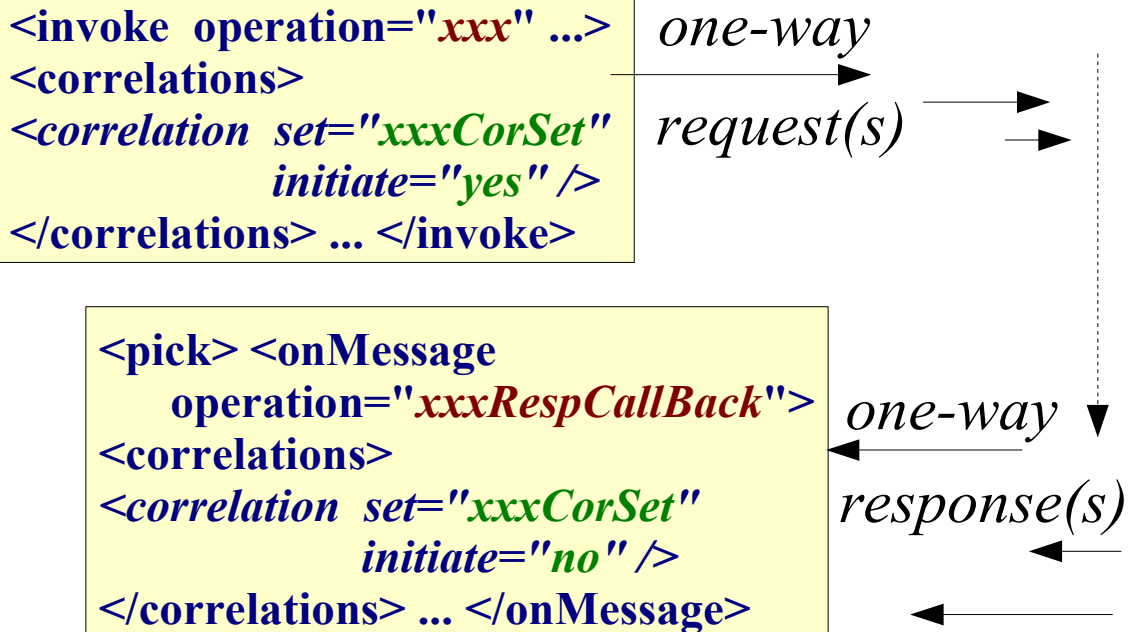


3. BPEL en mode asynchrone

BPEL (dans tout son potentiel) en mode asynchrone



BPEL - mode asynchrone et corrélations



BPEL – *pick* (receive (if correlation) ... or)

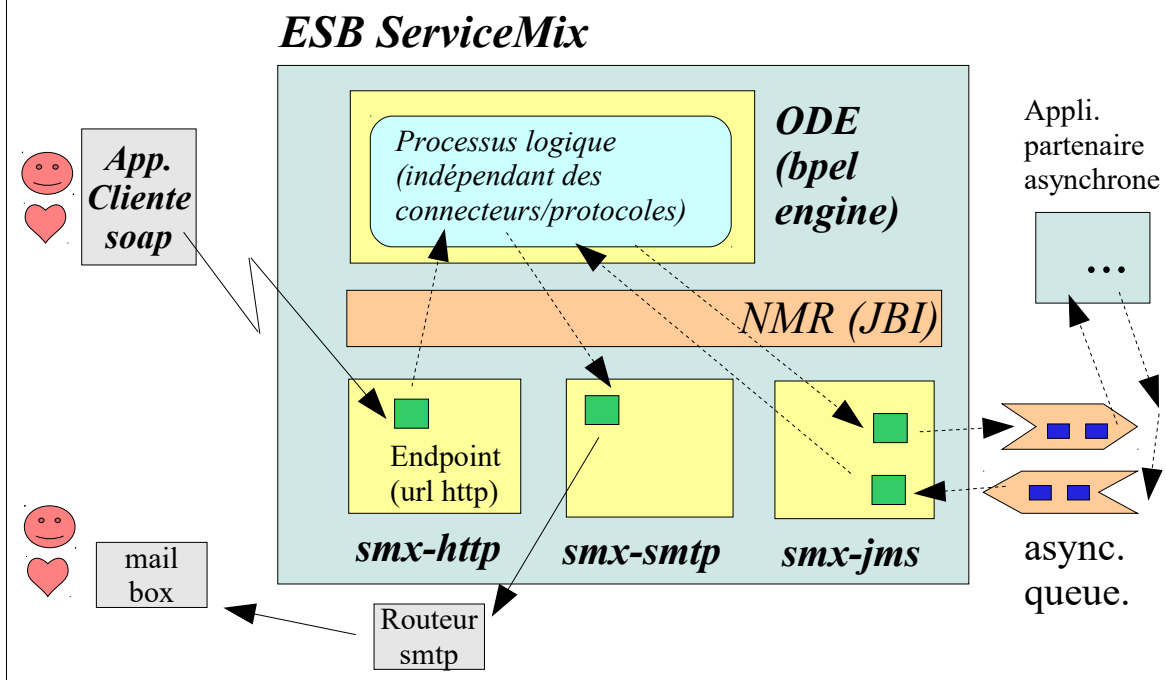
```

<pick>
  <onMessage operation="xxxResponse"
    ...> <correlations> ...</correlations>
    <!-- gérer la réponse positive (acceptation) -->
  </onMessage>
  <onMessage operation="xxxReject"
    ...> <correlations> ...</...>
    <!-- gérer le refus -->
  </onMessage>
  ...
</pick>

```

callback

Ex. d'intégration 2: BPEL asynchrone via ODE dans ServiceMix



4. BPMN

4.1. Présentation de BPMN

BPMN signifie *Business Process Management Notation*

- Il s'agit d'un formalisme de modélisation spécifiquement adapté à la modélisation fine des processus métiers (sous l'angle des activités) et prévu pour être transposé en BPEL ou jBpm.
- Un diagramme **BPMN** ressemble beaucoup à un diagramme d'activité UML . Les notions exprimées sont à peu près les mêmes.

Les différences entre UML et BPMN sont les suivantes:

- la syntaxe des diagrammes d'activités UML est dérivée des diagrammes d'états UML et est plutôt orientée "technique de conception" que "processus métier"
- à l'inverse la syntaxe des diagrammes BPMN est plus homogène et plus parlante pour les personnes qui travaillent habituellement sur les processus métiers.
- UML étant très généraliste et avant tout associé à la programmation orientée objet, il n'y a pas beaucoup de générateurs de code qui utilisent un diagramme d'activité UML
- à l'inverse quelques éditeurs BPMN sont associés à un générateur de code BPEL et la version 2 de Bpmn peut être accompagnée d'extensions "java / jbpm ou activiti)" pour le rendre exécutable. Les éléments fin d'une modélisation BPMN ont été pensés dans ce sens.

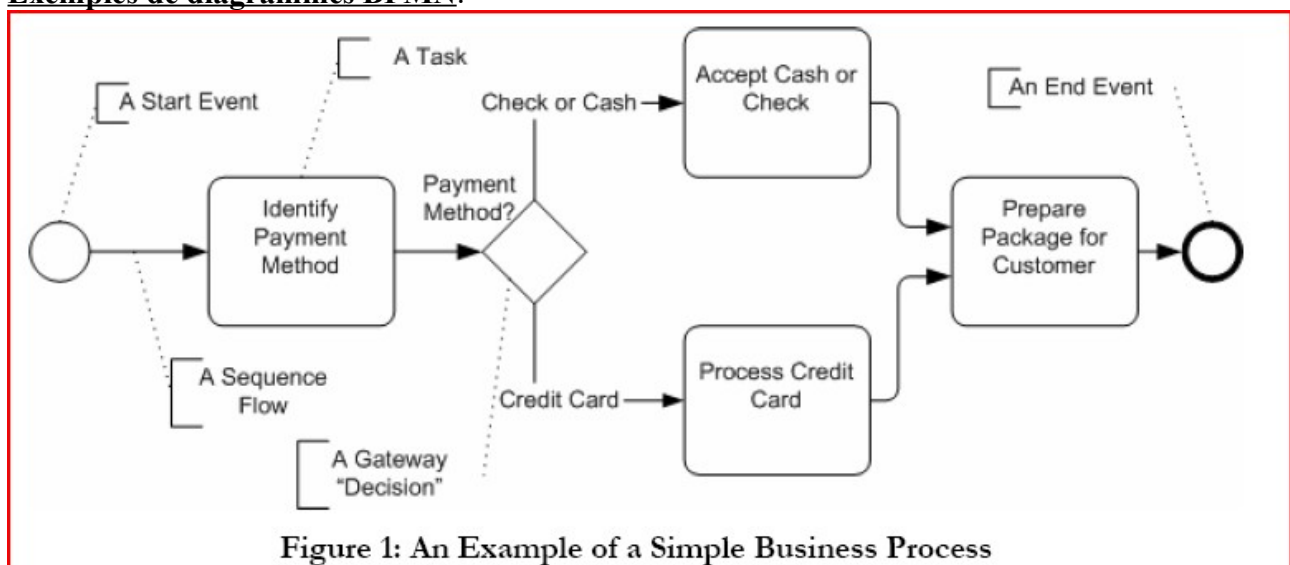
Outils concrets pour la modélisation BPMN :

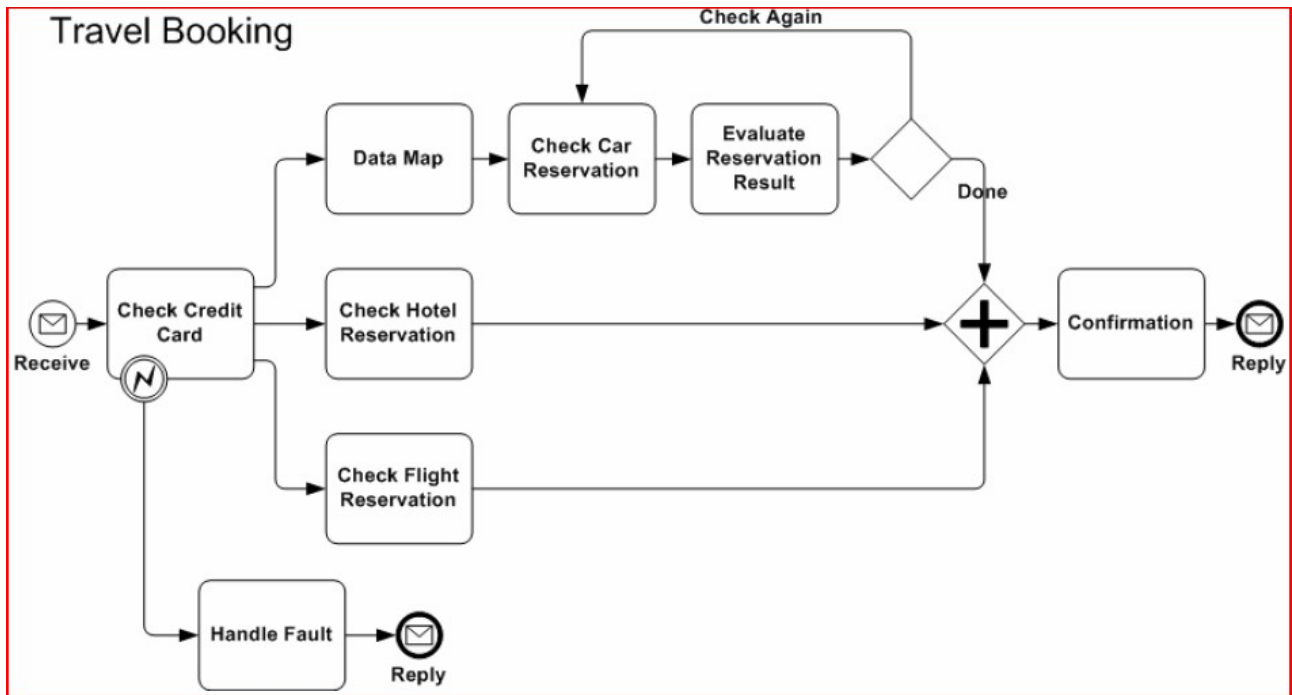
- **Intalio BPMN** .
- **Bizagi Process Modeler** (version gratuite)
- **Editeur BPMN2 intégré à l'IDE eclipse** (**drools / jbpm5**).
- **Yaoqiang bpmn(2) editor** (très bien : versions gratuites et payantes).

NB : Certains outils BPMN 1.x (tels que **Bizagi**) sont capables d'effectuer des imports/exports au format **XPDL** . **XPDL** signifie "*Xml Process Definition Language*" : c'est une sorte de sérialisation Xml de BPMN.

Des outils BPMN récents (supportant **BPMN 2**) peuvent quelquefois directement utiliser xml comme format natif (ex : **Jboss drools/jbpm5** , **yaoqiang bpmn editor**) .

Exemples de diagrammes BPMN:





4.2. BPMN 1 et 2 , utilisations

BPMN1 est exclusivement utilisé pour la **modélisation**.

BPMN2 est essentiellement utilisé pour la **modélisation**.

Etant maintenant associé à un format de fichier "xml" rigoureusement normalisé par l'OMG, BPMN2 peut aussi être directement utilisé pour de l'**exécution de code** (via certaines extensions "xml" telles que les extensions bpmn2 pour jbpms ou pour activiti).

"**Intalio Bpmns**" est un **outil de développement concret** qui permet de **générer du code BPEL** à partir d'une **modélisation BPMN** et de certains ajouts de spécificités techniques (WSDL,XSD,...).

Les grands éditeurs "IBM, Oracle, ..." offrent souvent des solutions proches/équivalentes.

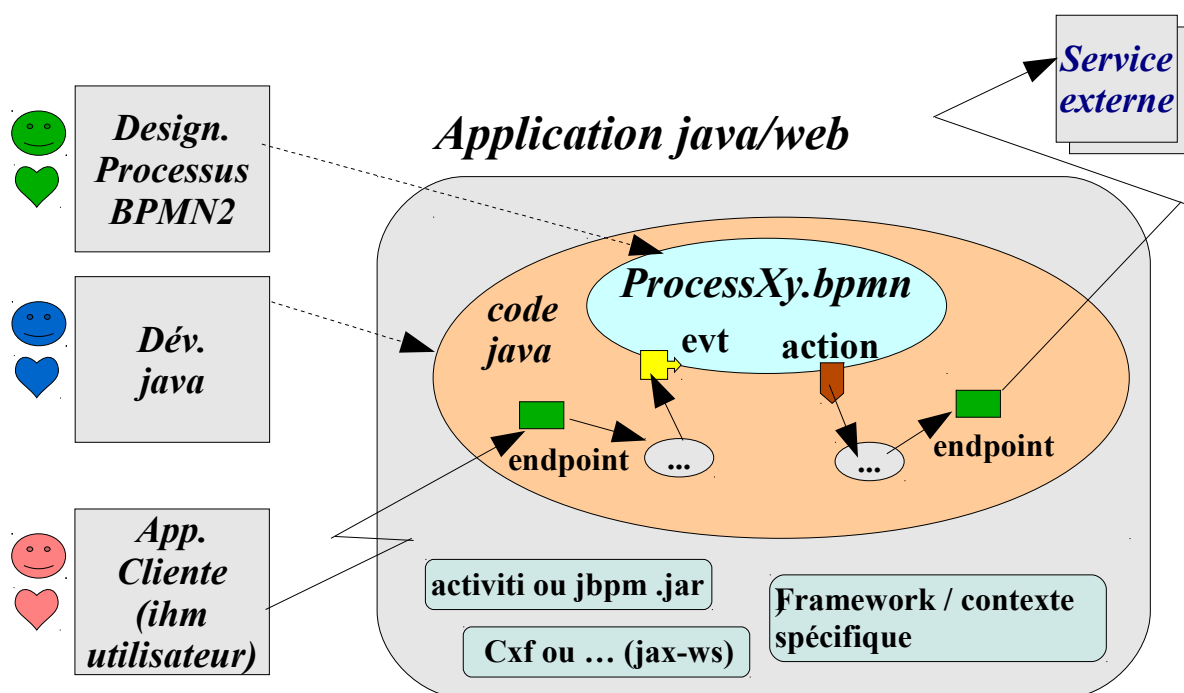
Activiti (compatible "spring") et **jbpm5** (associé à jboss/drools) sont deux technologies proches (en java) qui permettent d'effectuer une **modélisation BPMN2** (via quelques plugins eclipse) pour ensuite **permettre une exécution de code java directement pilotée par la modélisation bpmn2**.

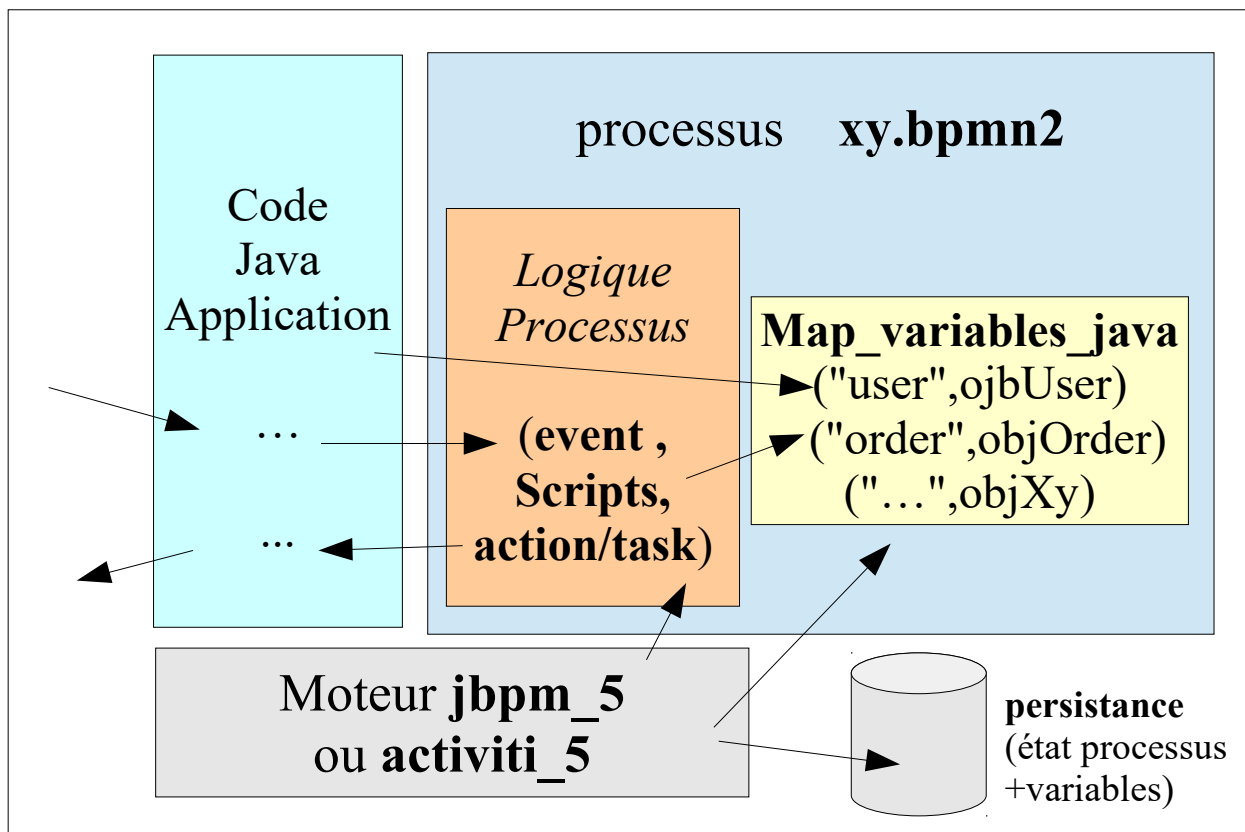
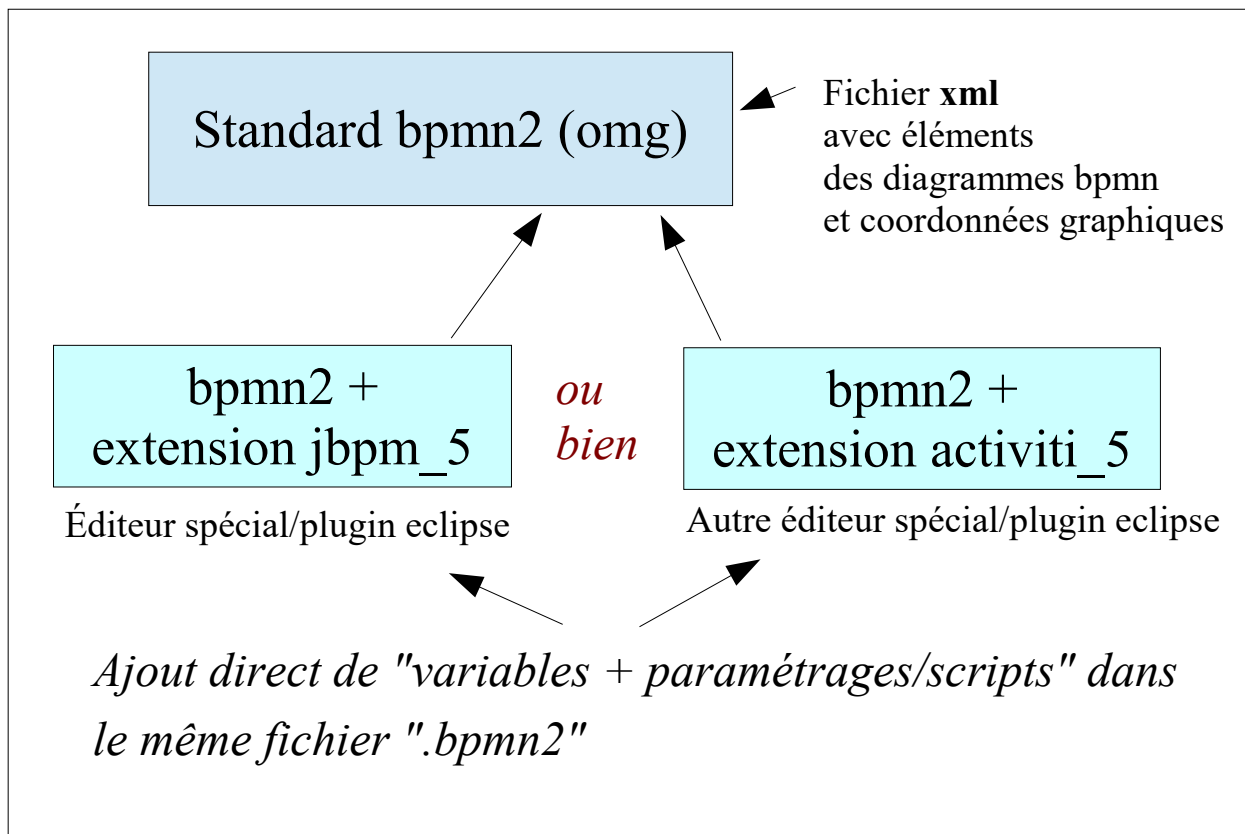
5. jBPM et activiti (présentation)

Eventuelle alternative (java/BPM) à BPEL (xml)

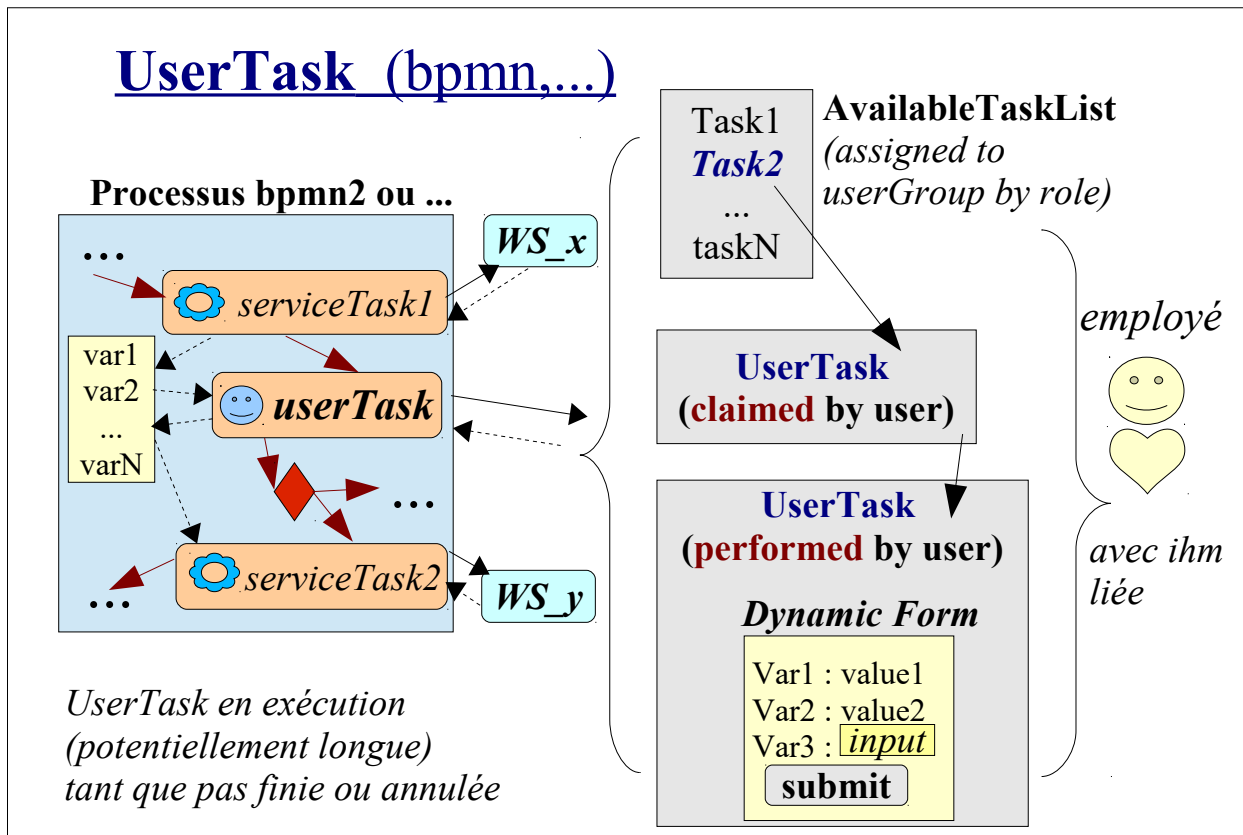
- * **jBPM_5** et **activiti_5** sont des **moteurs de workflow** à intégrer des applications **java** (sous forme de ".jar").
- * *Ces deux technologies (très proches) s'appuient directement sur le standard **BPMN2**.*
- * **jBPM_5** est en fait un des constituants de **Drools 5** et s'intègre facilement dans **Jboss 7.1** ou **Jboss EAP 6**
- * **activiti_5** est plus léger et peut facilement s'intégrer dans le framework **Spring** (et donc dans **tomcat 7**)
- * Bien que nécessitant pas mal de code java (variables, invocations,...), jBPM5 et activiti peuvent être utilisés pour effectuer de l'orchestration de services.

Principe de fonctionnement de (java/BPM)





6. UserTask



Appelée "UserTask" ou "HumanTask", ce type de tâche correspond à :

- Une tâche réalisée par une personne humaine (ex : employé) à l'aide d'un terminal informatique.
- Une tâche qui ne sera considérée comme "terminée et réussie" que lorsqu'un des employés aura effectué la tâche (ce qui peut prendre plusieurs minutes/heures/jours)
- Une tâche dont les entrées/sorties du formulaire dynamique sont liées (par paramétrages) à certaines variables du processus bpmn2 ou bpel .

V - Le "Cloud Computing"

1. Le "Cloud Computing" (présentation)

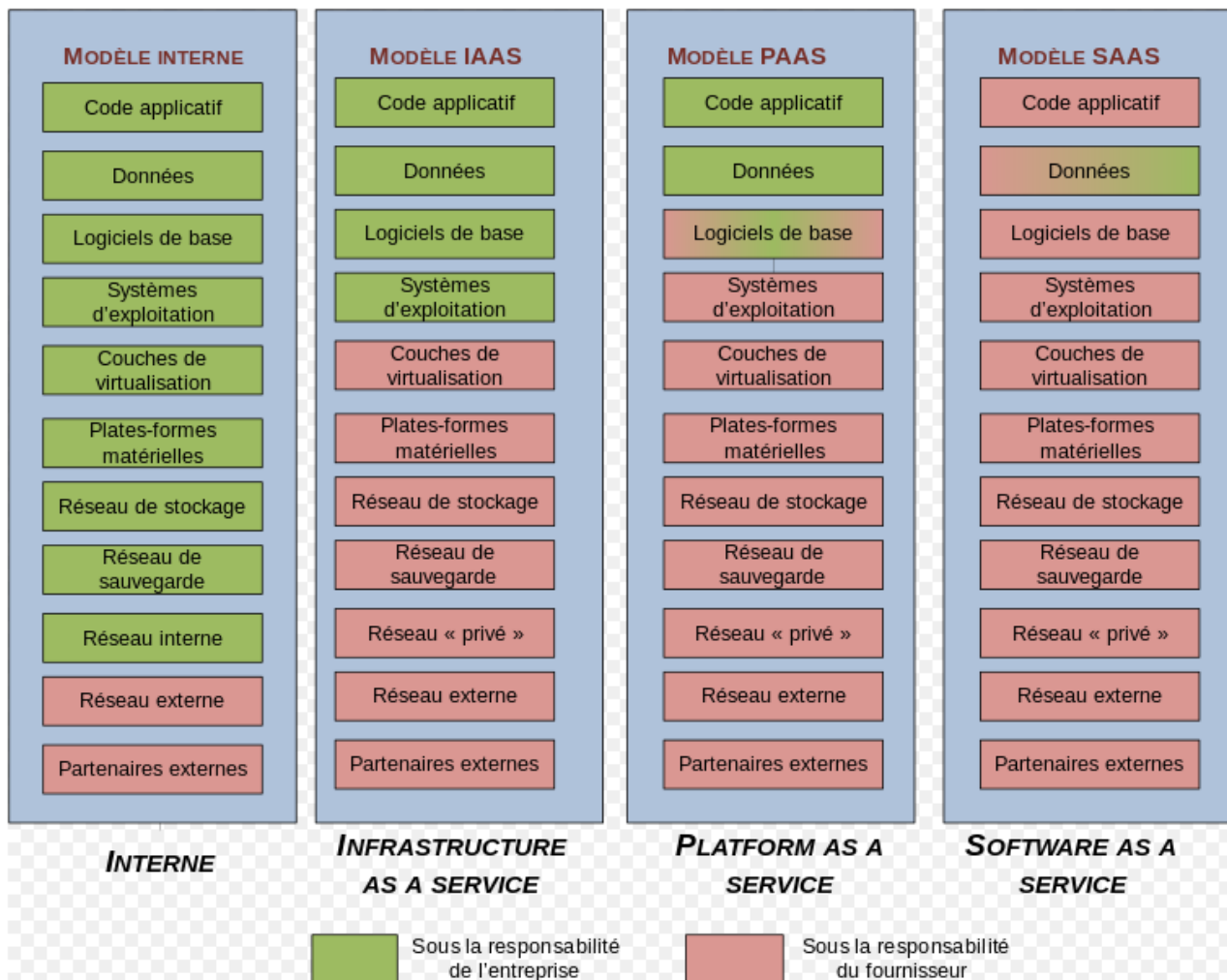
1.1. Architecture / terminologie "Cloud"

Un nuage (anglais *cloud*) est une métaphore désignant un ensemble de matériel, de raccordements réseau et de logiciels qui fournit des services sophistiqués que les individus et les collectivités peuvent exploiter à volonté depuis n'importe où dans le monde.

Le *cloud computing* est un basculement de tendance : au lieu d'obtenir de la puissance de calcul par acquisition de matériel et de logiciel, le consommateur se sert de puissance mise à disposition par un fournisseur via Internet.

Les caractéristiques essentielles d'un nuage sont la disponibilité mondiale en self-service, l'élasticité, l'ouverture, la mutualisation et le paiement à l'usage :

- *ressources en self-service*, et adaptation automatique à la demande. La capacité de stockage et la puissance de calcul sont adaptées automatiquement au besoin d'un consommateur. Ce qui contraste avec la technique classique des hébergeurs où le consommateur doit faire une demande écrite à son fournisseur en vue d'obtenir une augmentation de la capacité - demande dont la prise en compte nécessite évidemment un certain temps. En *cloud computing* la demande est automatique et la réponse est immédiate.
- *ouverture*. Les services de *cloud computing* sont mis à disposition sur l'Internet, et utilisent des techniques standardisées qui permettent de s'en servir aussi bien avec un ordinateur qu'un téléphone ou une tablette.
- *mutualisation*. La mutualisation permet de combiner des ressources hétérogènes (matériel, logiciel, trafic réseau) en vue de servir plusieurs consommateurs à qui les ressources sont automatiquement attribuées. La mutualisation améliore la scalabilité et l'élasticité et permet d'adapter automatiquement les ressources aux variations de la demande.
- *paiement à l'usage*: la quantité de service consommée dans le *cloud* est mesurée, à des fins de contrôle, d'adaptation des moyens techniques et de facturation



Niveaux (<i>aas=as a service</i>)	Caractéristiques
SAAS (Software aas)	<p>On loue un logiciel entièrement géré par le fournisseur (ServApp + Base_données ,).</p> <p>On a simplement besoin de postes clients avec des navigateurs internet .</p> <p>Seule la sauvegarde des données (copies locales) est facultativement à gérer.</p>
PAAS (Platform aas)	<p>On loue une plate-forme logicielle d'un certain type (ex : Serv App Java/Jee Tomcat" + bases MySql ou) ou bien "Node-js" + "MongoDB") entièrement gérée par un fournisseur d'hébergement puis l'on déploie de code d'une application à faire fonctionner</p>
IAAS (Infrastructure aas)	<p>On loue des machines virtuelles auprès d'un fournisseur d'hébergement (ex : Gandi , OVH) avec par exemple un O.S. "linux" , puis on installe la base de données et le serveur d'applications que l'on préfère (ex : Mysql et Tomcat7) pour ensuite installer et faire fonctionner des applications.</p>
Tout en interne	On achète les ordinateurs "serveurs" , les "systèmes

d'exploitation" , les logiciels "SGBDR" , "compta" , "...". On administre tout nous même.
--

Plus on s'appuie sur une couche basse , plus on a de liberté pour choisir l'O.S. , les logiciels que l'on préfère mais on doit gérer soit même plein de choses.

Plus on s'appuie sur une couche haute , moins on a de choses à gérer mais plus on devient dépendant de l'offre (large ou pas) et de la qualité de service (bonne ou pas) du fournisseur.

La maîtrise d'œuvre sera souvent intéressée par une offre Iaas ou Paas . Cela lui permettra de fournir sa "valeur ajoutée" sous forme de logiciel à déployer et faire fonctionner de façon à ensuite satisfaire les besoins d'une maîtrise d'ouvrage.

Spécificité importante d'une application Saas :

- bien gérer les sauvegardes des données
- bien gérer la sécurité et la disponibilité
- bien gérer la séparation des données "client1" , "client2" , ... , "clientN" si une application Saas est partagée/louée par plusieurs clients

1.2. Technologies pour le "cloud computing"

Windows Azure (en tant que plate-forme),

→ Plateforme "Cloud" de *Microsoft*

Amazon Web Services (AWS)

→ Offre cloud de la société Amazon (historiquement une des premières offres)

Google App Engine

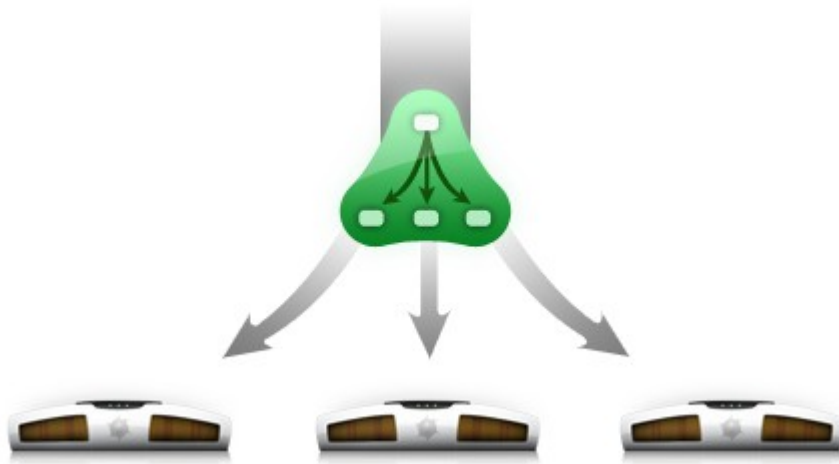
→ API ...

1.3. Exemples d'hébergements "cloud"

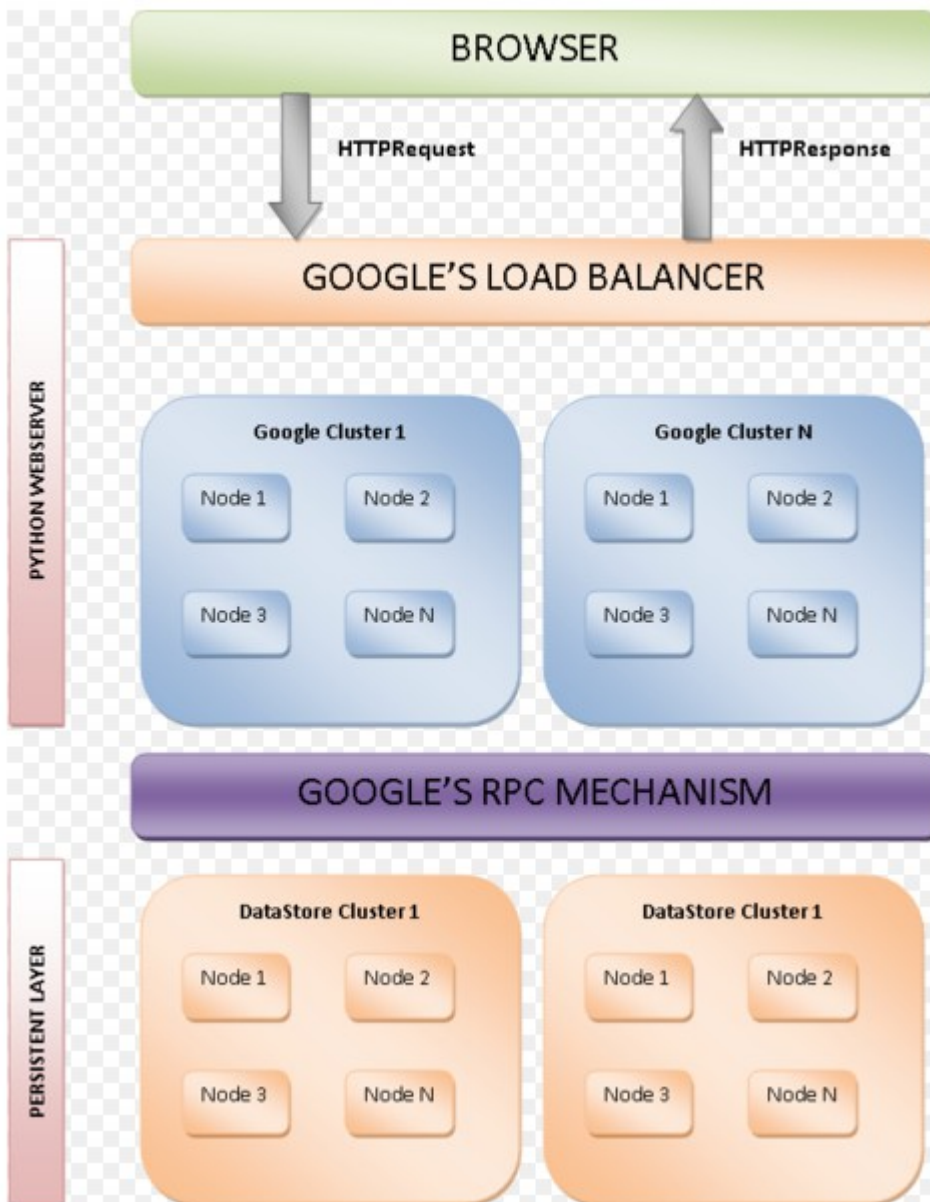
Gandi.net	Essentiellement Iaas , un peu Paas (php , node-js, ...)
Windows Azure (hébergement)	Microsoft
"CloudLayer / SoftLayer"	IBM

"Gandi.net"

Load balancing via "accélérateur web" :



"Google App Engine" :



ANNEXES

VI - Annexe - Api JAX-WS (pour SOAP) et CXF

1. Présentation de JAX-WS

JAX-WS signifie *Java Api for Xml Web Services*

L'api **JAX-WS** est une api utilisable à partir du jdk 1.5 et qui remplace maintenant l'ancienne api JAX-RPC . L'ancienne Api JAX-RPC n'est aujourd'hui intéressante que pour maintenir d'anciennes applications basées sur le jdk 1.4 .

L'api **JAX-WS** *ne fonctionne qu'avec le jdk 1.5 , 1.6 ou 1.7* et n'est donc utilisable qu'au sein des serveurs **JEE** récents (ex: *JBoss 4.2 , Tomcat 5.5 + Apache_CXF ,*).

Important : **JAX-WS est déjà intégré dans le jdk >=1.6** (pas de ".jar" à ajouter mais "update récent du jdk1.6 conseillé pour une version récente et optimisée/performante de JAX-WS) .

Dans le cas du *jdk1.5* , **JAX-WS s'ajoute en tant que ".jar"** d'une des implémentations disponibles (metro , *CXF* ,).

Les principales spécificités de **JAX-WS** (vis à vis de JAX-RPC) sont les suivantes:

- paramétrage par **annotations Java5** (ex: @WebService , @WebMethod)
- utilisation interne de **JAXB2**

NB:

- L'utilisation "**côté client**" de JAX-WS ne nécessite que le **jdk >=1.6** (ou des ".jar" additionnels pour le jdk 1.5)
- L'utilisation "**côté serveur**" de JAX-WS nécessite généralement une *prise en charge évoluée et intégrée* (ex : *EJB3* ou bien "*CxfServlet*" dans une *appli web Spring*).

2. Mise en oeuvre de JAX-WS (côté serveur)

La mise en oeuvre d'un service web côté serveur avec **JAX-WS** consiste essentiellement à utiliser les annotations @WebService , @WebParam , @WebMethod , @WebResult, au niveau du code source.

L'interface d'un service WEB nécessite simplement l'annotation @WebService et n'a pas absolument besoin d'hériter de java.rmi.Remote .

```
package calcul;
import javax.jws.WebService;

@WebService
public interface Calculator_SEI {
    //Calculator ou Calculator_SEI ou .... avec SEI = Service Endpoint Interface
    public int addition(int a,int b);
}
```

NB: les noms associés au service (*namespace, PortType, ServiceName*) sont par défaut basés sur les noms des éléments java (package , nom_interface et/ou classe d'implémentation) et peuvent

éventuellement être précisés/redéfinis via certains attributs facultatifs de l'annotation **@WebService**

NB: de façon à ce que les noms des paramètres des méthodes d'une interface java soient bien retranscrits dans un fichier WSDL, on peut les préciser via **@WebParam(name="paramName")**.

```
public int addition(int a,int b); ==> addition(xsd:int arg0,xsd:int arg1) dans WSDL
public int addition(@WebParam(name="a") int a, @WebParam(name="b") int b)
==> addition(xsd:int a ,xsd:int b) dans WSDL
```

L'annotation facultative **@WebResult** permet entre autre de spécifier le nom de la valeur de retour (dans SOAP et WSDL) . Par défaut le nom du résultat correspond à **"return"** .

NB: le sigle *SEI* signifie **Service EndPoint Interface** . il désigne simplement l'interface d'un service WEB et ne fait l'objet d'aucune convention de nom.

Au dessus de la classe d'implémentation , l'annotation **@WebService** peut éventuellement être sans argument si la classe n'implémente qu'une seule interface. Dans le cas où la classe implémente(ra) plusieurs interfaces, il faut préciser celle qui correspond à l'interface du Service Web:

```
@WebService (endpointInterface="package.Interface_SEI")
```

La classe d'implémentation d'un service Web peut éventuellement préciser via l'annotation **@WebMethod(exclude=true or false)** les méthodes qui seront vues du coté client et qui constituent le service web.

Si aucune annotation **@WebMethod** n'est mentionnée, alors toutes les méthodes publiques seront exposées.

```
package calcul;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

@WebService (endpointInterface="calcul.Calculator_SEI")
@SOAPBinding(style = SOAPBinding.Style.RPC)
public interface .... {

    @WebMethod(operationName="add")
    public int addition(int a, int b) .....
}
```

- **NB:** L'annotation *SOAPBinding* est facultative. Elle permet de choisir (quand c'est possible) un style SOAP (ex: Style.RPC ou Style.DOCUMENT) .
- Le mode par défaut **"document/literal"** convient **très bien** dans presque tous les cas .

NB: Ces annotations ne sont utiles que si elles sont interprétées par un serveur JEE5 ou 6 (ex: GlassFish de Sun ou Jboss 4.2 ou 5) ou bien par une technologie JAX-WS (telle que CXF) ajoutée à

un serveur J2EE (ex: Tomcat 5.5).

NB: Selon l'implémentation retenue de JAX-WS (metro , jboss-ws , CXF,) , les annotations sont acceptées ou pas lorsqu'elles sont directement placées au dessus de la classe d'implémentation.

Autres paramétrages importants (packages/namespaces):

- Lorsque l'interface et la classe d'implémentation d'un service WEB sont placées dans des packages java différents , ceci se traduit par différents "namespaces" xml dans le fichier WSDL. Préciser l'attribut "**targetNamespace**" dans l'annotation **@WebService** de l'interface et aussi dans l'annotation **@WebService** de la classe d'implémentation permet de bien contrôler la (ou les) valeurs attendues dans le WSDL .
- Certaines méthodes ont des paramètres (en entrée ou en sortie) qui correspondent à des classes de données (JavaBean/POJO) .
Placer **@XmlType(namespace="http://yyy/data")** permet de bien contrôler le namespace XML qui sera associé à ces classes de données.

Exemple :

```
@XmlType(namespace="http://entity.tp/")
@XmlRootElement(name="stat")
public class Stat {
    private int num_mois; //de 1 à 12
    private double ventes; //+get/set
    ...
}
```

```
@WebService(targetNamespace="http://minibank.myapp.tp/")
public interface GestionComptes {

    public Compte getCompteByNum(@WebParam(name="numCpt")long numCpt)
        throws MyServiceException;

    public List<Stat> getStats(@WebParam(name="annee")int annee);

    public void transferer(@WebParam(name="montant")double montant,
        @WebParam(name="numCptDeb")long numCptDeb,
        @WebParam(name="numCptCred")long numCptCred)
        throws MyServiceException;
}
```

```
@WebService(targetNamespace="http://impl.minibank.myapp.tp/",
    endpointInterface="tp.myapp.minibank.itf.domain.service.GestionComptes")
public class GestionComptesImpl implements GestionComptes {
    ...
}
```

namespace par défaut : "http://" + nom_package_java_à_1_envers + "/"

2.1. Implémentation sous forme d'EJB3 (pour serveur JEE5 tel que JBoss4.2 ou 5 , Glassfish de Sun , Geronimo 2 d'apache , Jonas d'OW2 ou ...)

- Combiner les annotations précédentes (@WebService ,) avec l'annotation **@Stateless** au niveau de la classe d'implémentation d'un EJB3 Session sans état.
- Déployer le tout selon les spécifications JEE (.jar , .ear) au sein d'un serveur d'application comportant un conteneur d'EJB3
- Repérer l'url du service WEB et de la description WSDL (**au cas par cas selon serveur**)
[ex: <http://localhost:8080/xyz/XXXBean?wsdl>]

2.2. Test sans serveur et wsgen (jdk 1.6)

NB: à des simples fins de "tests unitaires" , il est éventuellement possible de tester un service WEB en lançant une simple application java (non web) compilée avec le jdk 1.6 (*sans CXF*) qui comporte déjà en lui l'api "JAX-WS" et un mini mini conteneur Web appelé "Jetty" .

==> Ceci n'est valable que pour des petits tests (sans serveur) et nécessite tout de même l'invocation d'un script (.bat / .sh ou ant) qui lance la commande bin/**wsgen** du jdk 1.6 .

```
set JAVA_HOME=C:\.....\java\jdk1.6.....
set WKSP=D:\....\workspace
set PRJ=myProject
set SEI_Impl=basic.SimpleCalculatorImpl
"%JAVA_HOME%\bin\wsgen" -cp %WKSP%\%PRJ%\bin
                        -d %WKSP%\%PRJ%\bin          %SEI_Impl%
```

NB: ceci est complètement **inutile** (et même quelquefois source d'incompatibilités) avec une utilisation standard de *CXF* dans *Tomcat* ou bien des *EJB3* dans *Jboss* . ==> **CXF et les EJB3 déclenchent automatiquement un équivalent de wsgen** .

2.3. Lancement mini serveur sans Container Web pour Tests

[==> OK avec jdk >=1.6 ou bien CXF .]

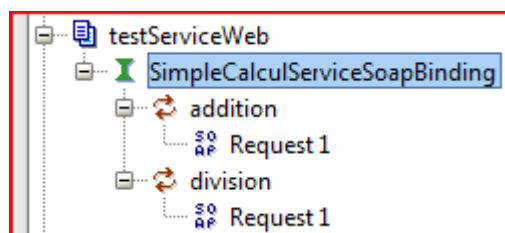
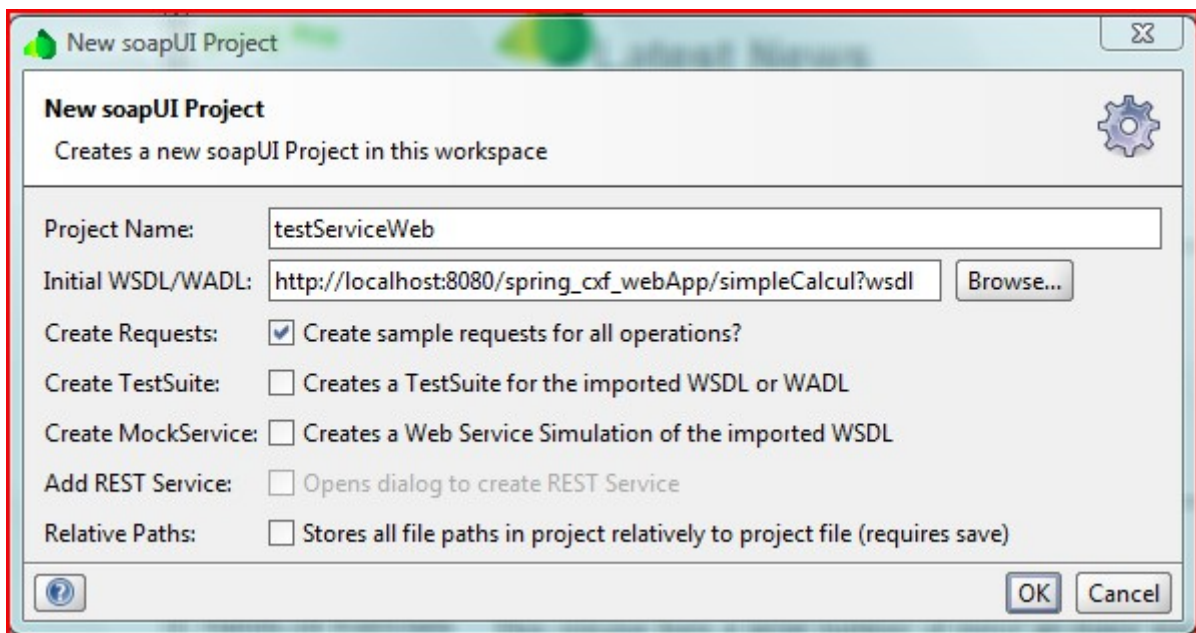
```
...
public class StandaloneServTestApp {
    protected StandaloneServTestApp() throws Exception {
        System.out.println("Starting Server");
        CalculateurImpl implementor = new CalculateurImpl();
        String address ="http://localhost:8080/myApp/services/calculateur";
        Endpoint.publish(address, implementor);
    }
    public static void main(String args[]) throws Exception {
        new StandaloneServTestApp();      System.out.println("Server ready...");
        Thread.sleep(15 * 60 * 1000);// 15 minutes
        System.out.println("Server exiting"); System.exit(0);
    }
}
```

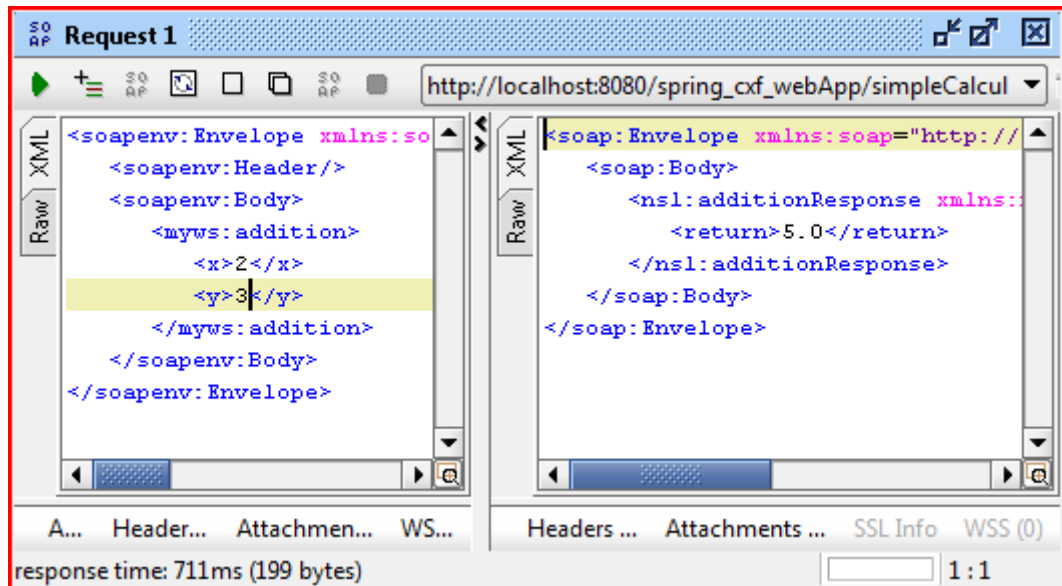
3. Tests via SOAPUI

==> pour tester rapidement un service Web , le plus simple est d'utiliser le programme utilitaire "soapui" (dont une version gratuite est disponible) .

Mode opératoire:

- Télécharger et installer soapui .
- Lancer le produit via le script ".bat" du sous répertoire bin.
- Nouveau projet (de test) avec nom à choisir.
- Préciser l'URL (se terminant généralement par *"?wsdl"*) de la description du service à invoquer.
- Sélectionner une des méthodes pour la tester en mode requête / réponse.
- Renseigner quelques valeurs au sein d'une requête.
- Déclencher l'invocation de la méthode via SOAP
- Observer la réponse (ou le message d'erreur)





Utilisation de JAX-WS coté client

3.1. Mode opératoire général (depuis WSDL)

1. générer (à partir du fichier WSDL) toutes les classes nécessaires au "*proxy JAX-WS*"
2. écrire le code client utilisant les classes générées:

```
package client;
import calcul.CalculatorSEI;
import calcul.CalculatorService;

public class MyWSClientApp {

    public static void main (String[] args) {
        try {
            CalculatorSEI calc = new CalculatorService().getCalculatorPort();

            int number1 = 10; int number2 = 20;
            System.out.printf ("Invoking addition(%d, %d)\n", number1,
                               number2);

            int result = calc.addition (number1, number2);
            System.out.printf ("The result of adding %d and %d is %d.\n\n",
                               number1, number2, result);
        } catch (Exception ex) {
            System.out.printf ("Caught Exception: %s\n", ex.getMessage ());
        }
    }
}
```

Le tout s'interprète très bien avec le **jdk >=1.6** (comportant déjà une implémentation de **JAX-WS**) ou bien avec un **jdk 1.5** augmenté de quelques librairies (ex: .jar de CXF).

3.2. Génération du proxy jax-ws avec wsimport du jdk >=1.6

```
set JAVA_HOME=C:\Prog\java\jdk\jdk1.6
cd /d "%~dp0"
"%JAVA_HOME%\bin\wsimport" [- ....] http://localhost:8080/...../serviceXY?wsdl
pause
```

sans l'option `-keep` ==> proxy au format compilé seulement

avec l'option `-keep` ==> code source du proxy également

l'option `-d` permet d'indiquer le répertoire *destination* (ou le proxy sera généré).

D'autres options existent (à lister via `-help`)

4. Redéfinition de l'URL d'un service à invoquer

4.1. Redéfinition de l'URL "SOAP"

Si l'on souhaite *invoquer un service Web localisé à une autre URL "SOAP" que celle qui est précisée dans le fichier WSDL* (ayant servi à générer le proxy d'appel) , on peut utiliser l'interface cachée "*BindingProvider*" de la façon suivante:

```
Calculateur calculateur = serviceCalculateur.getCalculateurServicePort();
javax.xml.ws.BindingProvider bp = (javax.xml.ws.BindingProvider) calculateur;
Map<String,Object> context = bp.getRequestContext();
context.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
            "http://localhost:1234/myWebApp/services/calculateur");
System.out.println(calculateur.getTva(200, 19.6));
```

4.2. Redéfinition de l'URL WSDL (et indirectement SOAP)

Il est souvent plus simple (et efficace) de redéfinir l'URL du fichier WSDL qui indirectement précise l'URL SOAP à utiliser :

exemple :

```
URL urlWsdL = new URL("http://localhost:8080/convertisseur-web/services/convertisseur?wsdl");
QName sQname = new QName("http://conversion/", "ConvertisseurImplService");
ConvertisseurImplService s = new ConvertisseurImplService(urlWsdL,sQname);
Convertisseur proxyConv = s.getConvertisseurImplPort(); ....
```

NB : En mode non dynamique (basé sur WSDL et wimport) , le fichier WSDL doit absolument être accessible lors de l'exécution (sinon exception et pas de connexion)

5. Client JAX-WS sans wsimport et directement basé sur l'interface java

Dans le cas (plus ou moins rares) où les cotés "client" et "serveur" sont tous les deux en Java, il est possible de se passer de la description WSDL et l'on peut générer un client d'appel (proxy/business delegate) en se basant directement sur l'interface java du service web (*SEI : Service EndPoint Interface*).

NB: L'interface java doit au minimum comporter @WebService (et souvent @WebParam)

5.1. avec le jdk 1.6 et un accès au wsdl (sans CXF ni Spring)

```
private static void testCalculeteurServiceWithoutWsImport() {

    QName SERVICE_NAME = new QName("http://myws/", "CalculeteurService");
    QName PORT_NAME = new QName("http://myws/", "CalculeteurServicePort");

    // en précisant une URL WSDL connue et accessible
    String wdlUrl =
        "http://localhost:8080/spring_cxf_webApp/services/calculateur?wsdl";

    URL wsdlDocumentLocation=null;
    try {wsdlDocumentLocation = new URL(wdlUrl);
        } catch (MalformedURLException e) { e.printStackTrace();}

    //avec import javax.xml.ws.Service;
    Service service = Service.create(wsdlDocumentLocation, SERVICE_NAME);

    Calculeteur caculeteurWSPProxy = (Calculeteur)
        service.getPort(PORT_NAME, Calculeteur.class);

    double tauxTvaReduitPct =
        caculeteurWSPProxy.getTauxTva("TauxReduit");
    System.out.println("TauxReduit=" + tauxTvaReduitPct);
}
```

5.2. avec le jdk 1.6 et quelques ".jar" de CXF

```
private static void testCalculeteurServiceWithCxfAndWithoutWsImport() {

    QName SERVICE_NAME = new QName("http://myws/", "CalculeteurService");
    QName PORT_NAME = new QName("http://myws/", "CalculeteurServicePort");

    Service service = Service.create(SERVICE_NAME); //javax.xml.ws.Service
    // Endpoint Address
    String endpointAddress =
        "http://localhost:8080/spring_cxf_webApp/services/calculateur";

    // Add a port to the Service , javax.xml.ws.soap.SOAPBinding
    service.addPort(PORT_NAME, SOAPBinding.SOAP11HTTP_BINDING,
        endpointAddress);

    Calculeteur caculeteurWSPProxy = (Calculeteur)
        service.getPort(PORT_NAME, Calculeteur.class);

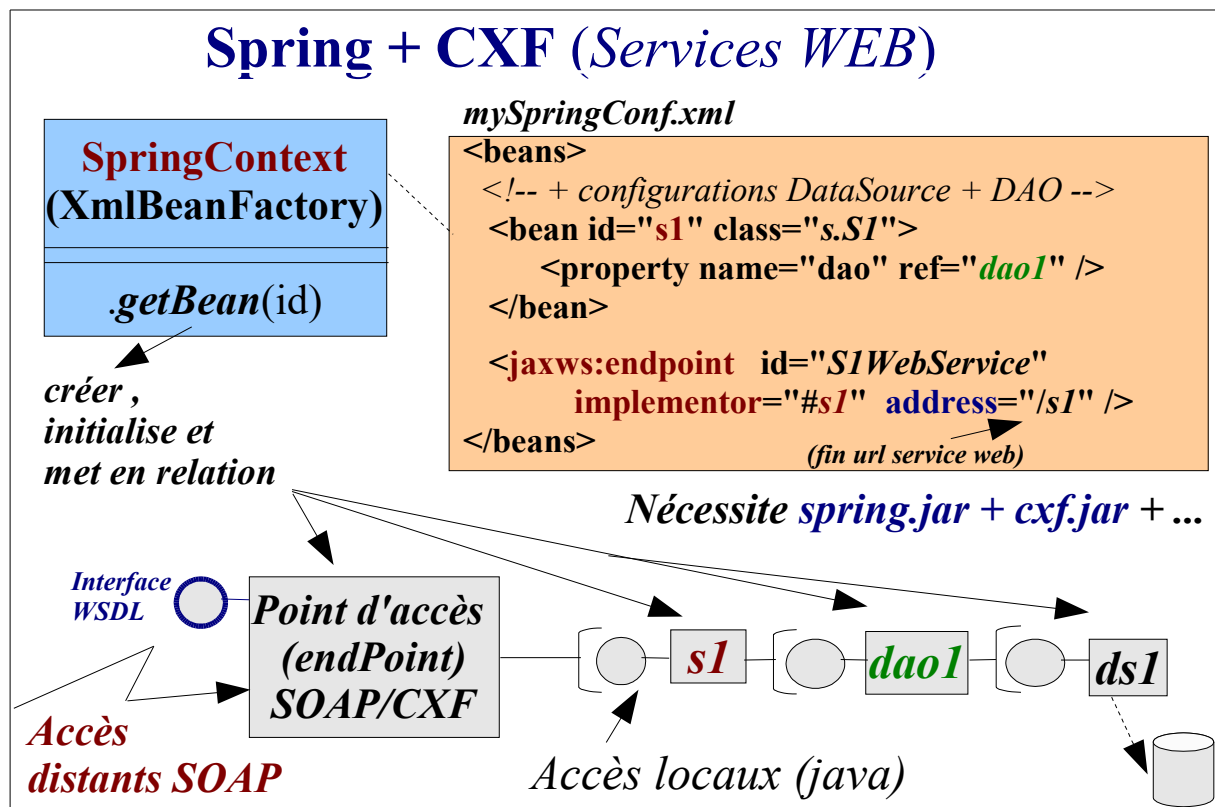
    double tauxTvaReduitPct =
        caculeteurWSPProxy.getTauxTva("TauxReduit");
    System.out.println("TauxReduit=" + tauxTvaReduitPct);
}
```


6. CXF

6.1. Présentation de CXF (apache)

- **CXF** est une technologie facilitant le développement et la construction de **Web Services** en se basant sur l'API **JAX-WS**.
- Le framework **CXF** (de la fondation Apache) permet de mettre en place des **points d'accès (SOAP)** vers les **services métiers** d'une application.
- Les points d'accès (endpoints) sont pris en charge par le servlet prédéfini "**CxfServlet**" qu'il suffit de paramétrer et d'intégrer dans une application java web (*sans nécessiter d'EJB*).

L'intégration de CXF avec Spring est simple :



- Une description WSDL du service web sera alors automatiquement générée et le service métier Spring sera non seulement accessible localement en java mais sera accessible à distance en étant vu comme un service Web que l'on peut invoquer par une URL http .
- Il faudra penser à déclarer le servlet de CXF dans WEB-INF/web.xml et placer tous les ".jar" de CXF dans WEB-INF/lib .
- Le paramétrage fin du service Web s'effectue en ajoutant les annotations standards de JAX-WS (@WebService , @WebParam , ...) dans l'interface et la classe d'implémentation du service métier .

Remarque importante: CXF (et CxfServlet) peut à la fois prendre en charge *SOAP* (via JAX-WS) et aussi *REST* (via JAX-RS) . CXF peut également s'utiliser sans Spring.

6.2. implémentation avec CXF (et Spring) au sein d'une application Web (pour Tomcat 5.5, 6 ou 7)

- Rapatrier les ".jar" de CXF au sein du répertoire **WEB-INF/lib** d'un projet Web. (ou bien configurer convenablement les dépendances "maven").

- Ajouter la configuration suivante au sein de **WEB-INF/web.xml**:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>WEB-INF/classes/beans.xml</param-value> <!-- ou .... -->
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
</listener-class>
</listener>
<servlet>
    <servlet-name>CXFServlet</servlet-name>
    <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>CXFServlet</servlet-name>
    <url-pattern>/services/*</url-pattern> <!-- ou autre que services/* -->
</servlet-mapping>
```

- Ajouter la configuration Spring suivante dans WEB-INF ou bien dans
[src ou src/main/resources ==> WEB-INF/classes] :

beans.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">
    <import resource="classpath:META-INF/cxf/cxf.xml" />
    <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
    <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />
    <jaxws:endpoint id="calculateurService"
        implementor="service.CalculateurService" address="/Calculateur" />
</beans>
```

variante plus pratique si plusieurs niveaux d'injections (ex: DAO , DataSource, ...):

```
<bean id="calculateurService_impl" class="service.CalculateurService" >
    <property name="...." ref="...." />
</bean>
```



```
<jaxws:endpoint
    id="calculateurService"
    implementor="#calculateurService_impl"
    address="/Calculateur" />
```

NB:

- La syntaxe est "**#idComposantSpring**"
- Dans l'exemple ci dessus "*service.CalculateurService*" correspond à la classe d'implémentation du service Web (avec @WebService) et "*/Calculateur*" est la partie finale de l'url menant au service web pris en charge par Spring+CXF .
- Après avoir déployé l'application et démarré le serveur d'application (ex: run as/ run on serveur),il suffit de préciser une URL du type <http://localhost:8080/myWebApp/services> pour obtenir la liste des services Web pris en charge par CXF. En suivant ensuite les liens hypertextes on arrive alors à la description WSDL .

==> c'est tout simple , modulaire via Spring et ça fonctionne avec un simple Tomcat !!!

Remarque :

En ajoutant le paramétrage `bindingUri="http://www.w3.org/2003/05/soap/bindings/HTTP/"` on peut générer un deuxième "endpoint" en version "**soap 1.2**" plutôt que soap 1.1 :

```
<jaxws:endpoint id="serviceCalculateur_soap12_endPoint"
    bindingUri="http://www.w3.org/2003/05/soap/bindings/HTTP/"
    implementor="#calculateurService_impl" address="/calculateur_soap12">
    <!-- version SOAP 1.2 -->
</jaxws:endpoint>
```

6.3. Client JAX-WS sans wsimport avec CXF et Spring

Rappel: Dans le cas (plus ou moins rares) où les cotés "client" et "serveur" sont tous les deux en Java , il est possible de se passer de la description WSDL et l'on peut générer un client d'appel (proxy/business delegate) en se basant directement sur l'interface java du service web (*SEI : Service EndPoint Interface*)

La configuration Spring suivante permet de générer automatiquement un "business_delegate" (dont l'id est ici "client" et qui est basé sur l'interface "service.Calculateur" du service Web).

Ce "business delegate" délèguera automatiquement les appels au service Web distant dont l'url est précisé par la propriété "address".

src/bean.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jaxws="http://cxf.apache.org/jaxws"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```

http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schema/jaxws.xsd">
<bean id="clientFactory" class="org.apache.cxf.jaxws.JaxWsProxyFactoryBean">
  <property name="serviceClass" value="service.Calculateur" /> <!-- interface_SEI -->
  <property name="address" value="http://localhost:8080/serveur\_cxf/Calculateur" />
</bean>
<bean id="client" class="service.Calculateur" factory-bean="clientFactory"
  factory-method="create" />
</beans>

```

autre possibilité (syntaxe plus directe) :

```

<jaxws:client id="client"
  serviceClass="service.Calculateur"
  xmlns:tp="http://service.tp/"
  serviceName="tp:CalculateurImplService"
  endpointName="tp:CalculateurServiceEndpoint"
  address="http://localhost:8080/serveur\_cxf/Calculateur" />
<!-- avec serviceClass="package.Interface_SEI" et la possibilité d'ajouter des
intercepteurs -->

```

il ne reste alors plus qu'à utiliser ce "business_delegate" au sein du code client basé sur Spring:

```

import org.springframework.context.support.ClassPathXmlApplicationContext;
...
public class WsTestApp {

    public static void main(String[] args) {
        ClassPathXmlApplicationContext context = new
            ClassPathXmlApplicationContext(new String[] { "beans.xml" });

        Calculateur calculateur = (Calculateur) context.getBean("client");
        System.out.println(calculateur.getTva(200, 19.6));
    }
}

```

NB: il faut penser à rapatrier les ".jar" de cxf et le code source de l'interface SEI (ici service.Calculateur) .

6.4. avec CXF sans Spring

→ voir fin du chapitre "essentiel jax-ws" .

VII - Annexe – WS REST en java

1. API java pour REST (JAX-RS)

JAX-RS est une API java assez récente (depuis les spécifications **JEE 6**) qui permet de mettre en œuvre des services REST en JAVA :

Une classe java avec annotations JAX-RS sera exposée comme web service REST.

JAX-RS se limite à l'implémentation serveur, la spécification ne propose rien du côté client (cependant, certaines implémentations (ex : Jersey) offrent un support du côté client).

Pour implémenter les services REST, on utilise principalement les annotations JAX-RS suivantes:

- **@Path** : définit le chemin (fin d'URL) de la ressource. Cette annotation se place sur la classe et/ou sur la méthode implémentant le service.
- **@GET, @PUT, @POST, @DELETE** : définit le mode HTTP (selon logique CRUD)
- **@Produces** spécifie le ou les Types MIME de la réponse du service
- **@Consumes** : spécifie le ou les Types MIME acceptés en entrée du service

Les principales implémentations de JAX-RS sont :

- **Jersey**, implémentation de référence de SUN (bien : simple/léger et efficace + côté client)
- **Resteasy**, l'implémentation interne à jboss (bien)
- **CXF** (gérant à la fois "SOAP" et "REST", remplacer si besoin la sous couche "jettison" par "jackson" pour mieux générer du "json")
- **Restlet** (???)

1.1. Code typique d'une classe java (avec annotations de JAX-RS)

Exemple d'une classe de donnée (ex : DTO) pouvant être transformée en XML:

```
package pojo;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "user") /* pour préciser balise xml englobante et pour jaxb2 */
public class User {
    private Integer id; // +get/set
    private String name; // +get/set

    @Override
    public String toString() {
        return String.format("{id=%s,name=%s}", id, name);
    }
}
```

Classe d'implémentation d'un service REST:

```
package service;

import java.util.HashMap; import java.util.Map;
import javax.ws.rs.GET;      import javax.ws.rs.Path;
import javax.ws.rs.PathParam; import javax.ws.rs.QueryParam;
```

```

import javax.ws.rs.Produces;    import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.Status;
import pojo.User;

@Path("myservice") //service gérant des utilisateurs
//@Produces("text/xml")
@Produces("application/json") //par défaut pour toutes les méthodes de la classe
//+ éventuel @Named selon contexte (Spring ou CDI/JEE6 ou ...)
public class ServiceImpl /* implements ServiceDefn */{ //pas d'interface obligatoire

    private static Map<Integer,User> users = new HashMap<Integer,User>();

    static { //jeux de données (pour simulation de données en base)
        users.put(1, new User(1, "foo"));    users.put(2, new User(2, "bar"));
    }

    // + éventuel injection d'un service local interne pour déléguer les appels :
    // @Autowired ou @Inject ou @EJB ou ....
    // private ServiceInterne serviceInterne ;

    @GET
    @Path("users")
    // pour URL = http://localhost:8080/mywebapp/services/rest/myservice/users
    public Collection<User> getUsers() {
        return users.values();
    }

    @GET
    @Path("users/{id}")
    // pour URL = http://localhost:8080/mywebapp/services/rest/myservice/users/2
    public User getUser(@PathParam("id") Integer id) {
        return users.get(id);
    }

    @GET
    @Path("users")
    // pour URL = http://localhost:8080/mywebapp/services/rest/myservice/users?name=foo
    // et quelquefois ...?p1=val1&p2=val2&p3=val3
    public User getUserByCriteria(@QueryParam("name") String name) {
        if(name!=null) .... ;
        return .... ;
    }

    @GET
    @Path("bad")
    public Response getBadRequest() {
        return Response.status(Status.BAD_REQUEST).build();
        //le comble d'un service est de ne rendre aucun service !!!! (bad: pas bien: is no good) !!!
    }

    @POST // (REST recommande fortement POST pour des ajouts )
    @Path("users")
    // pour action URL = http://localhost:8080/mywebapp/services/rest/myservice/users
    // dans form avec <input name="id" /> et <input name="name" /> et method="POST"
    public Response addNewUser(@FormParam("id") Integer id,@FormParam("name") String name) {
        users.put(id,new User(id,name));
        return Response.status(Status.OK).build();
        //une autre variante du code pourrait retourner le "User" avec une clef primaire quelquefois auto incrémentée
        // c'est ce qu'attend par défaut "Angular-Js" avec une logique @POST pour un "saveOrUpdate"
    }
}

```