

Web-Services

- 1) présentation SOA , WS - diapo 3
- 2) SOAP , WSDL - diapo 23
- 3) Api JAX-WS – diapo 44
- 4) variantes jax-ws , wsdl , ... - diapo 53
- 5) approfondissements jax-ws , CXF - diapo 72
- 6) intercepteurs et sécurité ws - diapo 86
- 7) Web Services REST - diapo 103
- 8) Api JAX-RS - diapo 113

Orchestration de services (bpmn, bpel)

1) orchestration , données pivots (concepts)

- diapo 133

2) orchestration BPEL – diapo 149

3) orchestration via Activiti (bpmn2 + java)

– diapo 177

4) UserTask – diapo 194

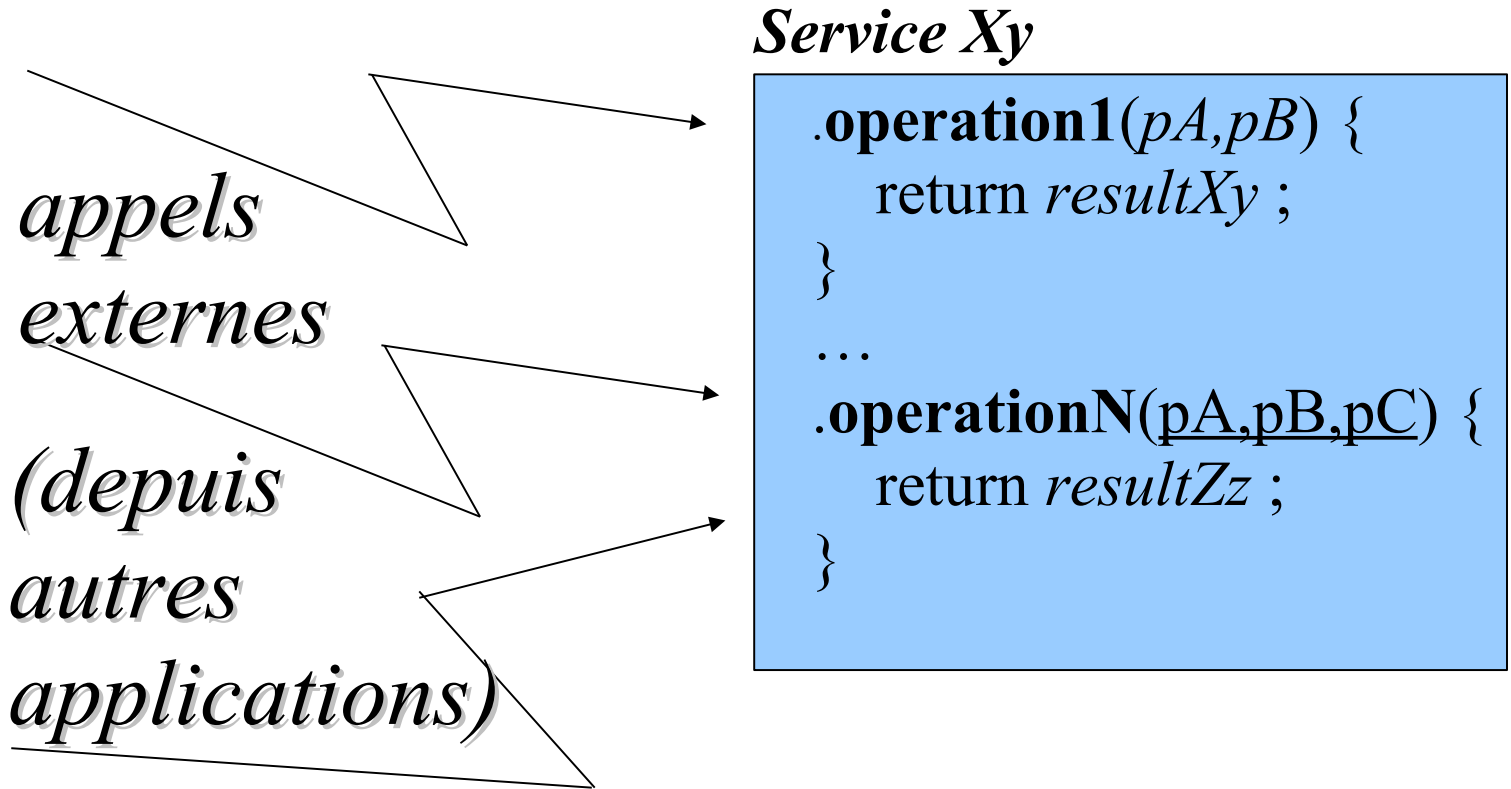
Présentation "ESB" - diapo 202

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

Grands traits de l'architecture SOA:

- Interopérabilité entre **langages différents** (*c++ , java, php , js , ...*)
- découplage entre invocation et implémentation (avec éventuels *intermédiaires reconfigurables* [adaptateur, ESB, ...])
→ Système informatique **souple/agile** .
- **Services de "haut niveau"** appelant des **sous services intermédiaires ou élémentaires** pour partager des données , des traitements , règles métiers , ...
- **Intégration**/adaptation de services rendus par des applications externes (de support, ...).

Service = paquet de "fonctions" accessibles



RPC = Remote Procedure Call

Chaque opération à son "contrat fonctionnel"

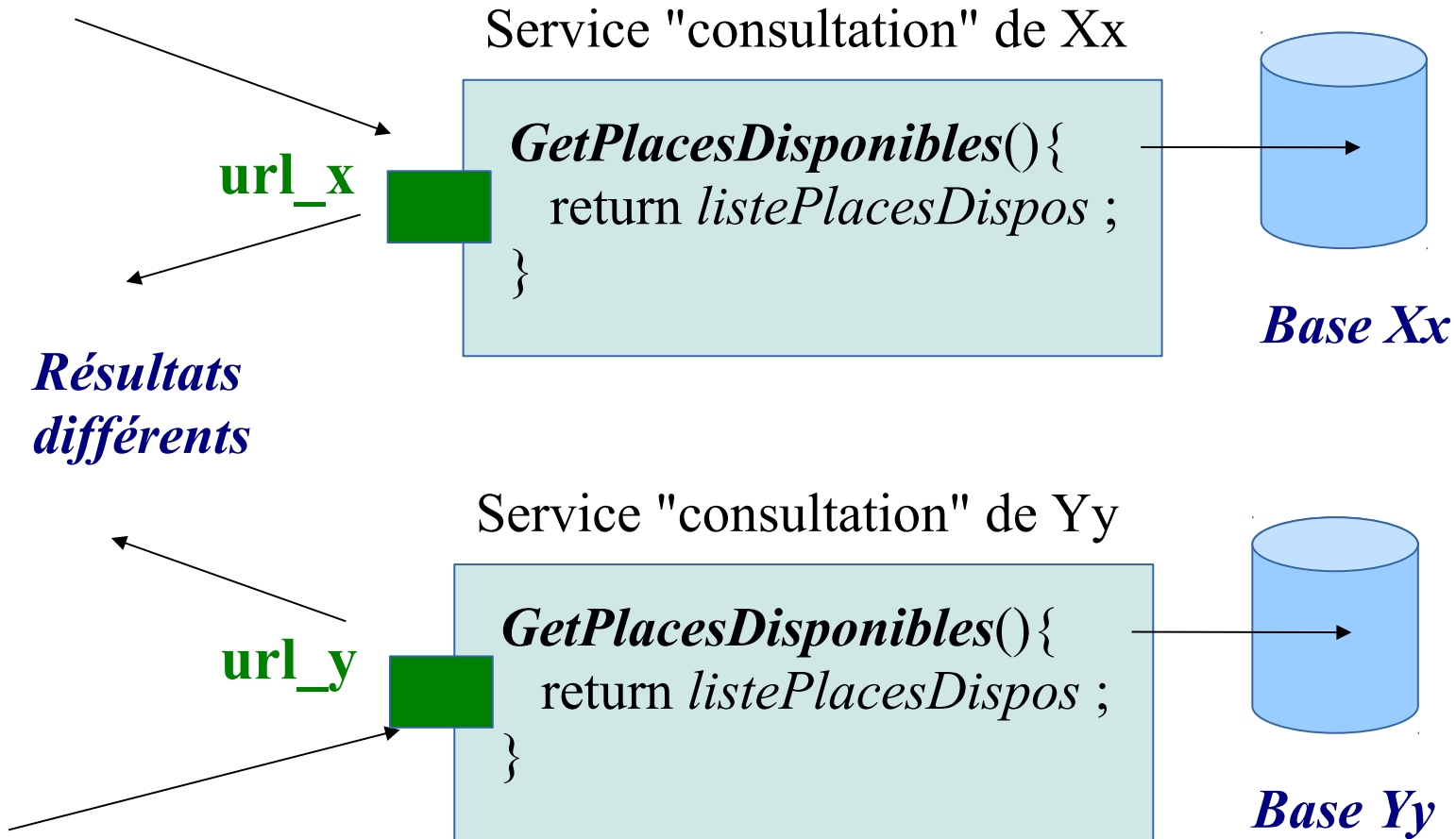
Requête (avec paramètres
et types significatifs) →

```
.operationXy(pA,pB) {  
    return resultXy ;  
}
```

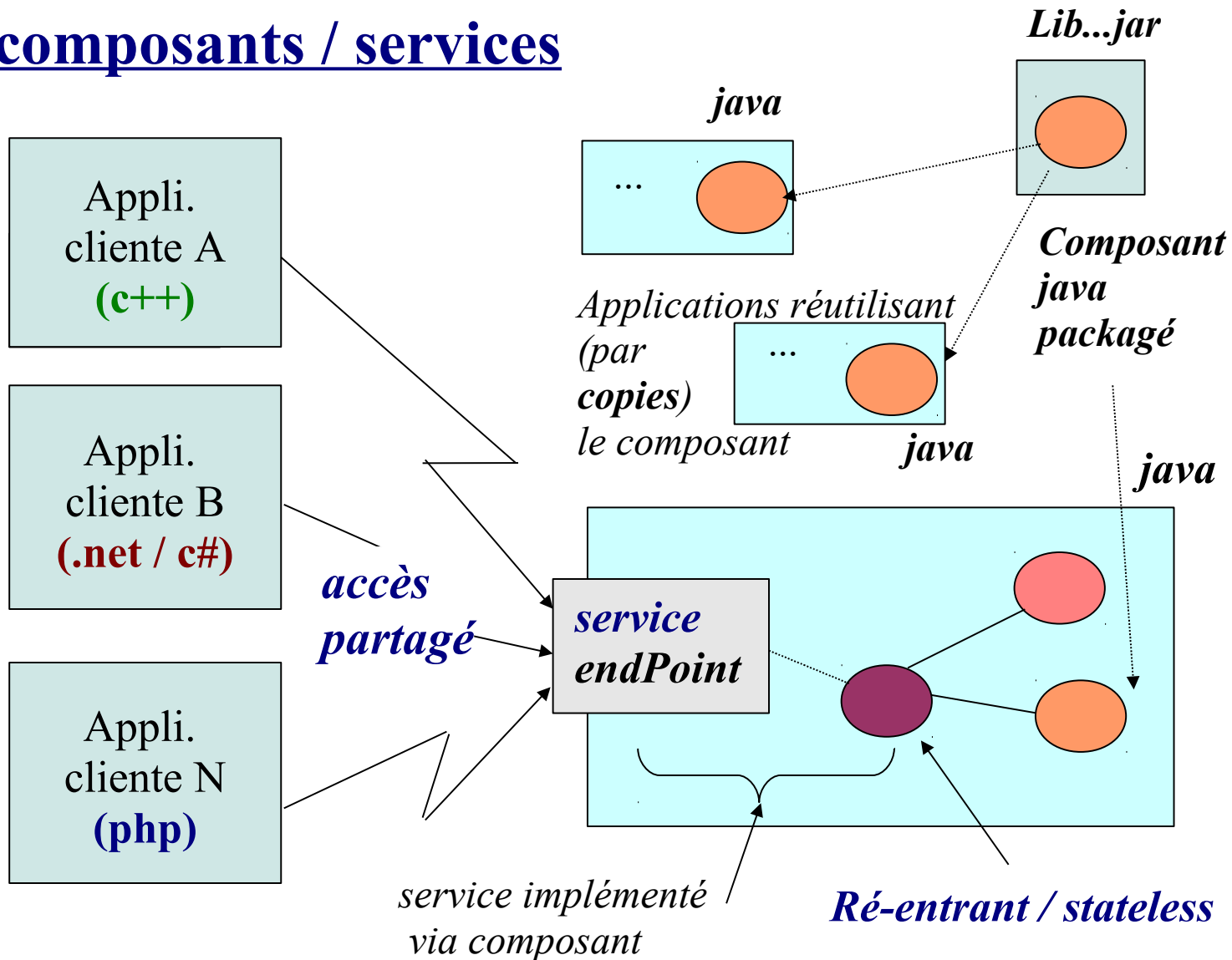
← **Réponse** (ou *exception*)
avec type et sémantique précis

Exemple : l'opération "*addition(a , b)*" attend en entrée les 2 opérandes à additionner et retourne le résultat de l'addition [*a , b et le résultat sont des nombres réels*].

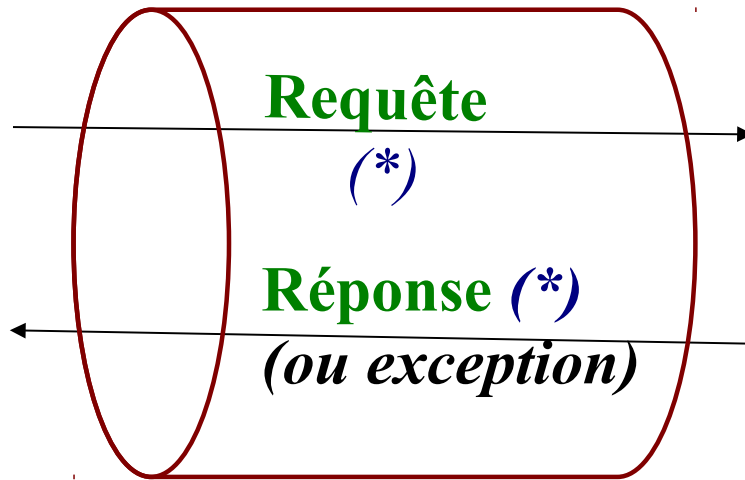
2 instances de services avec même(s) opération(s)



composants / services



Service "Web" = service mise en œuvre
avec technologie(s) "web" (*http* et/ou *xml*)



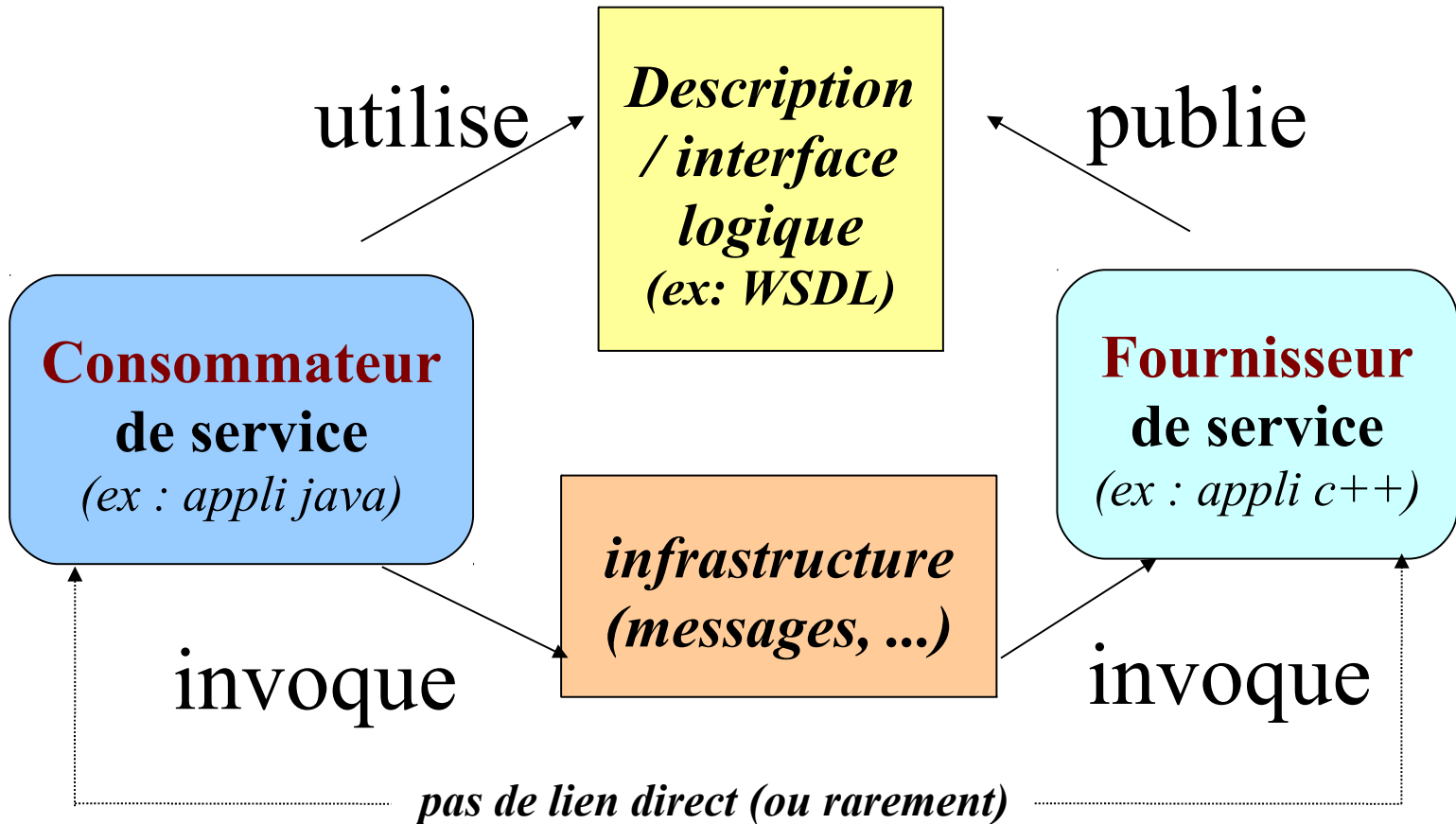
HTTP ou ...

() encodage
en XML ou JSON ou ...*

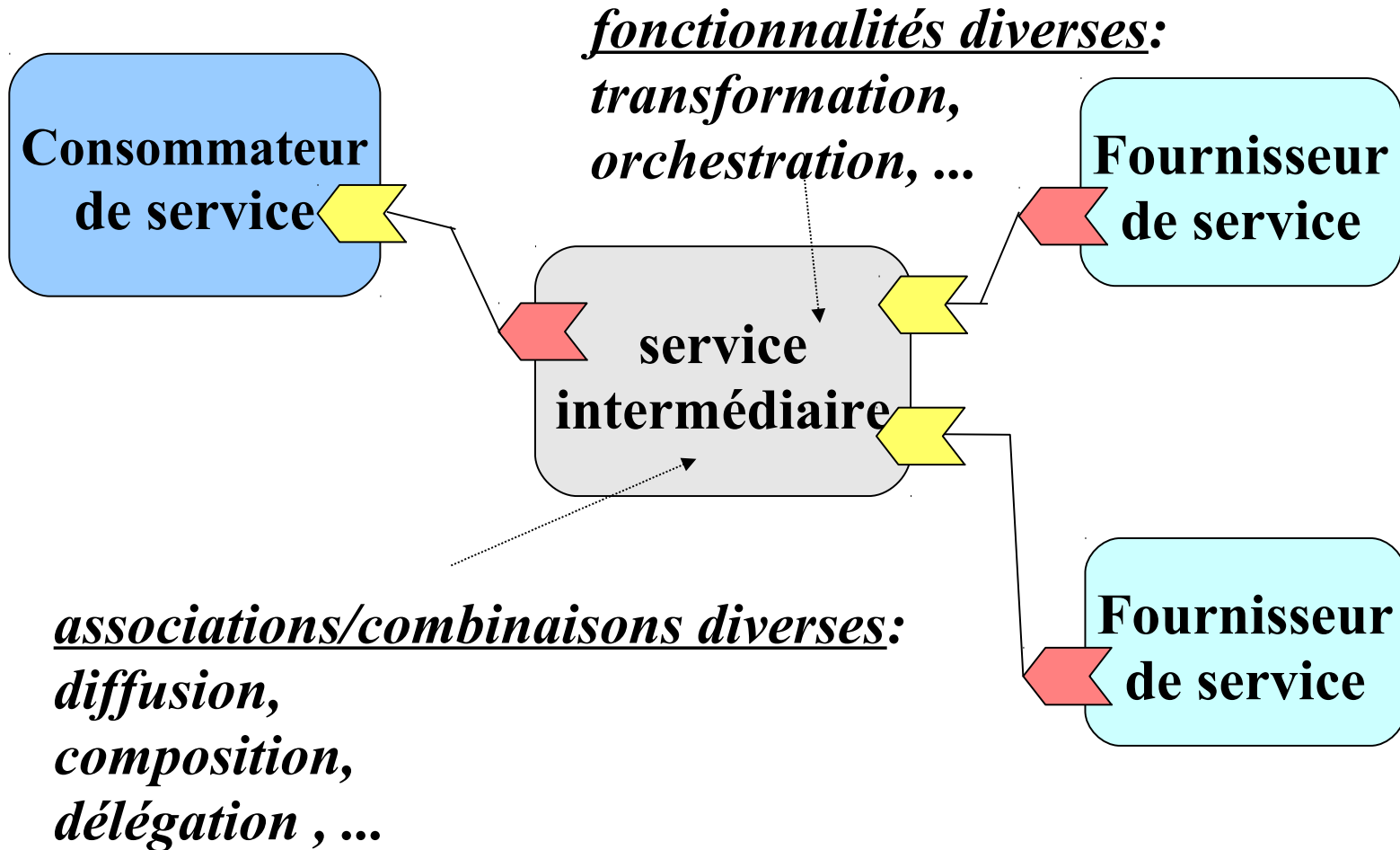
Point d'accès
(endPoint)
Côté "serveur"
avec **URL précise**

```
.operationXy(pA,pB) {  
    return resultXy ;  
}  
.operationZzz() { ... }
```

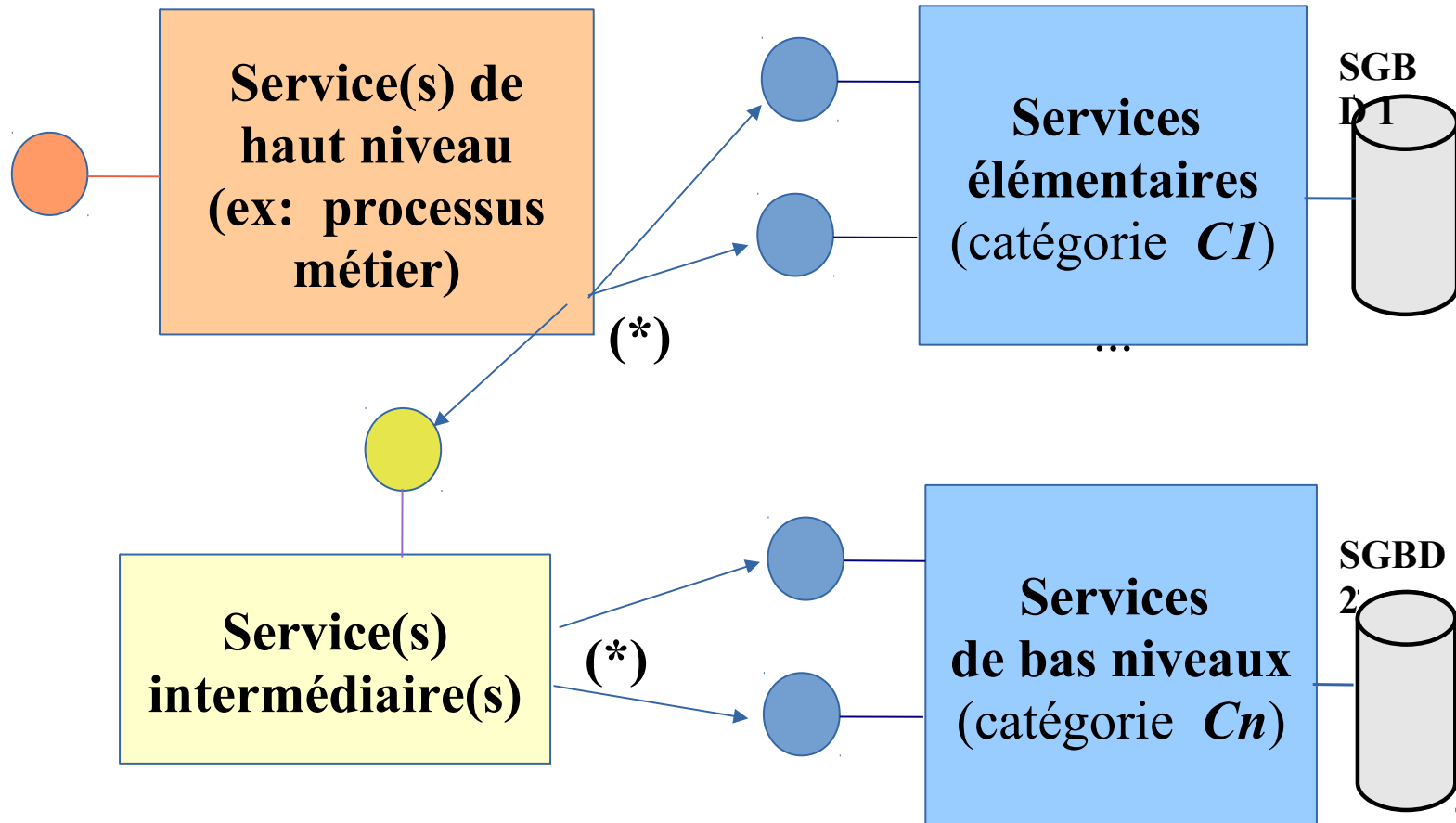

SOA : Combinaison de services sur le mode
"fournisseurs/consommateurs volontairement découplés"



SOA : Services intermédiaires = fournisseurs et consommateurs.

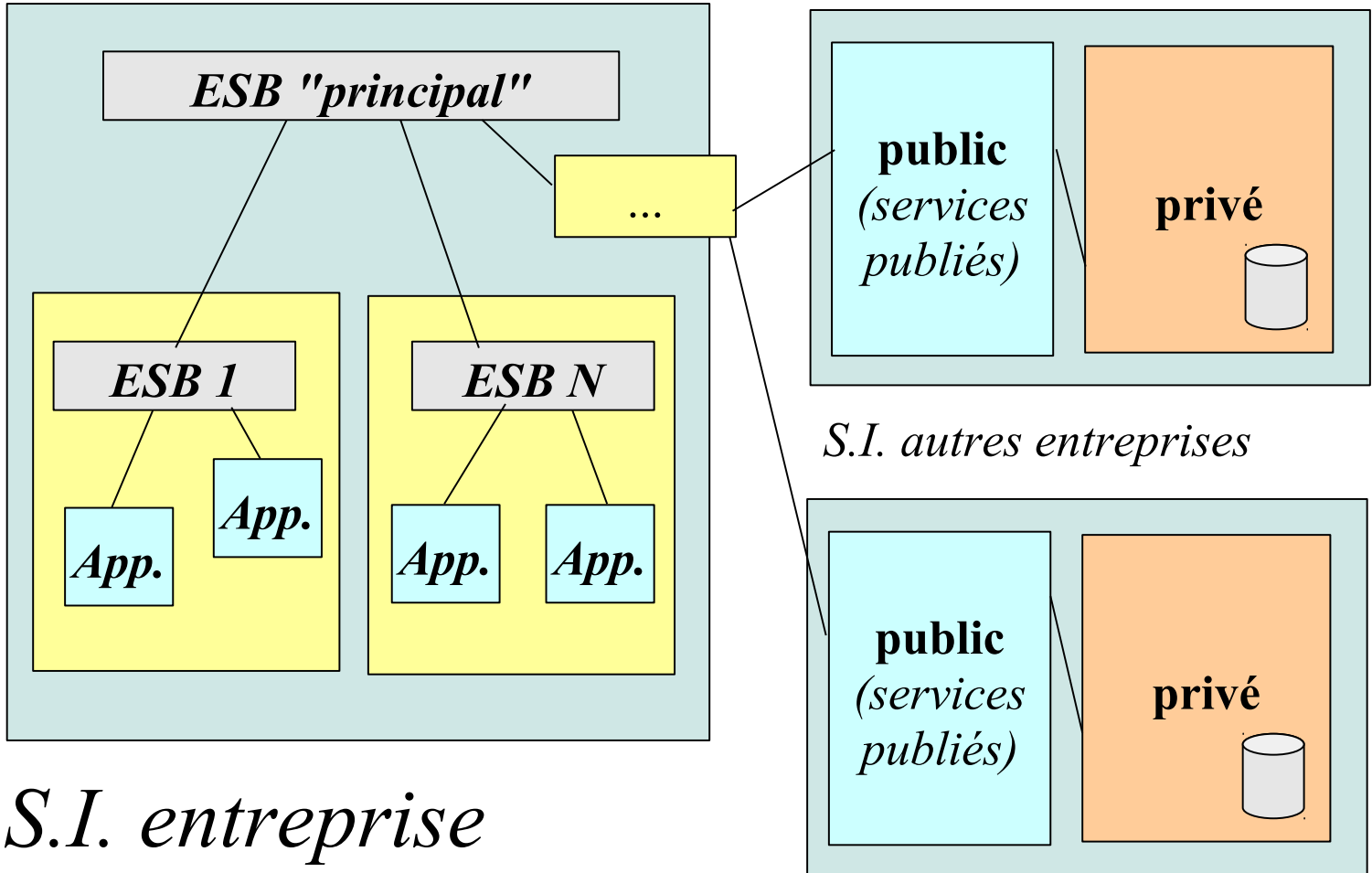


Liens entre Services



(*) **Orchestration/combinaison** potentielle de services .

Très grande portée de SOA



Principales technologies SOA

Services (web ou ...)

SOAP

WSDL

WS-Security ,
WS-* (ext.)

REST/HTTP

Annuaire d'URL
(UDDI,...)

Intermédiaire/ Adaptateur

E.S.B.

(Adaptateur,
transformateur,
intercepteur, filtre)
java ou ... , eip

Routage
(flow, camel,
...)

endpoint
et
connecteurs

Orchestration

BPMN
(processus)

ou

BPEL

JBpm
ou
Activiti

Java , ...

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

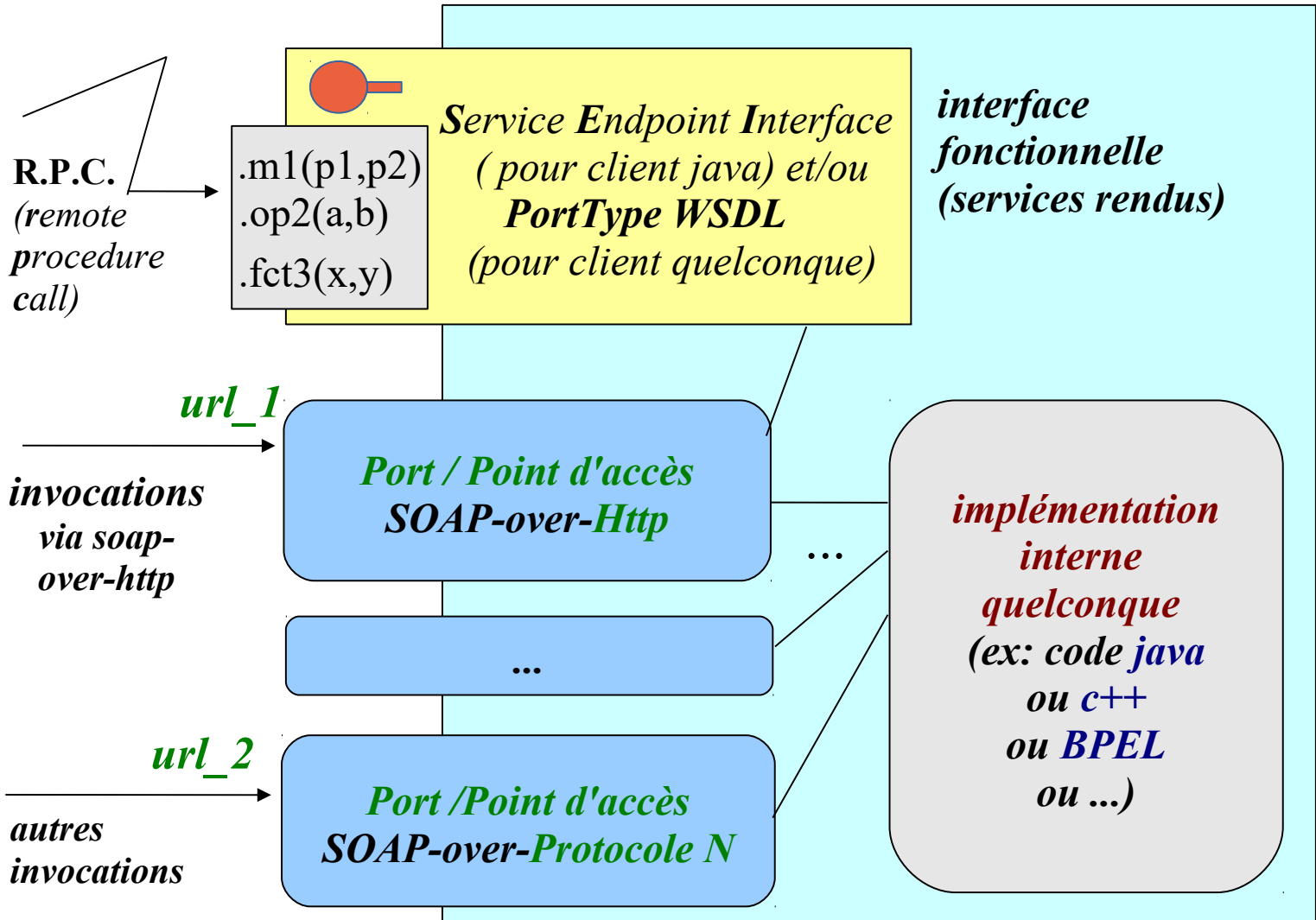
WS- (SOAP / XML)*

- "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 (*appels méthodes*) , **description WSDL**
- **Plutôt orienté Middleware SOA (arrière plan)**

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 IHM Web/Web2 (avant plan)**

Service Web "SOAP" (avant tout RPC en Xml)



Points clefs des Web services "SOAP"

Le format "xml rigoureux" des requêtes/réponses (définis par ".xsd" , ".wsdl") permet de retraiter sans aucune ambiguïté les messages "soap" au niveau certains services intermédiaires (dans ESB ou ...). Certains **automatismes génériques** sont **applicables** .

Fortement typés (xsd:string , xsd:double) les web-services "soap" conviennent très bien à des appels et implémentations au sein de **langages fortement typés** (ex : "c++" , "c#" , "java" , "...").

A l'inverse , des **langages faiblement typés** tels que "php" ou "js" sont **moins appropriés pour soap** (appels cependant faisables)

La **relative complexité et la verbosité "xml" des messages "soap"** engendrent des **appels moins immédiats** (*mode http "post" avec enveloppe à préparer*) et des **performances moyennes**.

Soap peut être utilisé en mode "envoi de document" mais c'est rare.

Les messages "soap" peuvent être véhiculés par "jms" mais c'est rare.

Service Web "REST" avec modes HTTP

*Url (avec
variantes)*

***Port / Point d'accès
Http (lié à début url)***

***Switch selon
* modes HTTP***

- POST
- GET
- PUT
- DELETE

**** fin url
(fin *path*,
param http)***

***implémentation
interne
quelconque
(ex: code *java*
ou *js* ou *php*)***

...

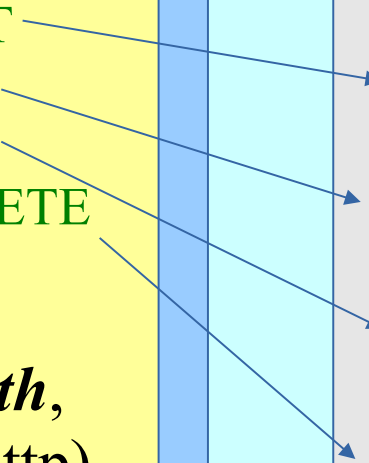
.createXxx()

.findXxxByZz()

.updateXxx()

.deleteXxx()

***invocations
via *http****



Points clefs des Web services "REST"

Retournant des données dans un format quelconque ("XML" , "JSON" et éventuellement "txt" ou "html") les web-services "REST" offrent des **résultats qui nécessitent généralement peu de re-traitements pour être mis en forme** au sein d'une IHM web.

Le format "**au cas par cas**" des données retournées par les services REST permet peu d'automatisme(s) sur les niveaux intermédiaires.

Souvent associés au format "JSON" les web-services "**REST**" **conviennent parfaitement** à des appels (ou implémentations) au sein du **langage javascript** .

La **relative simplicité des URLs d'invocation des services "REST"** permet des **appels plus immédiats** (*un simple href="..." suffit en mode **GET** pour les recherches de données*) .

La **compacité/simplicité des messages "JSON" souvent associés à "REST"** permet d'obtenir **d'assez bonnes performances** .

Protocoles et normes associés au web-services

SOAP = **S**imple **O**bject **A**ccess **P**rotocol (basé sur Xml)

WSDL = **W**eb **S**ervice **D**escription **L**angage (basé sur Xsd)

→ **WSDL** décrit la structure d'un service **SOAP**.

D'origine "IBM+Microsoft", les services soap sont maintenant associés à tout un tas de normes "**WS-...**" du **W3C**.

Ex: **WS-Security** normalise l'**entête Soap** du point de vue sécurité.
Beaucoup de normes proviennent de l'organisme "**OASIS**".

REST = **R**epresentational **S**tate **T**ransfert

= simples **conventions** sur l'usage de **HTTP** pour Web-services

UDDI = **U**niversal **D**escription **D**iscovery and **I**ntegration

correspond à une structure normalisée d'annuaire de services.

Très/trop complexe, UDDI (facultatif) est rarement utilisé.

Tout ceci peut se mettre en œuvre dans différents langages de programmation (c++ , java , c# , php , js , ...).

Principales API associées au web-services

Langage **C++** ---> **gSoap** , ...

Langage **C#** ---> infrastructure **".net"** et **"wcf"**

Langage **Java** --->

Ancienne API **"JAX-RPC"** (pour *jdk* ≤ 1.4)

Nouvelles API **"JAX-WS"** et **"JAX-RS"** (pour *jdk* ≥ 1.6)

L'api **"JAX-WS"** (Java Api for Xml Web-Services) se paramètre avec des annotations java (**@WebService** , **@WebParam** ,) et utilise en interne JAXB pour gérer les correspondances java <---> xml .

L'api **"JAX-RS"** (pour les Web Service **REST**) est assez récente (depuis JEE 6) et se paramètre via des annotations (**@Produces** , **@GET** , **@POST** ,) qui permettent de préciser les formats des données (XML , JSON ,) et les paramétrages HTTP (fin d'URL ,) .

Principales implémentations des spécifications "JAX-S"

La technologie Axis 1 correspond à une implémentation de l'ancienne api "JAX-RPC" (pour jdk <= 1.4). Axis 2 existe mais n'est pas normalisé.

*La technologie "**CXF**" (de la fondation open source "Apache") couvre à la fois les spécifications JAX-WS (pour WS "soap/xml") et les spécifications JAX-RS (pour WS "REST/HTTP") .*

*"**CXF**" s'intègre très facilement dans le framework **spring** et est souvent utilisé au sein des serveurs d'applications JEE (ex Jboss 7 avec EJB) .*

*D'autres technologies existent mais "**CXF**" est à considérer comme la technologie de référence .*

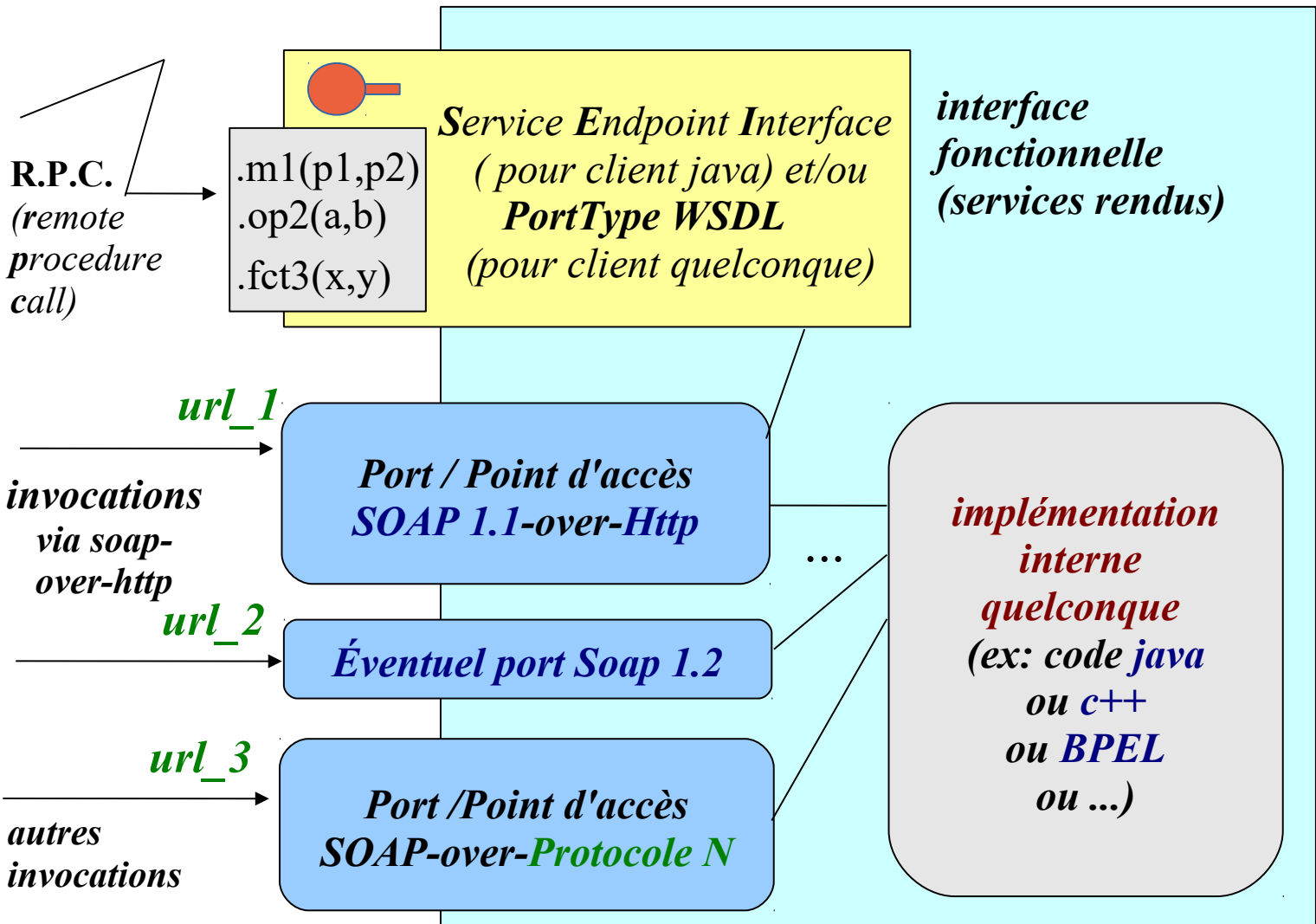
*Pour un besoin de type "service REST uniquement" , on pourra penser à la technologie "**jersey**" qui est très bien pour implémenter JAX-RS .*

Autres API "web services" secondaires

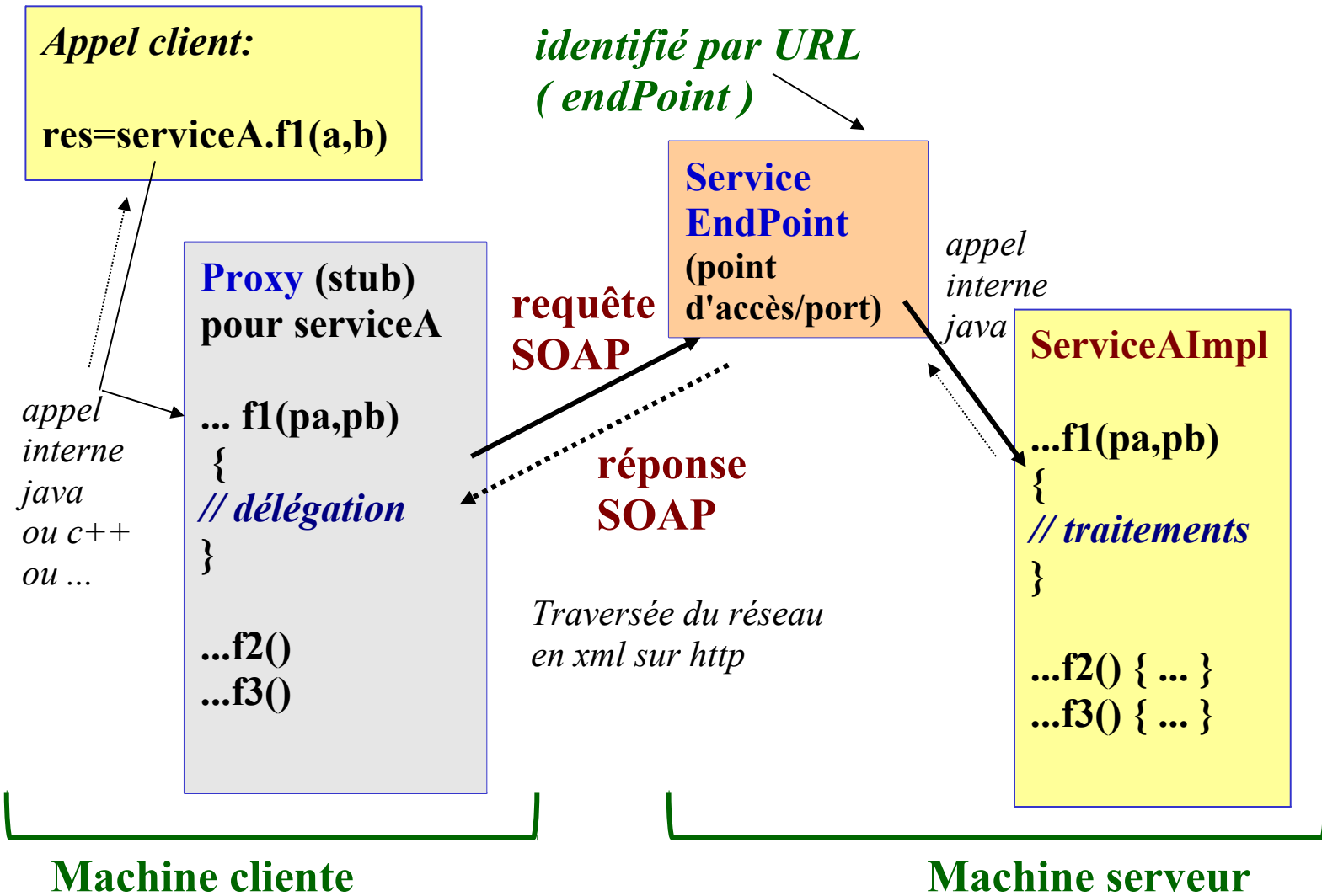
WSDL4J permet (si besoin) de manipuler directement un fichier "WSDL" depuis du code java . En couplant "WSDL4J" avec un peu d'introspection java on peut éventuellement déclencher des appels entièrement dynamiques (en groovy par exemple).

JAX-R (Java Api for Xml Registry) est une api officielle java permettant de se connecter à des annuaires de services (ex: UDDI , ebXml ,) .
UDDI4J (non officiel) existe aussi et est plus étroitement associé à UDDI.

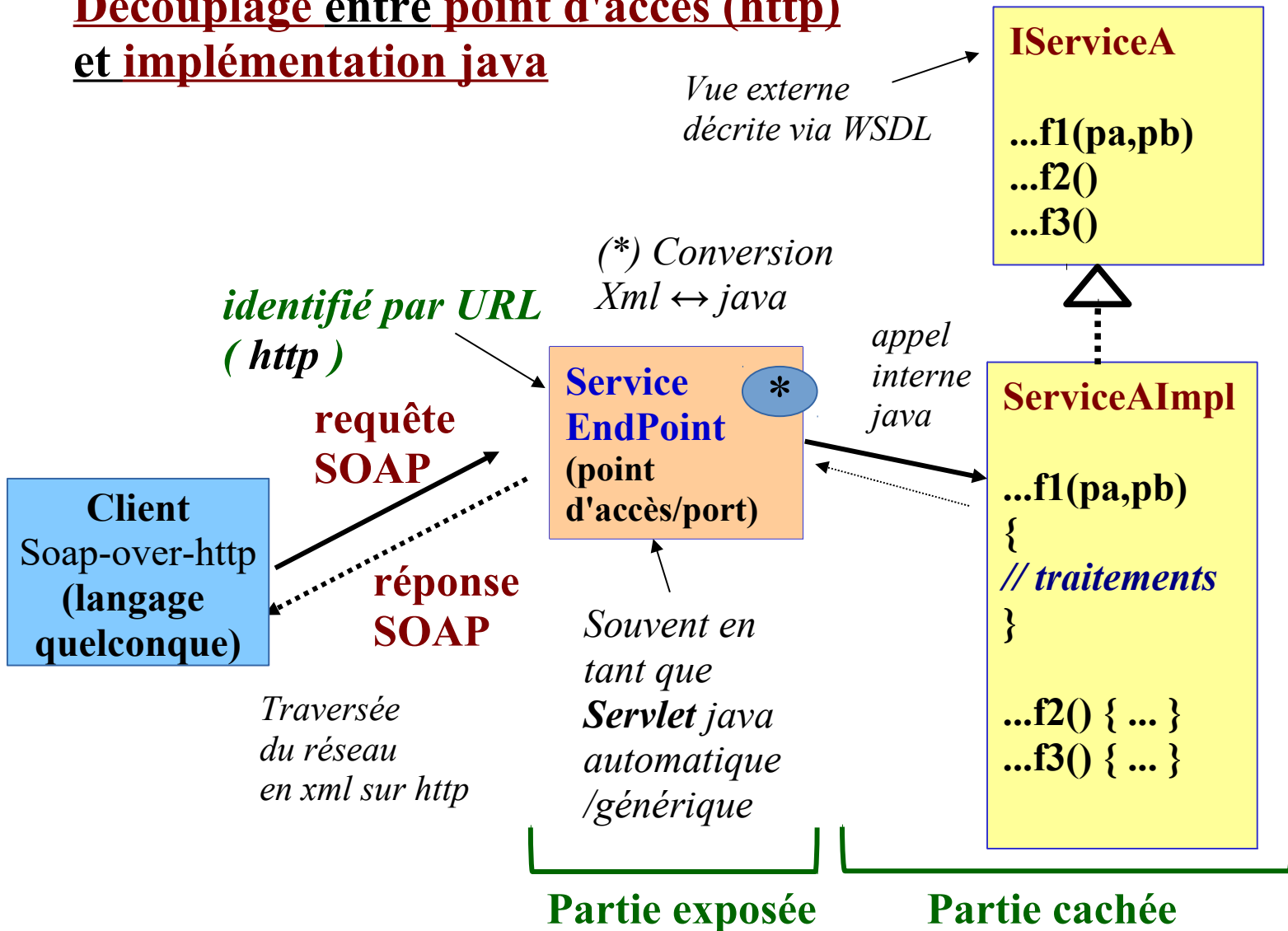
Anatomie d'un Service Web "SOAP"



Proxy et Point d'accès / Port avec SOAP

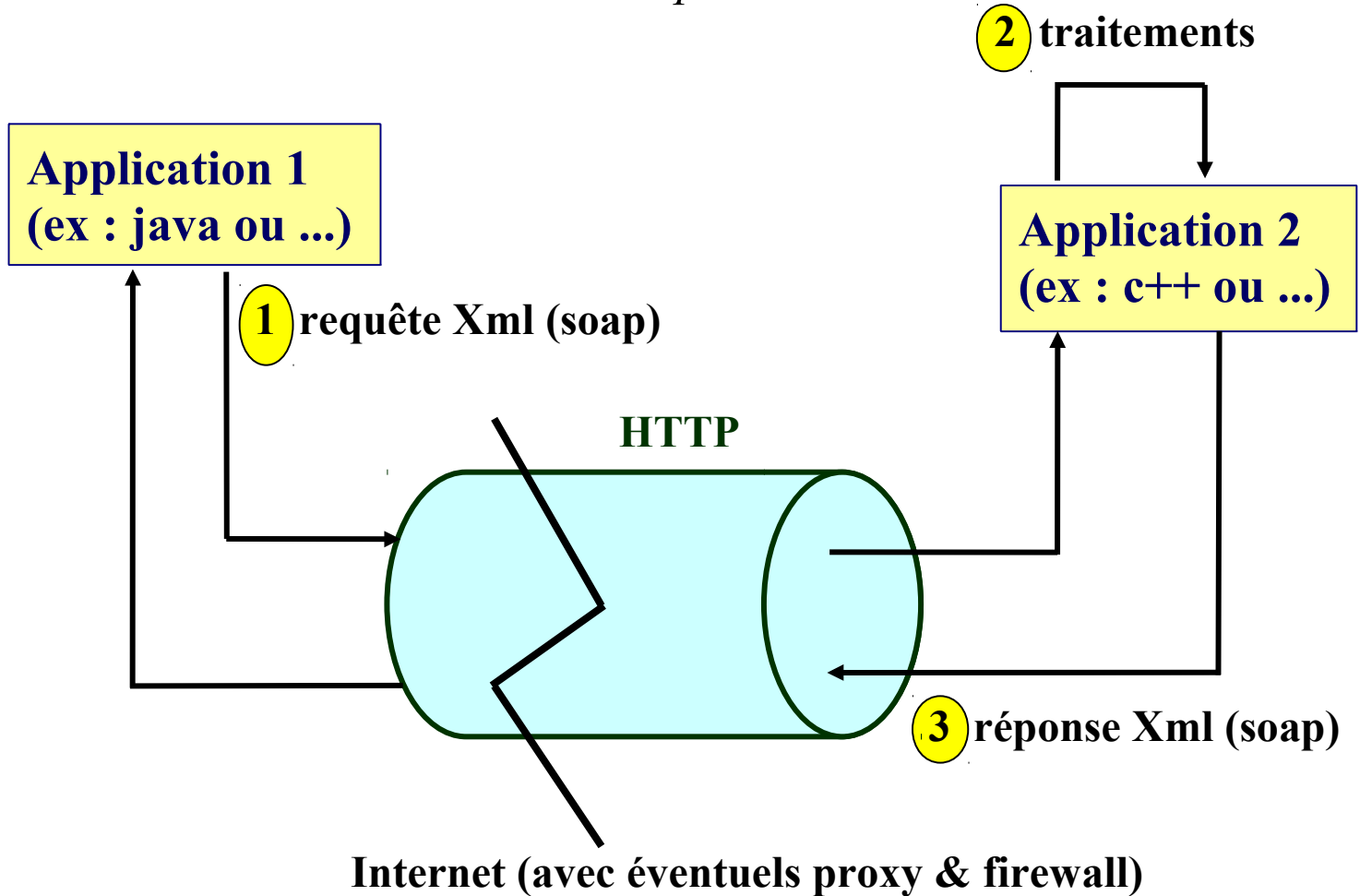


Découplage entre point d'accès (http) et implémentation java



SOAP (*Simple Object Access Protocol*)

ici en version "SOAP-over-Http"



Enveloppe SOAP (ici véhiculée via HTTP)

message http (requête / réponse)

```
POST /SoapEndUrl HTTP/1.1
Host: www.yyy.com
Content-type: text/xml
Content-Length: 524
SOAPAction:
```

Entête Http

```
<SOAP-ENV:Envelope
... xmlns:SOAP-ENV=
'http://schemas.xml.org/soap/envelope/'
>
```

Enveloppe SOAP
(Corps du message Http
au format *text/xml*)

```
<SOAP-ENV:Header>
...
</SOAP-ENV:Header>
```

Soap Header
facultatif

```
<SOAP-ENV:Body>
...
</SOAP-ENV:Body>
```

Soap Body

```
</SOAP-ENV:Envelope>
```

Contenu du corps de l'enveloppe "SOAP"

message http
(avec contenu SOAP)

requête

```
<methodeXy>
  <p1>val1</p1>
  <p2>val2</p2>
</methodeXy>
```

réponse

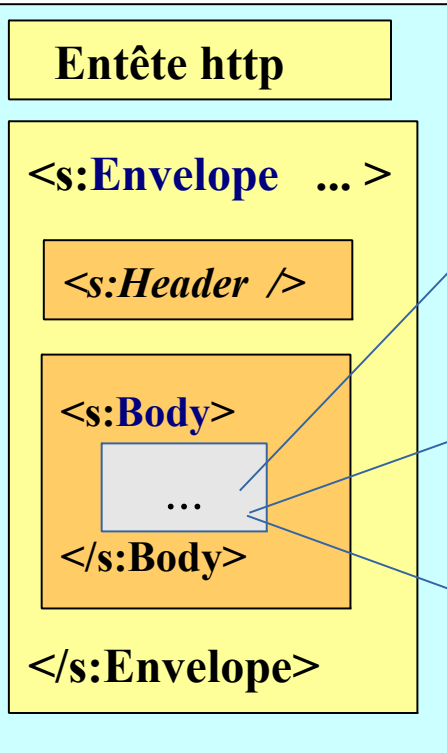
```
<methodeXyResponse>
  <return>val_resultat</return>
</methodeXyResponse>
```

Ou bien

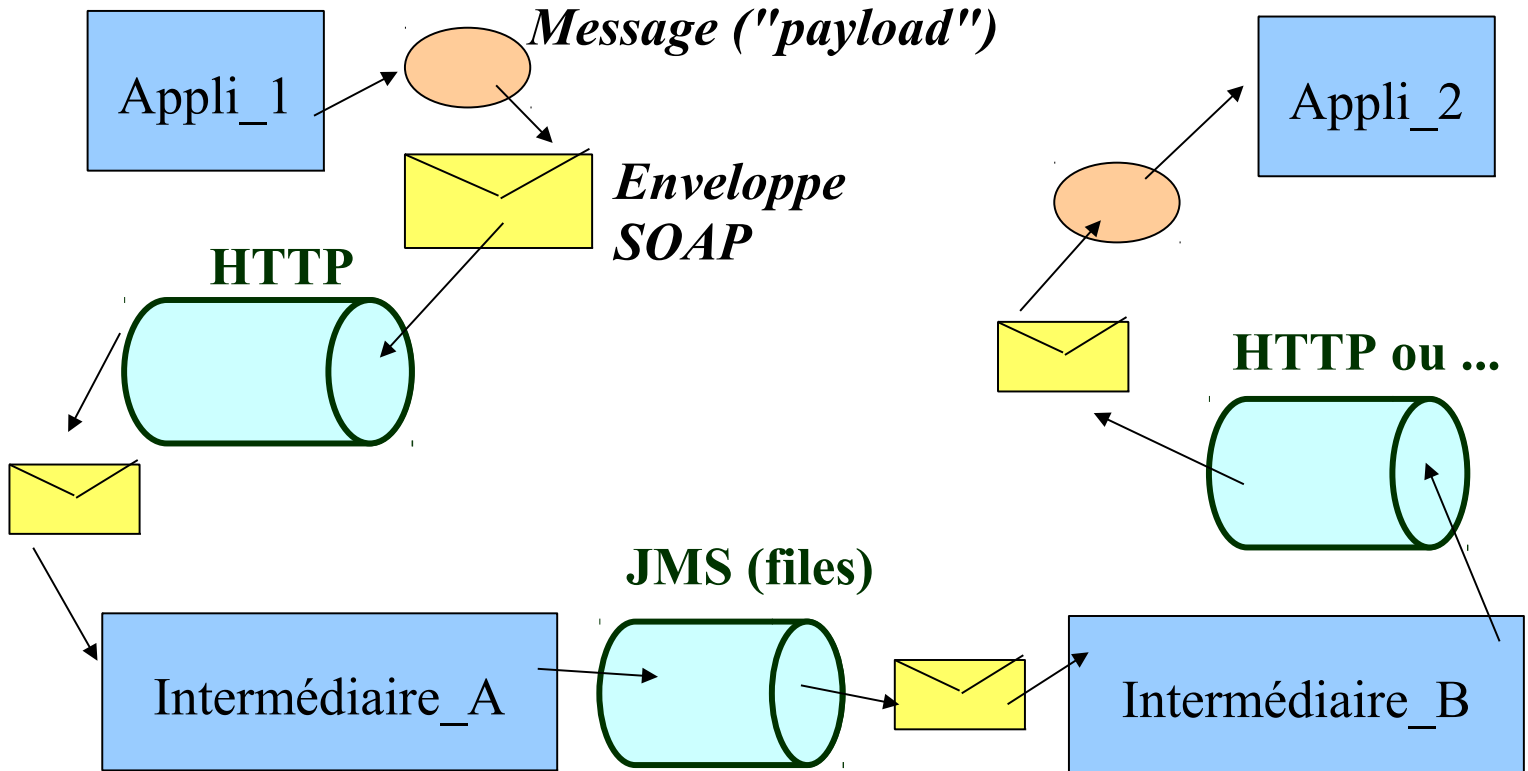
Ou bien

Exception (fault)

```
<s:fault>
  <faultcode>soap:Server</faultcode>
  <faultstring>msg_error</faultstring>
</s:fault>
```

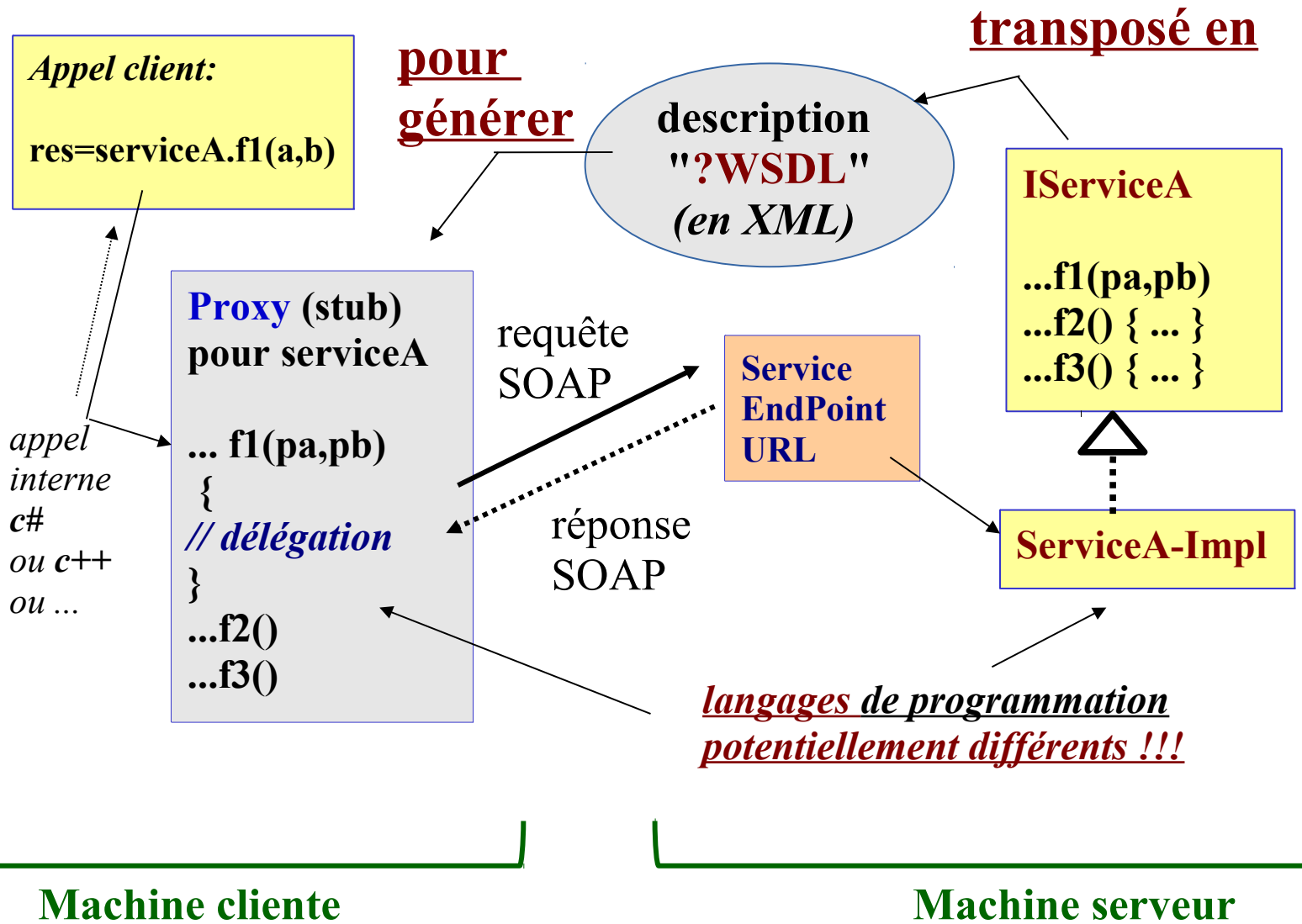


SOAP: "Payload", *enveloppe* et transports



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

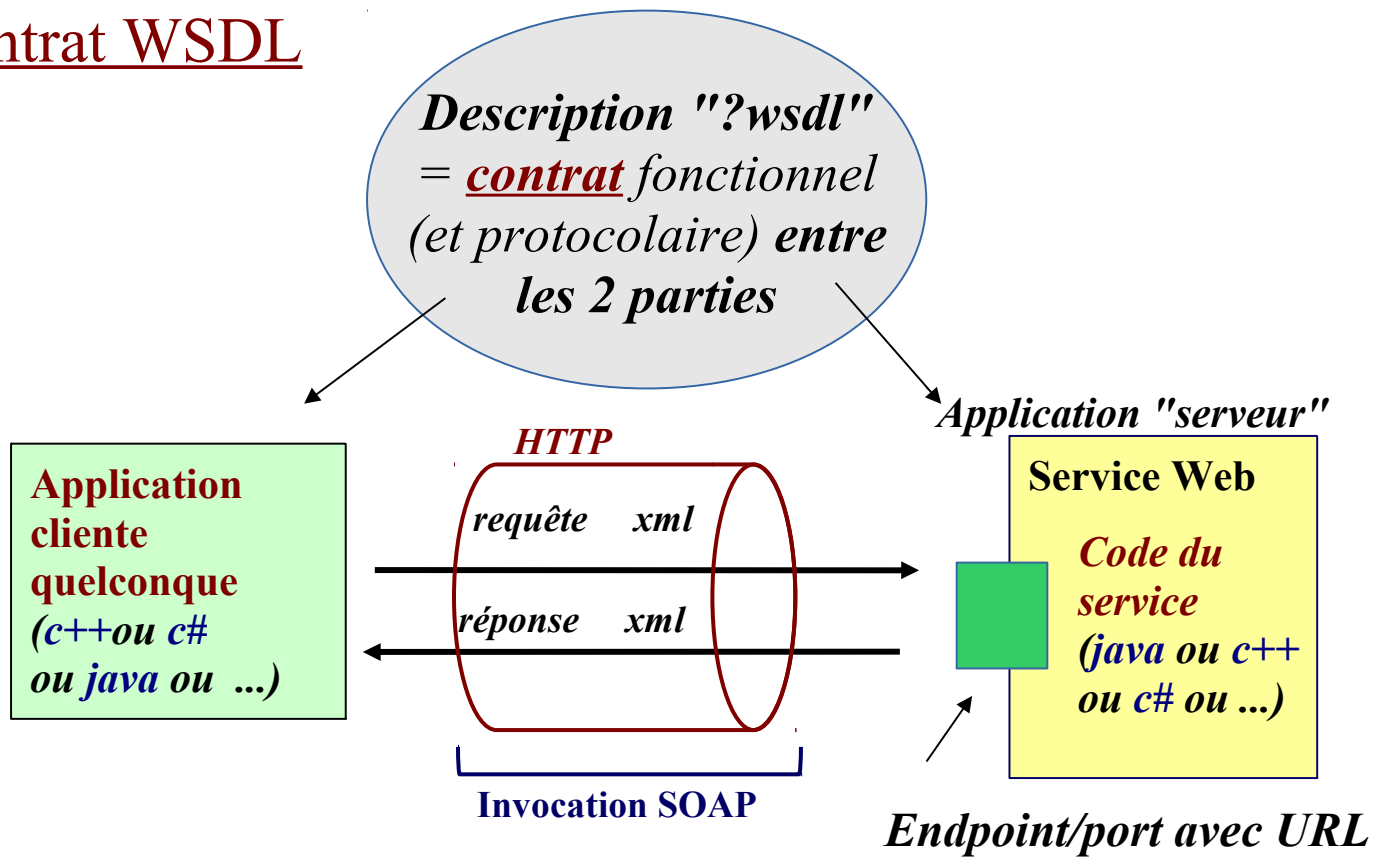
Utilité de la description WSDL



Description WSDL (*pour WS Soap*)

- **WSDL** = *Web Service Description Language*
- C'est une **description XML de la structure d'un service** soap accessible qui est volontairement **indépendante des langages de programmation** (*c++ ou java ou c# ou ...*).
- *L'essentiel d'une description WSDL peut être vu comme une transposition xml d'une interface java (liste d'opérations avec des types précis de paramètres en entrée et en retour).*
- *Un Fichier WSDL est souvent **généré automatiquement coté serveur**, publié au bout d'une URL, téléchargé, puis analysé par un programme du genre **wsdl2Java** ou **wsimport** pour générer un "**proxy**" coté client*

Contrat WSDL



Par convention, si l' url d'invocation soap d'un service publié est
<http://hostName/xxx/yyy>
alors l'URL menant à la description WSDL est
<http://hostName/xxx/yyy?wsdl> ← **?wsdl en plus**

Description WSDL (approches)

- Approche "à partir d'une implémentation" :
la description WSDL est automatiquement générée à partir de la structure orientée objet du code d'implémentation (ex : interface et classe java ou c++)
- Approche "à partir du contrat WSDL / xml" :
Le code d'implémentation du service web est construit (généré / codé / paramétré) en fonction d'un fichier WSDL existant (par exemple le fruit d'une norme ou d'une modélisation UML).

Dans tous les cas , le code des invocations "coté client" doit se conformer à un fichier WSDL.

*Ce code peut être complètement dynamique (ex : php)
ou bien sous forme de proxy généré/compilé (ex : java)*

Structure détaillée WSDL

definitions (avec *targetNamespace*)

types

schema

(*xsd*) avec *element* & *complexType* ,

message * (*message* de requête ou de réponse / paquet de paramètres)

part * (paramètre in/out ou bien return)

portType (interface , liste d'opérations abstraites)

operation *

input *mxxxMessageRequest*

output *mxxxMessageResponse*

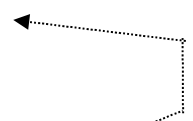
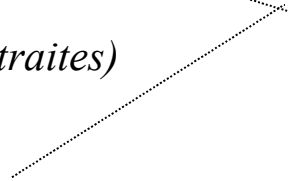
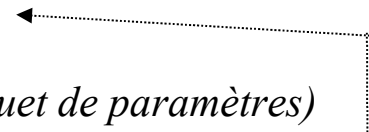
binding *

associations (portType ,
protocoleTransport[ex:HTTP] ,
style/use [ex: document/literal])

service (nom)

port * (point d'accès)

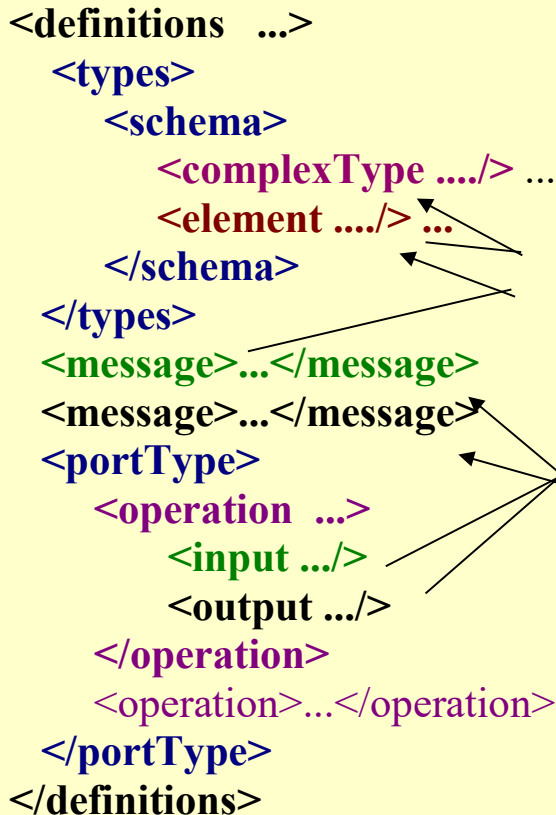
<address *location*="url_du_service" />



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

WSDL abstrait (*interface*)

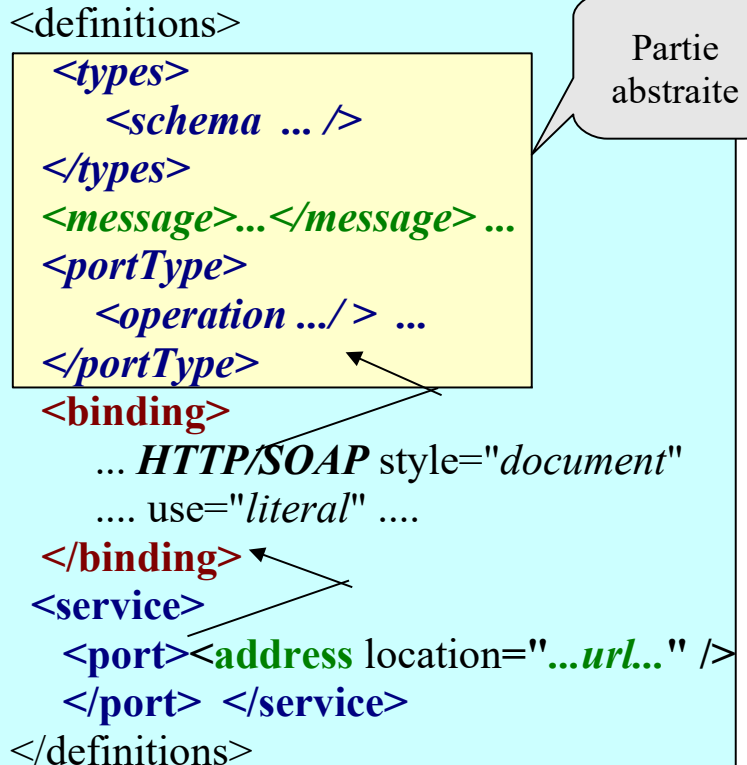
```
<definitions ...>
  <types>
    <schema>
      <complexType .../> ...
      <element .../> ...
    </schema>
  </types>
  <message>...</message>
  <message>...</message>
  <portType>
    <operation ...>
      <input .../>
      <output .../>
    </operation>
    <operation>...</operation>
  </portType>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

Sémantique de la partie abstraite WSDL

WSDL abstrait (*interface*)

```
<definitions ...>
  <types>
    <schema>
      <complexType .../> ...
      <element .../> ...
    </schema>
  </types>
  <message>...</message>
  <message>...</message>
  <portType name="IServiceA">
    <operation name="op1">
      <input .../>
      <output .../>
    </operation>
    <operation>...</operation>
  </portType>
</definitions>
```

Pour chaque opération opXy :

element="**opXy**" et "**opXyResponse**"

Avec *complexType* associés comportant la liste des *paramètres* (**noms** et **types**)

"pa" et "pb"

"xsd:int" et "xsd:string"

Simple renvois (pour compatibilité ancienne version)

<<interface>>

IServiceA

.op1(pa:int,pb:string)

.op2():int

PortType WSDL = interface

Structure et sémantique WSDL (partie concrète)

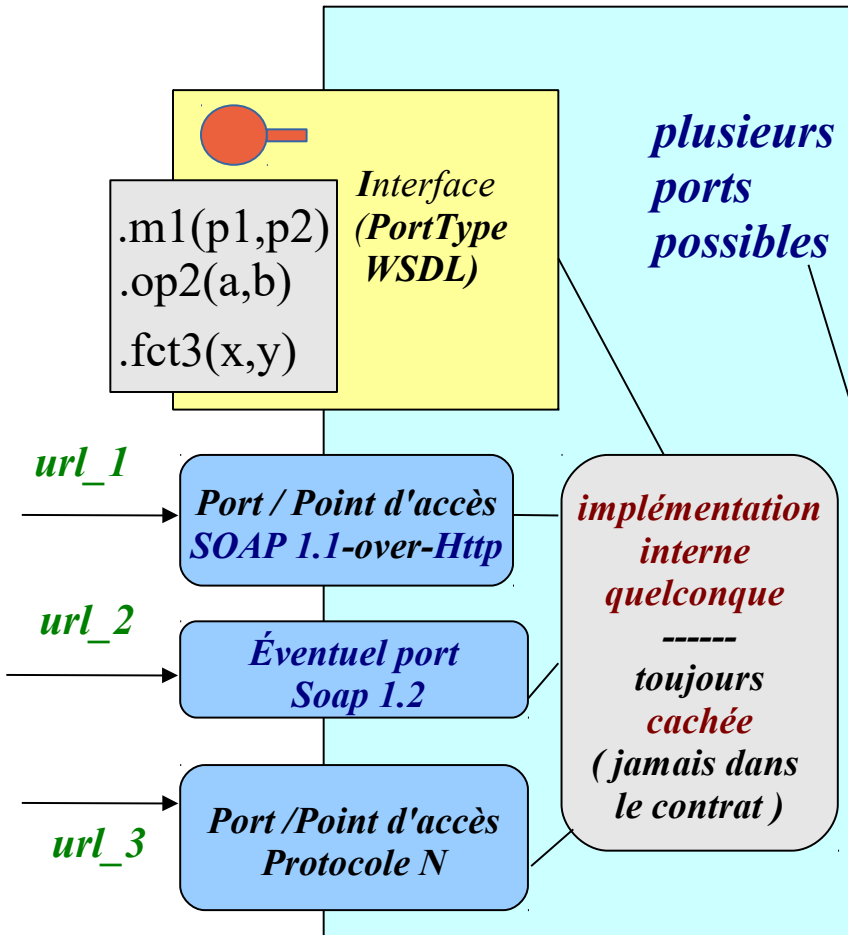
Un fichier "***WSDL concret***" comporte toute la partie abstraite précédente plus tous les paramétrages nécessaires pour pouvoir invoquer ***un point d'accès précis sur le réseau***.

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.

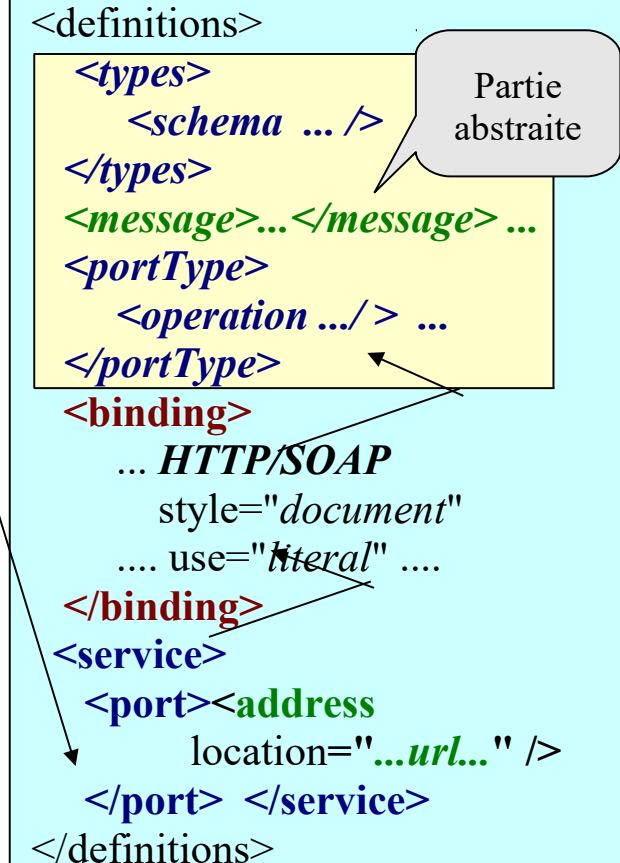
La partie "***binding***" (référéncée par un "port") permet d'***associer/relier*** entre eux les différents éléments suivants:

- * ***interface abstraite (portType)*** et ses opérations
- * ***protocole de transport (HTTP + détails ou)***
- * ***point d'accès (port)***

Sémantique de la partie concrète WSDL



WSDL concret/complet (*impl*)



Importance des namespaces XML d'un WSDL

- * La plupart des éléments décrits dans un fichier WSDL ont des **noms qualifiés** (par le **targetNamespace** de la balise <definition>)
- * Le nom complet d'une **classe java** est "**packageName.className**"
- * Le nom complet d'un **service** ou **portType** est "**tns:XyName**" avec **xmlns:tns="http://packageName_a_l_envers/"** (*coïncidant avec la valeur du **targetNamespace** du fichier WSDL*).

Exemple :

com.xy.Customer en tant que POJO de donnée en retour d'un WS devient en WSDL **xmlns:ns="http://xy.com/"** et "**ns:Customer**" et est re-traduit en **com.xy.Customer** au sein des applications clientes.

Quelquefois "package java" --> namespace xml --> namespace c++

Test d'un web-service "soap"

- 1) **Coder** le service web et **démarrer** l'application ou le serveur qui l'héberge.
- 2) Utiliser un **navigateur internet** pour **visualiser la description WSDL** qui devrait être accessible si tout va bien :

<http://localhost:8080/myApp/xx/yy/serviceA?wsdl>

```
<?xml version="1.0" encoding="UTF-8"?>
  <wsdl:definitions>
    ....
  </wsdl:definitions>
```

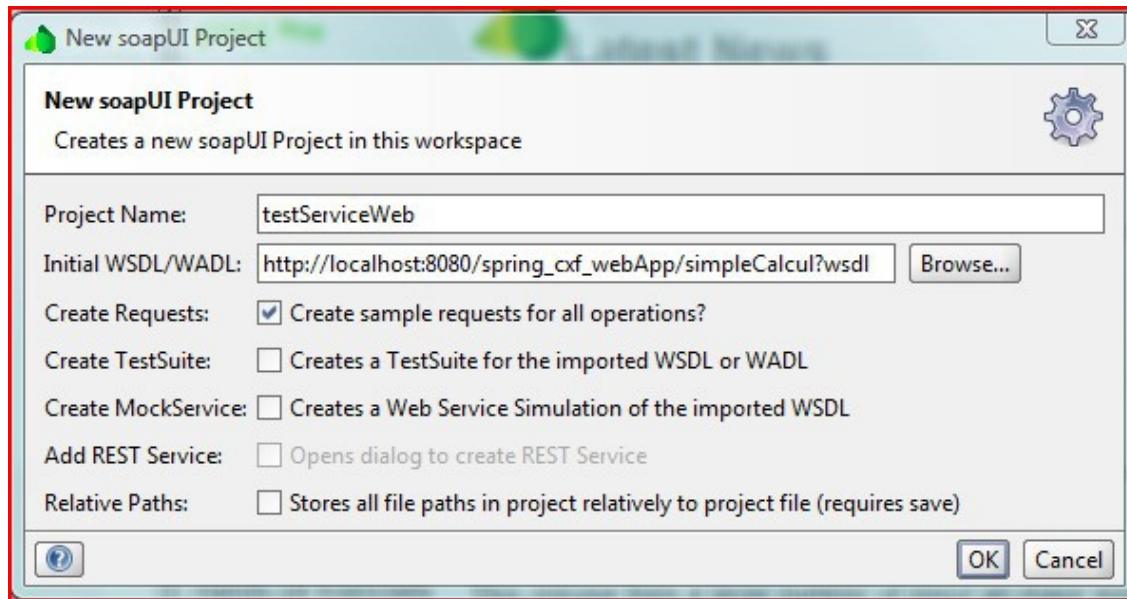
**ne pas
oublier
?wsdl**

- 3) Utiliser l'application de test "soap-ui" (ou un équivalent) pour analyser le fichier WSDL et **lancer des requêtes "soap"** .

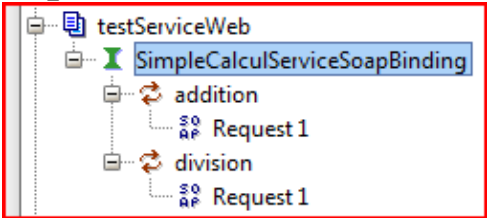
Une version gratuite de **soap-ui** peut se télécharger et s'installer facilement.

Après avoir démarrer l'application soap-ui , il faut **créer un nouveau projet de test (à nommer)**.

Paramètre à fixer : l'**url** complète de la description **wsdl** .




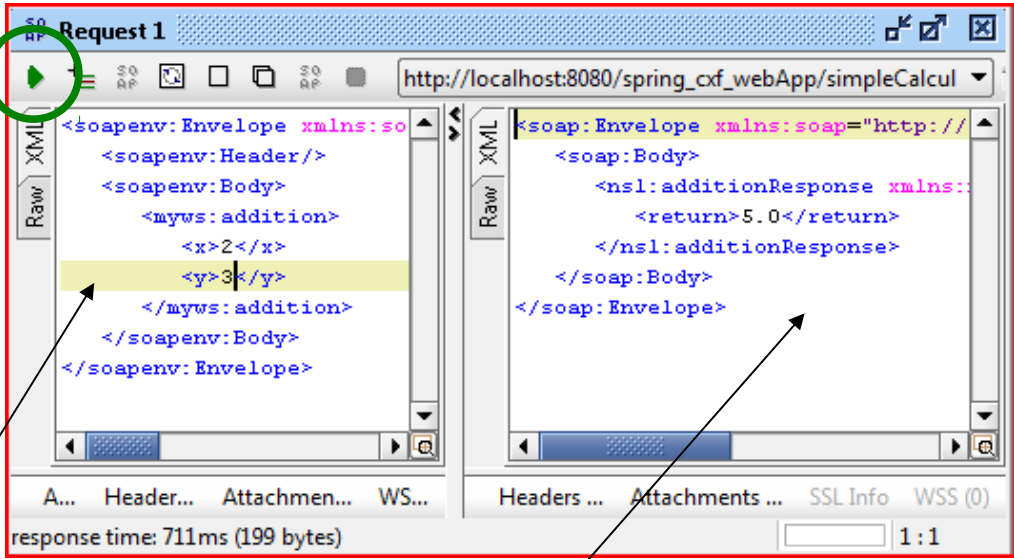
Soap-ui analyse la description wsdl et reconnaît les opérations existantes que l'on peut tester :



Un (double-)clic sur "request1" d'une méthode à tester fait apparaître une structure de requête soap/xml où il faut simplement spécifier quelques valeurs .

déclenchement





The screenshot shows the 'Request 1' window in SoapUI. The left pane displays the XML request structure: `<soapenv:Envelope xmlns:so...>` containing `<myws:addition>` with values `<x>2</x>` and `<y>3</y>`. The right pane displays the XML response structure: `<soap:Envelope xmlns:soap="http://...>` containing `<ns1:additionResponse>` with the value `<return>5.0</return>`. The status bar at the bottom indicates 'response time: 711ms (199 bytes)'.

requête soap

réponse soap (ou erreur)

Présentation de l'api JAX-WS

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

JAX-WS est déjà intégré dans le jdk ≥ 1.6

Principales caractéristiques :

- paramétrage par annotations (ex: *@WebService* , *@WebParam*)
- utilisation interne de *JAXB2* (et ses annotations *@Xml...*)

JAX-WS est dédié au services "SOAP" avec description WSDL.

Une partie de JAX-WS est dédiée à WS-S (WS-Security).

La technologie "CXF" complète le jdk sur certains points (sécurité , intercepteurs , intégration webApp , intégration spring, ...)

Paramétrage d'une **interface** de service web

```
//@WebService(targetNamespace="http://services.myapp.xy.com/")
@WebService
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;

}
```

Paramétrage d'une classe d'implémentation

```
/*@WebService(targetNamespace="http://impl.services.myapp.xy.com/",  
endpointInterface="com.xy.myapp.services.impl.GestionComptes")*/
```

```
@WebService(endpointInterface=  
    "com.xy.myapp.services.impl.GestionComptes")  
public class GestionComptesImpl implements GestionComptes {  
    ... // code d'implémentation java habituel (R.A.S.)  
}
```

Paramétrage d'une classe de données (en entrée ou sortie)

```
@XmlType(namespace="http://data.myapp.xy.com/")  
@XmlElement(name="stat")  
public class Stat {  
    private int num_mois; //de 1 a 12  
    private double ventes; //+get/set  
    ... }
```

Principaux paramétrages JAX-WS:

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** .

Namespace par défaut: "**http://**" + nom_package_java_à_l_envers + "/"

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 a "**return**" .

JAX-WS coté serveur en ne s'appuyant que sur le **jdk** >=1.6

Dans la plupart des cas réalistes d'entreprise , un service web est un simple composant d'une application web (jee, spring/cxf ou ...)

Cependant, pour effectuer rapidement de simples tests , un **service web JAX-WS peut démarrer avec une simple JVM** (et son mini conteneur web "jetty" intégré)

Les premières versions de JAX-RS (époque du début du jdk 1.6) nécessitaient le lancement de

`${JAVA_HOME}/bin/wsgen -d ... -cp ... WsImpClassName`
pour générer certaines classes techniques utiles à l'époque.

Aujourd'hui **les versions récentes de JAX-RS** (époque jdk >=1.7) **ne nécessitent plus le lancement de wsgen** car les classes techniques sont générées dynamiquement.

Code de démarrage (pour tests avec JVM seulement)

```
import javax.ws.* ;
public class StandaloneWsTestApp {
    protected StandaloneWsTestApp() throws Exception {
        System.out.println("Starting Server");
        CalculateurImpl wsImplementor = new CalculateurImpl();
        String address =
            "http://localhost:8080/myApp/services/calculateur";
        Endpoint.publish(address, wsImplementor);
    }
    public static void main(String args[]) throws Exception {
        new StandaloneWsTestApp();
        System.out.println("Server ready (waiting for soap request)...");
        Thread.sleep(15 * 60 * 1000); // 15 minutes avant arrêt
        System.out.println("Server exiting"); System.exit(0);
    }
}
```

→ <http://localhost:8080/myApp/services/calculateur?wsdl> à tester (navig. + soap-ui).

JAX-WS coté client (avec description WSDL et **wsimport** du **jdk** >= 1.6)

Dans le cas général , une application java "cliente" peut potentiellement invoquer un web-service "soap" développé dans un autre langage de programmation (ex : c++ , c# , ...). Seule la description WSDL du service à appeler est alors connue et accessible au bout d'une URL se terminant par "?wsdl" .

La commande `${JAVA_HOME}/bin/wsimport` du **jdk** >= 1.6 permet d'analyser une description WSDL et de générer un paquet de classes et interfaces java correspondant à un "wsProxy" .

L'option **-keep** de **wsimport** permet de garder le code source des classes générées (sinon par défaut pseudo-code ".class" seulement)

L'option **-d src** (ou **-d src/main/java**) permet de préciser le répertoire attendu pour le code généré

Code généré par wsimport du jdk>=1.6

lancer-wsimport.bat

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

→ *Parsing WSDL ... Generate code ...*

Après un "refresh eclipse" , on voit (en tant que code généré) :

- * **package(s) java** (selon *targetNamespace* du wsdl)
et à l'image du/des package(s) du serveur
- * une **interface java** correspondant au **web-service à appeler**
- * une **classe ".....Service"** avec une **méthode "*get....Port()*"** qui
sert à se connecter au web-service et récupérer un proxy JAX-WS.
- * d'éventuelles classes de données (en entrées/sorties) des méthodes
- * des classes techniques "...Response" , "... " pour mécanismes internes

Code type d'un client JAX-WS s'appuyant sur wsimport

```
package client;
import calcul.Calculator; // interface du Service (SEI)
import calcul.CalculatorService;

public class MyWsClientApp {

    public static void main (String[] args) {
        try {
            Calculator calcProxy = (new CalculatorService()).getCalculatorPort();
            int a = 10; int b = 20;
            System.out.printf ("Invoking addition(%d, %d)\n", a,b);
            int res = calcProxy.addition (a, b);
            System.out.printf ("The result of adding %d and %d is %d.\n\n", a, b, res);

        } catch (Exception ex) {
            System.out.printf ("Caught Exception: %s\n", ex.getMessage ());
        }
    }
}
```

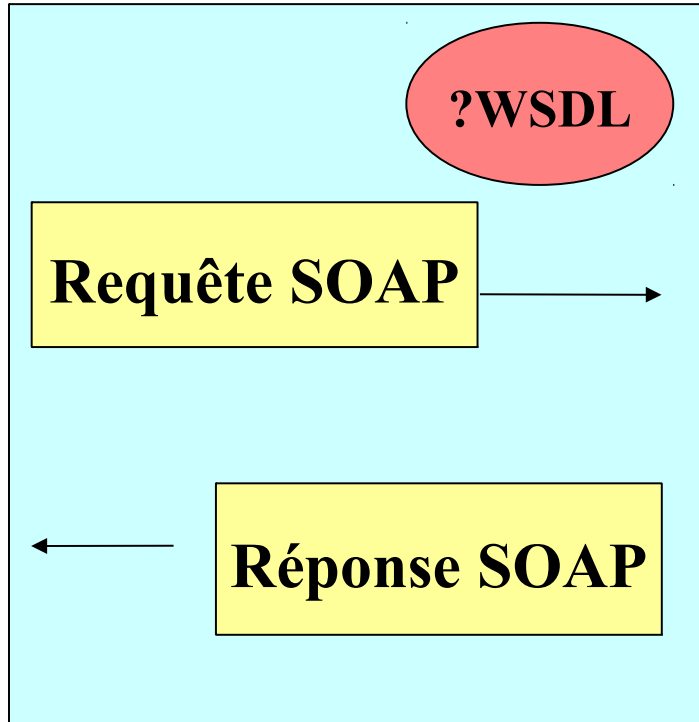
→ Résultat de l'appel distant "soap": $10+20=30$

Client (java, .net ,...)

http/xml

Serveur (java, .net ,...)

*envoi /
réception
de
document
ou
bien
appel de
méthode
distant*



rpc/literal
ou document/literal

*traitement
de
document
ou
bien
exécution
méthode*

"style / use" SOAP

rpc/encoded (*has been*):

Il s'agit historiquement du premier mode SOAP normalisé (parties "SOAP Encoding + Soap Rpc" de la version 1.1). Ce *mode ancien n'est pas compatible WS-I* et est aujourd'hui supplanté par rpc/literal.

rpc/literal :

Ce mode est plus performant que rpc/encoded car l'encodage est plus simple. Ce mode est de plus compatible WS-I (bonne interopérabilité). Le mode "rpc/literal" encode bien les noms des méthodes appelées mais ne se prête pas bien à une éventuelle validation des messages SOAP.

document/literal (*par défaut*):

Le mode *document/literal* est avant tout **adapté à l'échange de contenus xml** (dont les structures de données sont bien décrites au format "schéma xml" au sein du fichier WSDL).

==> Validation possible des messages entrants (via sous partie "xsd" du wsdl).

==> **peut également convenir pour des appels de méthodes (RPC)**

car qui peut plus peut moins (et la différence/surcharge est coté WSDL, pas SOAP)

Paramétrages JAX-WS secondaires:

```
package calcul;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public interface .... {
    @WebMethod(operationName="add") // "add" coté WSDL
    public int addition(int a, int b) .....
}
```

NB : L'annotation **SOAPBinding** est facultative. Elle permet de choisir (quand c'est possible) le **style SOAP** (**Style.RPC** ou **Style.DOCUMENT**) .

Le mode par défaut "**document/literal**" convient très bien dans presque tous les cas .

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)

Code java avec exception personnalisée coté serveur

```
public class CalculException extends Exception {  
    private static final long serialVersionUID = 1L;  
    public static enum ErrorCodeEnum { UNKNOWN , DIV_BY_ZERO ,  
                                           ERROR2 , ERROR3 };  
    private ErrorCodeEnum errorCode = ErrorCodeEnum.UNKNOWN;  
    private String details=""; // +get/set  
  
    public CalculException() {super(); }  
    public CalculException(String msg, Throwable cause) {super(msg, cause); }  
    public CalculException(String msg) {super(msg);}  
}
```

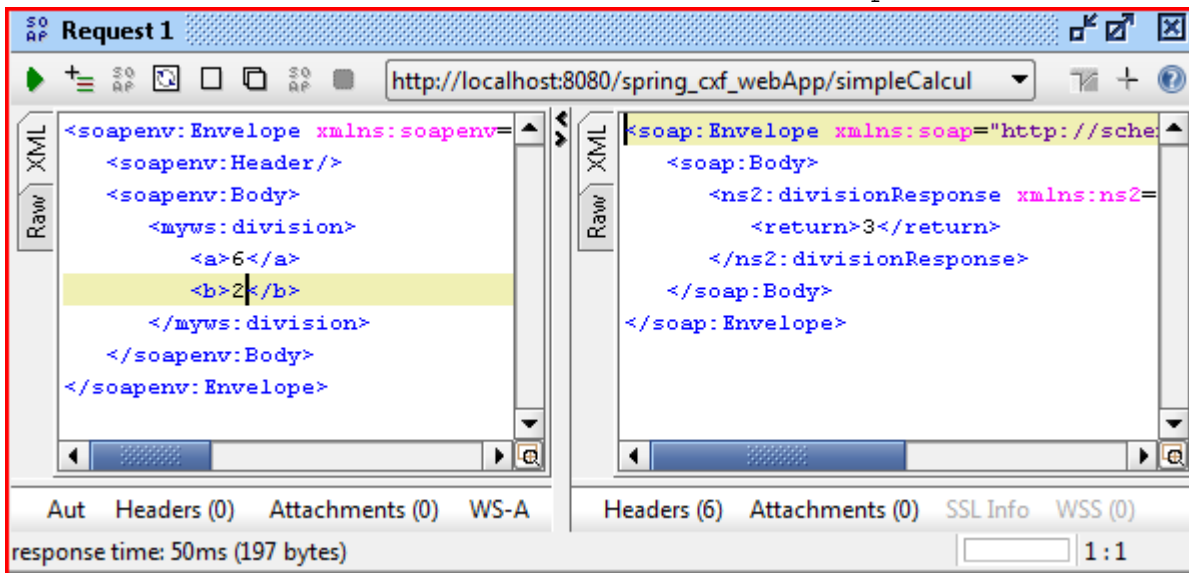
```
@WebService  
public interface SimpleCalcul {  
    ...  
    public int division(  
        @WebParam(name="a")int a,  
        @WebParam(name="b")int b)  
        throws CalculException;  
}
```

```
public int division(int a, int b)  
    throws CalculException {    int res=0;  
    if(b==0){  
        CalculException e = new CalculException(  
            "division par 0 interdite");  
        e.setErrorCode(CalculException  
            .ErrorCodeEnum.DIV_BY_ZERO);  
        e.setDetails("a="+a+" div by b="+b);  
        throw e;  
    }  
    else res=a/b;  
    return res; }
```

Transposition de l'exception en "fault" coté WSDL

```
<wsdl:definitions name="SimpleCalculService" targetNamespace="http://myws/" ...>
<wsdl:types>
  <xs:schema ...> ...
    <xs:simpleType name="errorCodeEnum">
      <xs:restriction base="xs:string">
        <xs:enumeration value="UNKNOWN"/> <xs:enumeration value="DIV_BY_ZERO"/>
        <xs:enumeration value="ERROR2"/> <xs:enumeration value="ERROR3"/>
      </xs:restriction>
    </xs:simpleType>
    <xs:element name="CalculException" type="tns:CalculException"/>
    <xs:complexType name="CalculException">
      <xs:sequence>
        <xs:element name="errorCode" nillable="true" type="tns:errorCodeEnum"/>
        <xs:element name="details" nillable="true" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:schema>
</wsdl:types>
<wsdl:message name="CalculException">
  <wsdl:part element="tns:CalculException" name="CalculException"/>
</wsdl:message>
<wsdl:portType name="SimpleCalcul">
  <wsdl:operation name="division">
    <wsdl:input message="tns:division" name="division"/>
    <wsdl:output message="tns:divisionResponse" name="divisionResponse"/>
    <wsdl:fault message="tns:CalculException" name="CalculException"/>
  </wsdl:operation>
</wsdl:portType>
...
```

test avec b différent de zéro ==> aucune exception :



The screenshot shows a SOAP client window titled "Request 1". The address bar displays the URL "http://localhost:8080/spring_cxf_webApp/simpleCalcul". The left pane shows the raw XML of the request, which is a division of 6 by 2. The right pane shows the raw XML of the response, which is a division result of 3. The status bar at the bottom indicates a response time of 50ms and 197 bytes.

```
<?xml version='1.0' encoding='UTF-8'>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header/>
  <soapenv:Body>
    <myws:division>
      <a>6</a>
      <b>2</b>
    </myws:division>
  </soapenv:Body>
</soapenv:Envelope>
```

```
<?xml version='1.0' encoding='UTF-8'>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:divisionResponse xmlns:ns2="http://myws/">
      <return>3</return>
    </ns2:divisionResponse>
  </soap:Body>
</soap:Envelope>
```

response time: 50ms (197 bytes) 1:1

*Retranscrit
automatiquement
en exception
java (ou c++
ou ...)
coté client*

Retour "soap" en cas d'exception (si b==0) :

```
<soap:Envelope ...> <soap:Body> <soap:Fault>
  <faultcode>soap:Server</faultcode>
  <faultstring>division par 0 interdite</faultstring>
  <detail> <ns1:CalculException xmlns:ns1="http://myws/">
    <errorCode xsi:type="ns2:errorCodeEnum"...>DIV_BY_ZERO</errorCode>
    <details xmlns:ns2="http://myws/"> a=6 div by b=0</details>
  </ns1:CalculException>
</detail>
</soap:Fault> </soap:Body> </soap:Envelope>
```

Besoin de redéfinir l'URL d'un web-service à invoquer

L' **URL** d'un **service web** change souvent et particulièrement dans les cas suivants :


- * mode développement (tests locaux) : **http://localhost:8080/...**
- * test d'intégration : **http://192.168.50.200:8080/...**
- * qualification : **http://nom_machine_qualif/...**
- * production : **http://www.virtualhostName/...**

Le code d'un proxy généré par **wsimport** comporte l'URL au moment du développement (**http://localhost:8080/...**) en tant que valeur par défaut qu'il faudra redéfinir.

Une bonne pratique consiste à récupérer l'**URL variable** dans un fichier "**ws_url.properties**" pour en tenir compte lors de l'initialisation de l'application cliente.

éventuelle redéfinition de l'URL "SOAP"

```
String soapURL = "http://localhost:1234/myWebApp/services/calculateur" ;  
    // ici en dur ou à récupérer depuis fichier ".properties"  
  
Calculateur wsProxy = serviceCalculateur.getCalculateurServicePort();  
  
javax.xml.ws.BindingProvider bp = (javax.xml.ws.BindingProvider) wsProxy;  
Map<String,Object> context = bp.getRequestContext();  
context.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, soapURL);  
  
System.out.println(wsProxy.getTva(200, 19.6));
```



*Code générique (technique , indépendant du fonctionnel)
à idéalement placer dans une classe utilitaire*

Attention : redéfinir l'URL soap ne suffit pas toujours !

Les mécanismes internes de l'api JAX-WS puisent certains détails dans le fichier "?wsdl" . Si l'emplacement de celui-ci a changé (depuis le développement – ce qui est souvent le cas) , il faudra également (ou alternativement) redéfinir l'url de la description WSDL !

Redéfinition (souvent indispensable) de l'URL "WSDL"

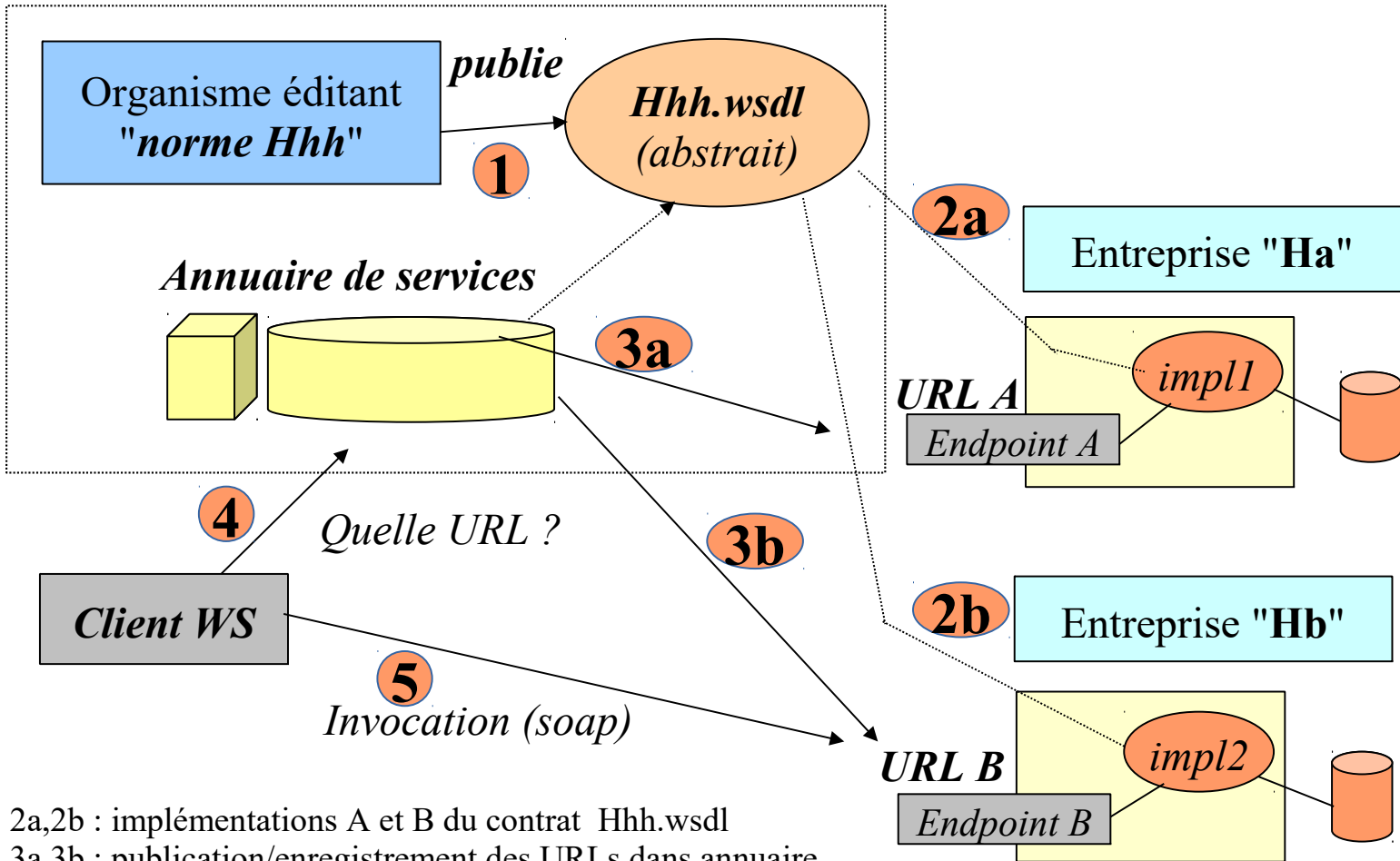
```
String wsdlUrlAsString = "http://localhost:8080/convWeb/services/conv?wsdl";  
    // ici en dur ou à récupérer depuis fichier ".properties"  
URL urlWsdl = new URL(wsdlUrlAsString);  
  
QName sQname = new  
    QName("http://conversion/", "ConvertisseurImplService");  
  
ConvertisseurImplService s = new ConvertisseurImplService(urlWsdl, sQname);  
  
Convertisseur wsProxy = s.getConvertisseurImplPort();  
System.out.println(wsProxy.convertir(200.0, "euro" , "dollar"));
```

NB: Les paramètres de `QName()` correspondent aux valeurs des attributs *targetNamespace* et *serviceName* de la balise *definition* du WSDL .

La classe*Service* générée par `wsimport` comporte plusieurs constructeurs dont un qui est prévu pour préciser l'url WSDL et le *qualifiedName* du service.

En redéfinissant l'URL du fichier WSDL , les mécanismes de JAX-WS récupèrent indirectement dans cette version (censée être à jour) l'URL SOAP à utiliser .

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

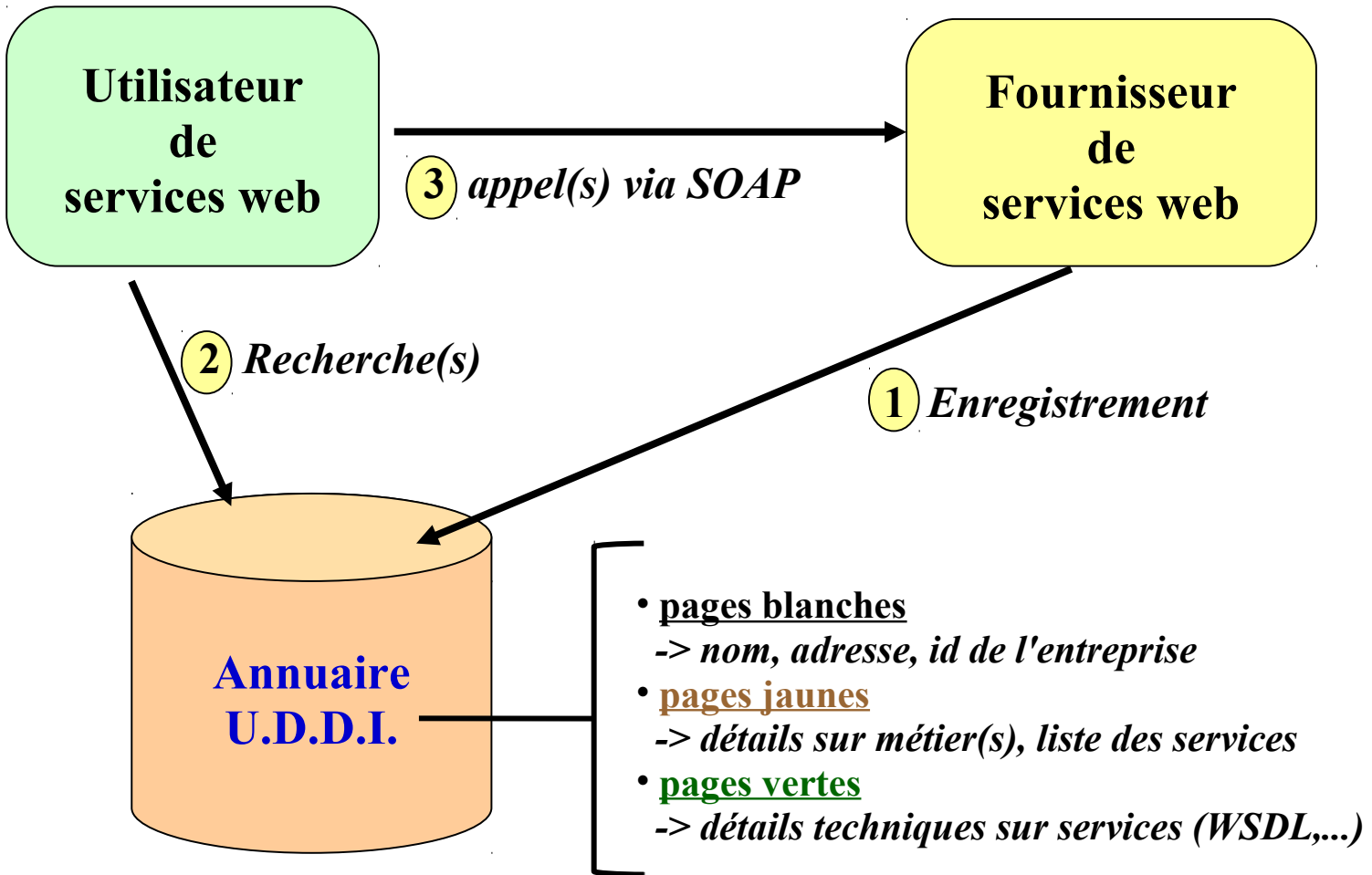


2a,2b : implémentations A et B du contrat Hhh.wSDL

3a,3b : publication/enregistrement des URLs dans annuaire

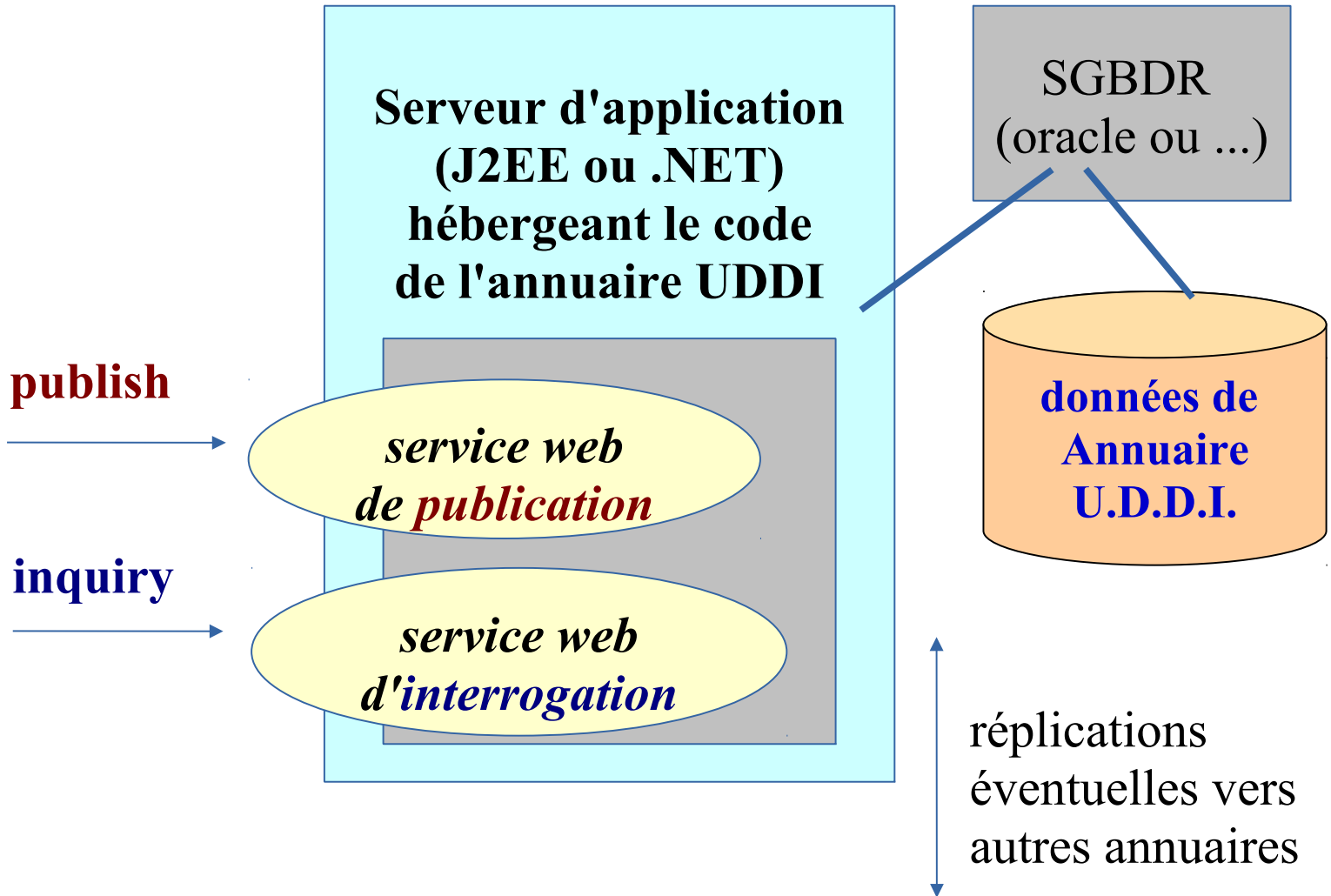
4 : interrogation de l'annuaire pour connaître URL possibles pour WS compatible(s) Hhh.wSDL

U.D.D.I. (norme **complexe** pour annuaire de services)



U.D.D.I = Universal Description Discovery and Integration

Annuaire UDDI (grandes lignes)



Avantages et inconvénients de UDDI

Rappel : un annuaire de services est une base de données d'URL avec des critères de recherches associés .

UDDI est une **technologie complexe** prévue pour le **commerce électronique international**.

Dans un contexte international, une recherche géographique ou par mot clef doit pouvoir s'effectuer dans plusieurs langues humaines (ex : "france" , "frankreich" , ...) ("hotel" , "hostel" , ...) .

Pour effectuer des recherches précises (sans ambiguïtés) , certains critères sont renseignés en tant que valeur de **taxonomie** (encodage normalisé ISO ou autres)

Chaque élément d'un annuaire UDDI à un **identifiant complexe** (uuid)

Une publication ou interrogation nécessite une **authentification préalable** (obtention d'un **token** à réinjecter)

La **structure (très décomposée) de l'annuaire UDDI est complexe**.

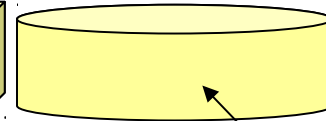
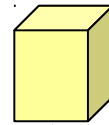
Au départ, porté par "IBM" et "Microsoft" , UDDI est maintenant **passé de mode**.

Très rare utilisation directe d'un annuaire

Application Cliente
codée avec JAX-WS

- 1. initialisation proxyWS*
- 2. recherche url via annuaire*
- 3. paramétrer proxyWs avec url trouvée*
- 4. effectuer appels "soap" vers WS.*

Annuaire de services



*structure
UDDI
ou
simplifiée*

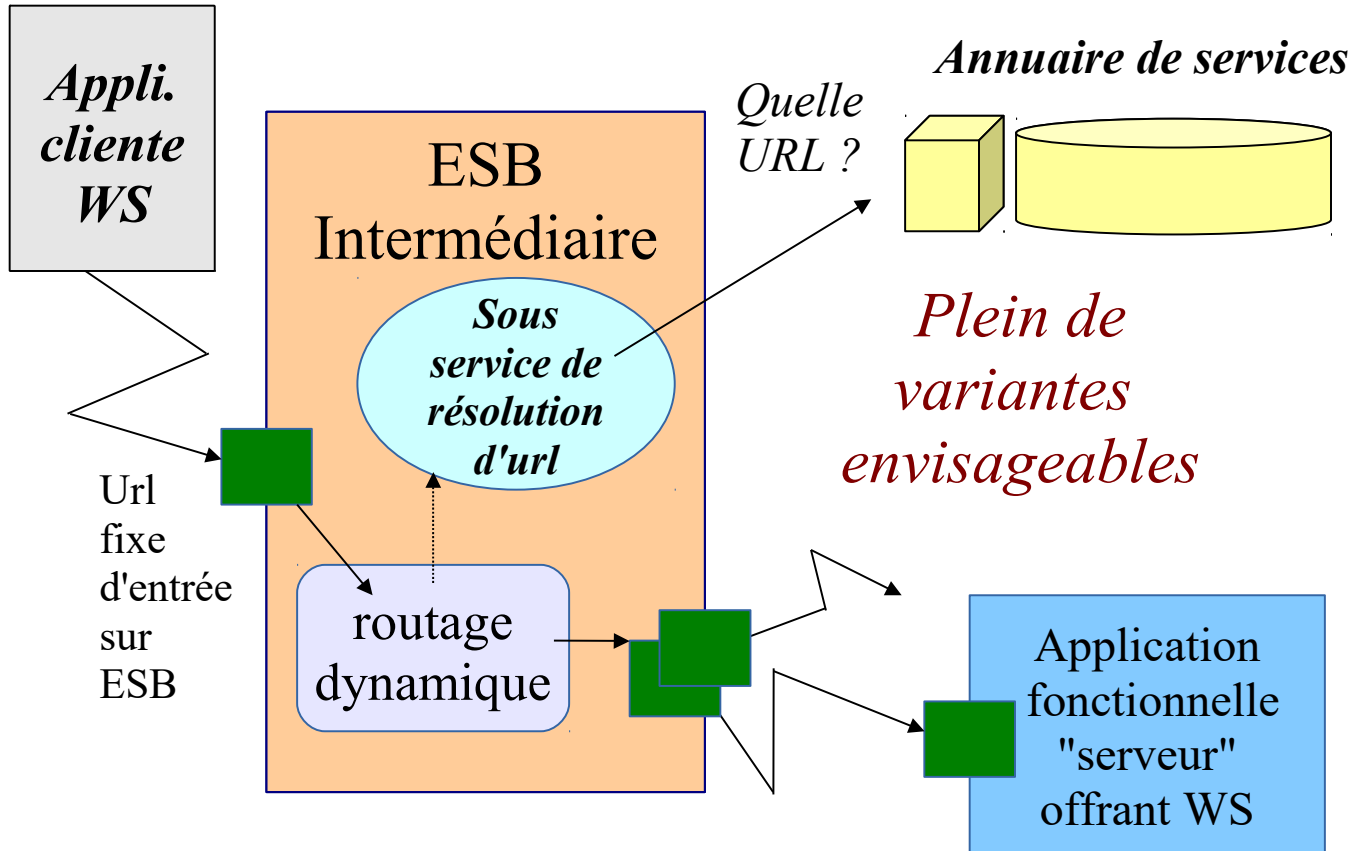
*Quelle
URL ?*

*Publication
préalable*

Application
fonctionnelle
"serveur"
offrant WS



Utilisation (très souvent) **indirecte** d'un annuaire



Invocation JAX-WS dynamique

Si l'on dispose de l'interface java du WS à appeler (**SEI*) on peut alors générer dynamiquement un client d'appel (*wsProxy*) sans devoir préalablement générer celui-ci via *wsimport* et la description WSDL.

NB : L'interface java (**SEI : Service EndPoint Interface*) doit au minimum comporter l'annotation **@WebService** (et souvent **@WebParam**)

L'interface java peut s'obtenir par copie partielle du code de l'application "serveur de WS" si celle ci est codée en Java.

Dans le cas contraire (WS non java) , on peut générer l'interface et les classes de données qui vont avec en entrées/sorties via *wsimport* (en se débarrassant de tout le reste).

Invocation dynamique avec jdk et accès au WSDL

```
private static void testCalculeurServiceWithoutWsImport() {
    QName SERVICE_NAME = new QName("http://myws/", "CalculeurService");
    QName PORT_NAME = new QName("http://myws/", "CalculeurServicePort");

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

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

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

    /*interface*/ Calculateur calculateurWSProxy = (Calculateur)
        service.getPort(PORT_NAME, Calculateur.class);

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

Invocation dynamique avec CXF, sans accès WSDL

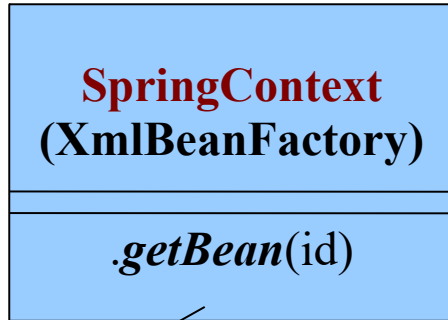
```
private static void testCalculeurServiceWithCxfAndWithoutWsImport() {  
    // noms qualifiés trouvés dans la description WSDL lorsqu'elle est accessible :  
    QName SERVICE_NAME = new QName("http://myws/", "CalculeurService");  
    QName PORT_NAME = new QName("http://myws/", "CalculeurServicePort");  
  
    Service service = Service.create(SERVICE_NAME); //javax.xml.ws.Service  
    // Soap 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);  
  
    /*interface*/ Calculateur caculateurWSProxy = (Calculateur)  
        service.getPort(PORT_NAME, Calculateur.class);  
  
    double tauxTvaReducitPct = caculateurWSProxy.getTauxTva("TauxReducit");  
}
```

Dans certains cas , la description WSDL peut ne pas être accessible à l'exécution (par exemple en production avec sécurité forte)

Présentation de CXF

- **CXF** (formée par **quelques ".jar"**) est une technologie open source facilitant le développement et la mise en oeuvre de Web Services en se basant sur l'api JAX-WS (dont le jdk n'est qu'une implémentation basique).
- 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
- **Facile à intégrer dans une application spring** (ou ..), **CXF** est une technologie "web-service" de référence qui existe depuis longtemps , qui est mature et qui vient récemment d'évoluer en version 3.
- Certains serveurs d'application (ex : Jboss 7.1.1) utilisent "CXF" en interne pour gérer les annotations **@WebService** sur des **EJB3 "@Stateless"** .

Spring + CXF (Services WEB)



créer ,
initialise et
met en relation

mySpringConf.xml

```
<beans>
  <!-- + configurations DataSource + DAO -->
  <bean id="s1" class="s.S1">
    <property name="dao" ref="dao1" />
  </bean>

  <jaxws:endpoint id="S1WebService"
    implementor="#s1" address="/s1" />
</beans>
```

(fin url service web)

nécessite spring.jar + cxf.jar + ...

*description
WSDL* ○
générée



*Accès
distants SOAP*

Accès locaux (depuis même jvm)



Paramétrage de "CxfServlet" dans WEB-INF/web.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>WEB-INF/classes/beans.xml</param-value>
    <!-- ou .... , chemin menant à la configuration spring -->
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
    </listener-class> <!-- initialisation de spring lors dès le démarrage webApp -->
</listener>

<servlet>
    <servlet-name>CXFServlet</servlet-name>
    <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
</servlet>    <!-- cxf....jar recherchés dans WEB-INF/lib ou ... -->
<servlet-mapping>
    <servlet-name>CXFServlet</servlet-name>
    <url-pattern>/services/*</url-pattern> <!-- ou autre que services/* →
    <!-- URL WS en "http://localhost:8080/myWebApp/services/serviceName" -->
</servlet-mapping>
```

Paramétrage "**spring**" des "**endPoints**" gérés par **Cxf**

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-servlet.xml" />

  <bean id="calculateurService_impl" class="service.CalculateurService" >
    <!-- <property name="...." ref="...." /> éventuelle injection de dépendance -->
  </bean>

  <jaxws:endpoint
    id="calculateurServiceEndpoint"
    implementor="#calculateurService_impl"
    address="/Calculateur" />
  <!-- pour fin d'url en /services/Calculateur et /services/Calculateur?wsdl -->
</beans>
```

Suite des paramétrage "**spring**" des "**endPoints**" Cxf

La syntaxe de l'attribut *implementor* est ici "**#idComposantSpring**"

Après avoir déployé l'application et démarré le serveur d'application (ex: *run as/ run on serveur*), il suffit de préciser une URL (volontairement incomplète) 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 d'un des services pris en charge par cxf , ...

Test "soap-ui" habituel pour continuer.

Remarque : En ajoutant le paramétrage **bindingUri=**

["http://www.w3.org/2003/05/soap/bindings/HTTP/"](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>
```

éventuelle Combinaison "spring + cxf" coté client

beans.xml

```
... <jaxws:client id="clientWsProxy"
    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 -->
```

Exemple d'utilisation:

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
...
public class SpringCxfClientWsTestApp {
    public static void main(String[] args) {
        ApplicationContext springContext = new
            ClassPathXmlApplicationContext(new String[] { "beans.xml" });

        /*interface*/ Calculateur calculateur = (Calculateur)
            springContext.getBean("clientWsProxy");
        System.out.println(calculateur.getTva(200, 19.6));
    }
}
```

Binary/base64 (pour soap) et MTOM

Cadre général et optimisation avec soap-over-http :

En règle générale, les données véhiculées au sein de l'enveloppe SOAP sont obligatoirement en mode texte (compatible avec xml).

Pour **véhiculer des données binaires quelconques** (*ex : fichier pdf, image jpeg, ...*) on peut dans tous les cas utiliser l'encodage **base64** (voir wikipédia). **Cependant, le temps de codage/décodage d'une longue partie d'un message en base64 peut être un frein non négligeable au niveau des performances.**

Dans le cas particulier "soap-over-http" , on peut utiliser une optimisation normalisée appelée **MTOM** qui consiste à véhiculer les données binaires à coté de l'enveloppe soap , en tant que pièces jointes (type mime **multipart**).

MTOM = Message Transmission Optimization Mechanism (for binary)

MTOM Upload Soap WS (1)

```
<jaxws:endpoint id="uploadfileEndpoint"
  implementor="tp.service.FileUploaderImpl"
  address="/uploadWs">
  <jaxws:properties>
    <entry key="mtom-enabled" value="true"/>
  </jaxws:properties>
</jaxws:endpoint>
```

```
package tp.data;

import javax.activation.DataHandler;

public class FileToUpload {

    private String fileName;
    private DataHandler fileDataHandler;

    //+get/set
}
```

MTOM Upload Soap WS (2)

```
package tp.service;  
import javax.jws.WebParam; import javax.jws.WebService;  
import tp.data.FileToUpload;
```

@WebService

```
public interface FileUploader {  
    void uploadFile(@WebParam(name="fileToUpload")  
                    FileToUpload fileToUpload);  
}
```

```
package tp.service;  
import java.io.* ;  
import javax.activation.DataHandler; import javax.jws.WebParam;  
import javax.jws.WebService;          import tp.data.FileToUpload;
```

@WebService(endpointInterface="tp.service.FileUploader")

```
public class FileUploaderImpl implements FileUploader {
```

```
    private String uploadDir = "/tmp/upload/";
```

```
    .../...
```


MTOM Upload Soap WS (3)

@Override

```
public void uploadFile(FileToUpload fileToUpload) {  
    DataHandler handler = fileToUpload.getFileDataHandler();  
    try {  
        InputStream is = handler.getInputStream();  
  
        OutputStream os = new FileOutputStream(  
            new File(uploadDir + fileToUpload.getFileName()));  
        byte[] b = new byte[100000];  
        int bytesRead = 0;  
        while ((bytesRead = is.read(b)) != -1) {  
            os.write(b, 0, bytesRead);  
        }  
        os.flush(); os.close(); is.close();  
  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

MTOM Upload Soap WS (4) – invocation coté "client"

```
public class FileUploaderClientSoapApp {  
    public static void main(String[] args) {  
        FileUploader uploader =new FileUploaderImplService().getFileUploaderImplPort();  
  
        FileToUpload file=new FileToUpload();  file.setFileName("maven-2014.pdf");  
  
        DataSource source = new FileDataSource(  
            new File("/home/formation/Bureau/docs/maven-2014.pdf"));  
        DataHandler dataHandler = new DataHandler(source);  
        try { file.setFileDataHandler(toBytes(dataHandler));  
        } catch (IOException e) { e.printStackTrace();  
        }  
        uploader.uploadFile(file);  
    }  
  
    private static final int INITIAL_SIZE = 1024 * 1024;  
    private static final int BUFFER_SIZE = 1024;  
  
    public static byte[] toBytes(DataHandler dh) throws IOException {  
        ByteArrayOutputStream bos = new ByteArrayOutputStream(INITIAL_SIZE);  
        InputStream in = dh.getInputStream();    byte[] buffer = new byte[BUFFER_SIZE];  
        int bytesRead;  
        while ( (bytesRead = in.read(buffer)) >= 0 ) {    bos.write(buffer, 0, bytesRead);    }  
        return bos.toByteArray();  
    }  
}
```

Appels "soap" complètement dynamiques

En combinant l'API JAX-WS , l'api complémentaire WSDL4J et un peu d'introspection java , on peut invoquer une méthode d'un service SOAP de manière entièrement dynamique (*à l'image des langages interprétés tels que php , javascript ou groovy*).

La classe utilitaire *generic.ws.util.client*.**DynReflectSoapClient** comporte la méthode clef suivante (à nombre d'arguments variable) :

```
public Object dynSoapCall(String wsUrl,String serviceName,  
                           String methodName,Object ... soapArgs) ;
```

Exemple d'utilisation:

```
DynReflectSoapClient dynSoapClient = new DynReflectSoapClient();  
String wsUrl="http://localhost:8080/wsCalculateur/services/calculateur";  
System.out.println("dynSoapCall result(3+5)="+  
    dynSoapClient.dynSoapCall(wsUrl,"tp.service.Calculateur",  
                                "addition", new Double(3),new Double(5)));
```

Approche "WSDL first" :

Lorsqu'un fichier WSDL important existe déjà et qu'il est considéré comme une norme ou une référence , il peut être important de coder une instance d'un nouveau service web dont la structure correspondra exactement à ce contrat "WSDL" .

La technologie "cxf" comporte de quoi générer un début de code d'implémentation de Web-service en précisant une URL de fichier WSDL existant :

wsdl2java = équivalent cxf de *wsimport* avec plus d'options telles que -server , -impl

wsdl2java peut également être lancé via script "ant" ou plugin "maven" .

Configuration maven du plugin "cxf-codegen-plugin"

<plugin>

<groupId>*org.apache.cxf***</groupId>** **<artifactId>***cxf-codegen-plugin***</artifactId>**

<version>*\${cxf.version}***</version>** *<!-- exemple : version 3.0.2 -->*

<executions>

<execution> **<id>**generate-sources**</id>**

<phase>generate-sources**</phase>**

<configuration>

<!-- <sourceRoot>\${project.build.directory}/generated/cxf</sourceRoot> -->

<wsdlOptions>

<wsdlOption>

<wsdl>*\${basedir}/src/main/resources/myService.wsdl***</wsdl>**

<extraargs>

<extraarg>-server**</extraarg>***<!-- facultatif : generer classe _Server/main -->*

<extraarg>-impl**</extraarg>** *<!-- indispensable : generer implementation -->*

</extraargs>

</wsdlOption>

</wsdlOptions>

</configuration>

<goals>

<goal>wsdl2java**</goal>**

</goals>

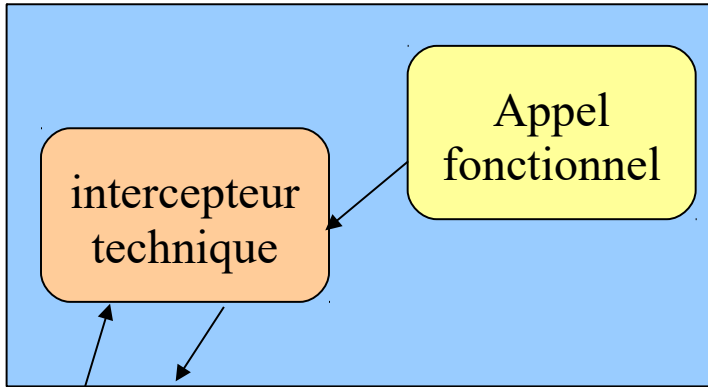
</execution>

</executions>

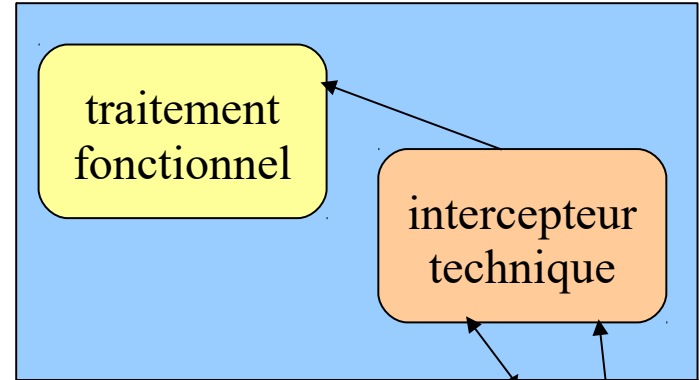
</plugin>

Aspects techniques gérés par *intercepteurs*

Appli. cliente



Appli. serveur



*Insère
infos auth.
et/ou crypte/signé*

http/soap

intermédiaire(s)

http/soap

*Vérifie
Auth. et/ou
décrypte*

NB: Les parties fonctionnelles ne voient pas les intercepteurs techniques (ceux-ci s'interposent de façon transparente).

Intercepteurs jax-ws gérés par "cxf"

Contrairement à des technologies purement java (de type "spring-aop" ou "intercepteur ejb3/cdi"), un intercepteur jax-ws doit intervenir dès les premiers traitements à appliquer sur une requête soap-over-http entrante → structure et configuration particulière.

Configuration d'un intercepteur jax-ws/cxf intégrée à spring :

```
<bean id="myCustomLogInterceptor"  
    class="generic.ws.util.interceptor.cxf.CxfLogInterceptor" />  
<bean id="loggingInInterceptor"  
    class="org.apache.cxf.interceptor.LoggingInInterceptor" /> <!--prédéfini-->  
  
<jaxws:endpoint id="calculateurEndPoint"  
    implementor="#calculateurService_impl" address="/calculateur" >  
    <jaxws:inInterceptors>  
        <ref bean="myCustomLogInterceptor"/>  
        <!-- <ref bean="loggingInInterceptor"/> -->  
    </jaxws:inInterceptors>  
</jaxws:endpoint>
```

Liste d'intercepteurs en entrée

*un commentaire suffit à désactiver
un intercepteur*

Exemple d'intercepteur (ici "log" requête "soap")

```
public class CxfLogInterceptor extends AbstractSoapInterceptor {
    protected Logger log = Logger.getLogger(getClass());

    public CxfLogInterceptor() {
        super(Phase.RECEIVE); // PHASE interception à préciser dans constructeur !!!
    }
    @Override
    public void handleMessage(SoapMessage message) throws Fault {
        InputStream is = message.getContent(InputStream.class);
        if (is != null) {
            CachedOutputStream bos = new CachedOutputStream();
            try { IOUtils.copy(is, bos);  bos.flush(); is.close();
                message.setContent(InputStream.class, bos.getInputStream());
                StringWriter swriter = new StringWriter();
                Transformer serializer = (TransformerFactory.newInstance()).newTransformer();
                serializer.transform(new StreamSource(bos.getInputStream())
                                     , new StreamResult(swriter));
                log.info(">>soap message received:" + swriter.toString());
                //System.out.println(">>soap message received:" + swriter.toString());
                bos.close();
            } catch (Exception e) { throw new Fault(e);
            }
        }
    }
}
```


Sécurité liée aux *services "WEB"*

- **Authentication du client** via (*username,password*) ou autre pour contrôler les accès aux services
 - > "*infos auth*" dans **entête Http** ou **entête SOAP** ou
 - > éventuelle authentication forte (double) avec WS-S .
- **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
- ...

Authentification du client invoquant un service WEB

Au moins 3
grands types
de solutions:

*dans entête http
(auth basic http)*

*(username,
password)
véhiculable
dans*

*dans entête soap
(ws-s)*

*paramètre(s) de la
méthode invoquée*

```
POST /SoapEndUrl
HTTP/1.1
Host: www.yyy.com
Content-type: text/xml
Content-Length: 524
.... : .....
```

```
<S:Envelope ... >
```

```
<S:Header>
```

```
...
```

```
</S:Header>
```

```
<S:Body>
```

```
<meth> ... </meth>
```

```
</S:Body>
```

```
</S:Envelope>
```

Approche "par jeton/token retransmis en paramètre"

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

* 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 .

* 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.

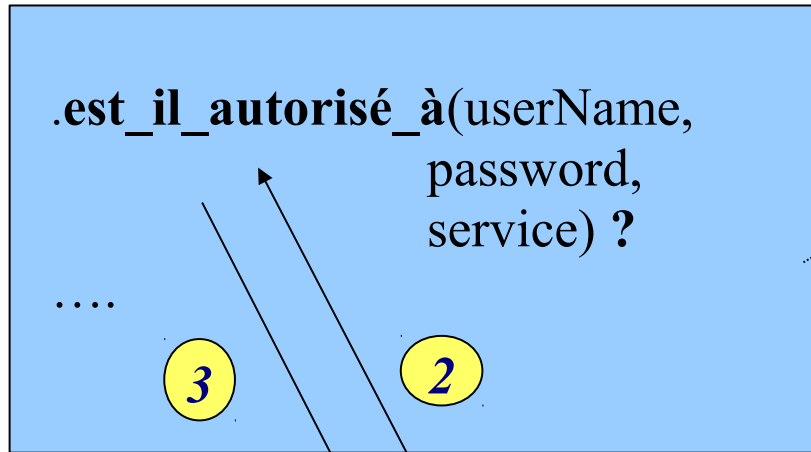
Variantes classiques:

inconvénient = mélange technique/fonctionnel

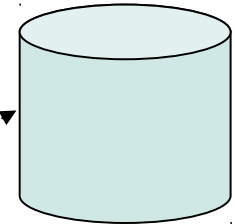
* repasser le jeton de sécurité/session au travers d'un argument plus large généralement appelé "**contexte**" (au sens "contexte technique" et non métier). Ce contexte pourra ainsi véhiculer d'autres informations techniques jugées pertinentes (ex: transactions , référence xy, ...).

* utiliser en plus SOAP over **HTTPS** pour crypter si besoin certaines informations échangées (ex: [username, password] ou plus).

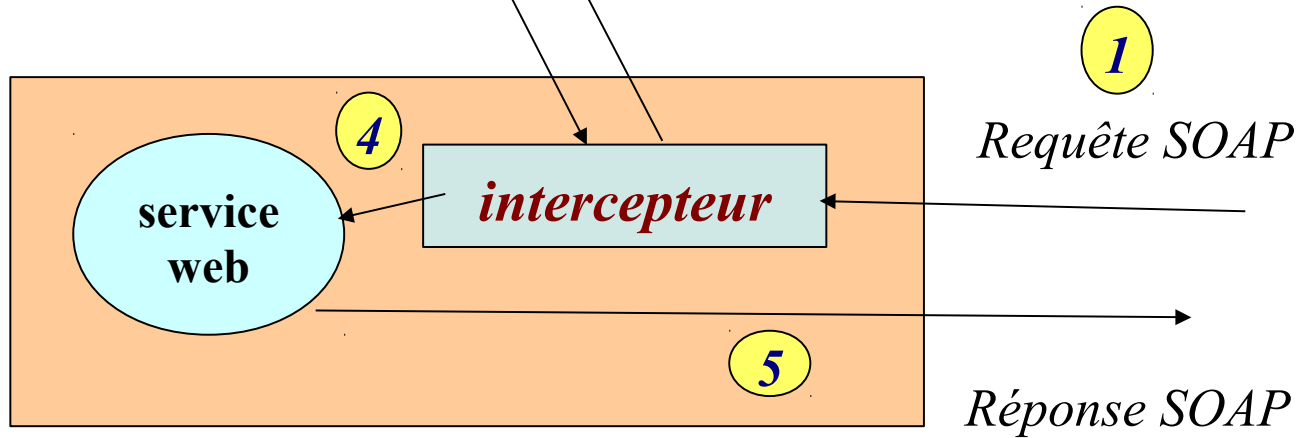
éventuel service d'habilitation



*authentications
& autorisations*



**sgbdr
et/ou ldap**



Application fonctionnelle

Authentication "basic http" coté client (via JAX-WS)

```
Calculateur calcProxy = serviceCalculateur.getCalculateurServicePort();  
// ou calcProxy = (Calculateur) service.getPort(PORT_NAME,  
Calculateur.class);  
javax.xml.ws.BindingProvider bp = (javax.xml.ws.BindingProvider)  
calcProxy;  
Map<String, Object> context = bp.getRequestContext();  
String userName = "powerUser" ;  
String password = "007" ;  
context.put(BindingProvider.USERNAME_PROPERTY, userName);  
context.put(BindingProvider.PASSWORD_PROPERTY, password);  
System.out.println(calcProxy.getTva(200, 19.6));
```

Ce code (à idéalement intégré au sein d'une classe utilitaire ou à placer dans un intercepteur du coté client) suffit à encoder l'information d'authentification (userName, password) en mode "**basic http auth**" et à placer cette information (*légèrement cryptée*) dans l'**entête http** de la requête envoyée au serveur.

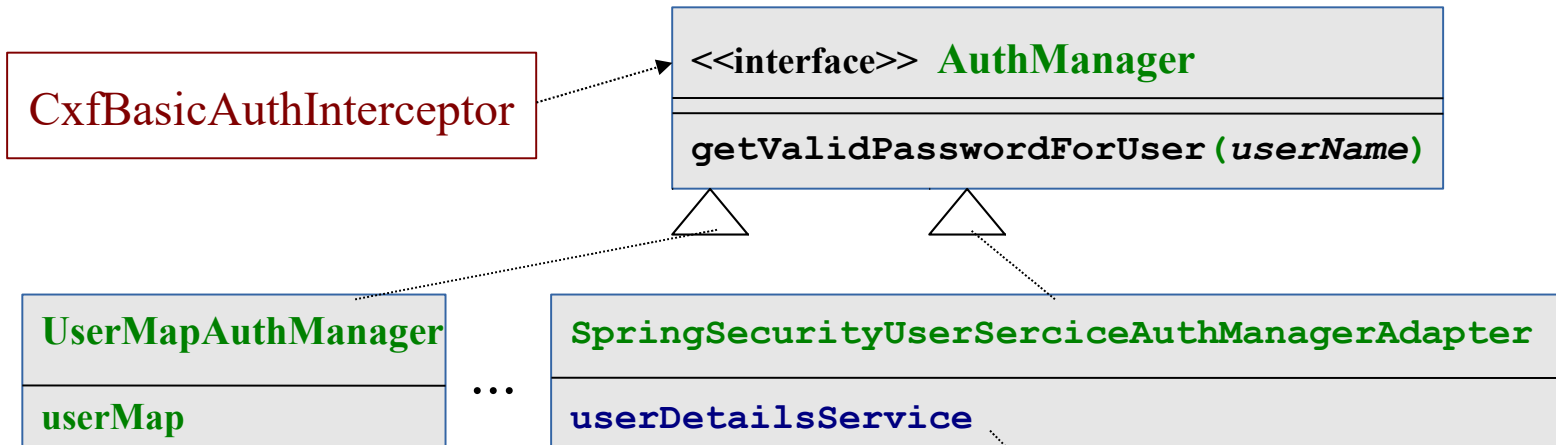
Authentication "basic http" coté serveur (intercepteur cxf)

```
<bean id="userMapAuthManager"
      class="generic.util.auth.local.UserMapAuthManager">
  <property name="users"><map>
    <entry key="user1" value="pwd1"/>
    <entry key="porwerUser" value="007"/>
  </map></property>
</bean>

<bean id="basicHttpSecurityInterceptor"
      class="generic.ws.util.interceptor.cxf.CxfBasicAuthInterceptorproperty name="authManager" ref="userMapAuthManager"/>
  <!-- test avec mode = "preemptive" dans soap-ui récent -->
</bean>

<jaxws:endpoint id="calculateurEndPoint"
  implementor="#calculateurService_impl" address="/calculateur" >
  <jaxws:inInterceptors>
    <ref bean="basicHttpSecurityInterceptor"/>
  </jaxws:inInterceptors>
</jaxws:endpoint>
```

Variations autour du gestionnaire d'authentification :



Spring-security-config.xml

```
<beans ... xmlns:security="http://www.springframework.org/schema/security" ... >
  <security:authentication-manager id="springSecurityAuthenticationManager">
    <security:authentication-provider>

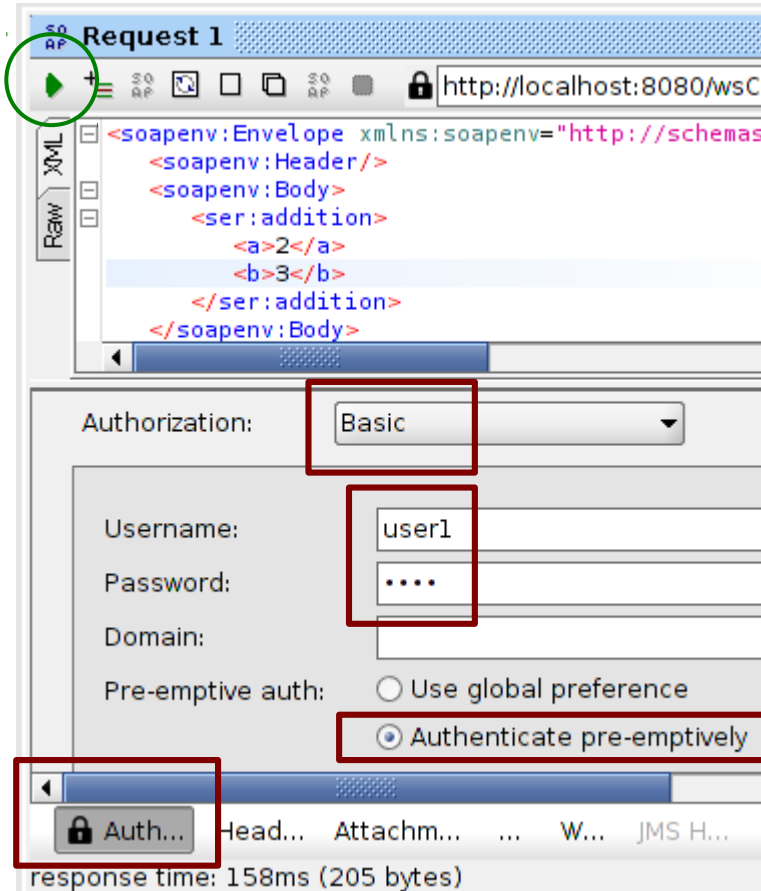
      <security:user-service id="simpleUserDetailsService">
        <security:user name="user1" password="pwd1" authorities="user" />
        <security:user name="powerUser" password="007" authorities="user" />
      </security:user-service>
    </security:authentication-provider>
  </security:authentication-manager> </beans>
```

*mini
standard
de
portée
spring*

Test d'authentification "basic http" avec soap-ui

Sans aucune information d'authentification envoyée -->

HTTP/1.1 401 Non-Autorisé



Encodage "basic http auth" :

Content-Type: text/xml; charset=UTF-8
SOAPAction: ""

Authorization: Basic dXNlcjE6cHdkMQ==

Content-Length: 264

Avec authentification valide :

HTTP/1.1 200 OK

```
<ns2:additionResponse xmlns:ns2="http://schemas.xmlsoap.org/soap/envelope/">
  <return>5.0</return>
</ns2:additionResponse>
```

Avec authentification invalide:

HTTP/1.1 403 Interdit
Server: Apache-Coyote/1.1
accept-encoding: gzip, deflate
Authorization: Basic dXNlcjE6cHdkMTc3
connection: Keep-Alive
host: localhost:8080

Limitations de l'approche "**basic http auth**"

Avantages de l'approche "**basic http auth**" :

- simple à mettre en œuvre → **interopérabilité aisée** (java, php, ...) .
- assez bonne séparation "entête http technique" , " corps fonctionnel"
- conseillé par l'organisme "**Web Service Interoperability**" pour les cas simples
- standard http , cryptage plus poussé possible via https.

inconvénients de l'approche "**basic http auth**" :

- niveau d'authentification limitée (simple mot de passe)
- **limité à un transport http** : *si l'enveloppe soap est retransmise via SMTP ou JMS entre deux intermédiaires , l'information d'authentification (trop lié à http) n'est alors pas retransmise.*

Authentication "ws-s" coté serveur (intercepteur cxf)

```
<bean id="myPasswordCallback" class="generic.ws.util.auth.wss.PasswordCallback" >
    <property name="authManager" ref="userMapAuthManagerViaSpringSecurity"/>
</bean>

<bean id="wssSecurityInterceptor"
    class="org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor" >
    <constructor-arg> <map>
        <entry key="action" value="UsernameToken"/>
        <!-- <entry key="passwordType" value="PasswordDigest"/> --> <!-- text by default -->
        <!-- <entry key="signaturePropFile" value="..." /> -->
        <entry key="passwordCallbackRef"> <ref bean="myPasswordCallback"/></entry>
    </map> </constructor-arg>
</bean>

<jaxws:endpoint id="calculateurEndPoint"
    implementor="#calculateurService_impl" address="/calculateur" >
    <jaxws:inInterceptors>
        <ref bean="wssSecurityInterceptor"/>
    </jaxws:inInterceptors>
</jaxws:endpoint>
```

PasswordCallback Handler for CXF WS-S

```
import javax.security.auth.callback.CallbackHandler;
import org.apache.wss4j.common.ext.WSPasswordCallback; // cxf 3
//import org.apache.ws.security.WSPasswordCallback; // cxf 2
...
public class PasswordCallback implements CallbackHandler
{
    private AuthManager authManager;
    ....
    public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException
    {
        WSPasswordCallback pc = (WSPasswordCallback) callbacks[0];
        // WSPasswordCallback is an indirect dependency of cxf-rt-ws-security

        String username = pc.getIdentifier();
        String validPwdToCompare = authManager.getValidPasswordForUser(username);
        pc.setPassword(validPwdToCompare);
        /* if (pc.getIdentifier().equals("user1")) {
            pc.setPassword("pwd1");
        }
        else if (pc.getIdentifier().equals("user2")) {
            pc.setPassword("pwd2");
        }
        */
    }
}
```

Test d'authentification "WS-S" avec soap-ui (1/2)

Sans aucune information d'authentification --> HTTP/1.1 500 Erreur Interne de Servlet

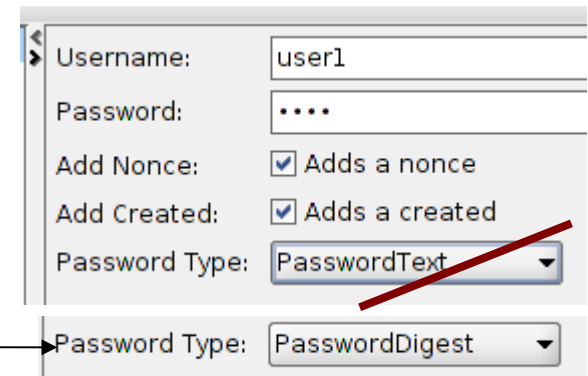
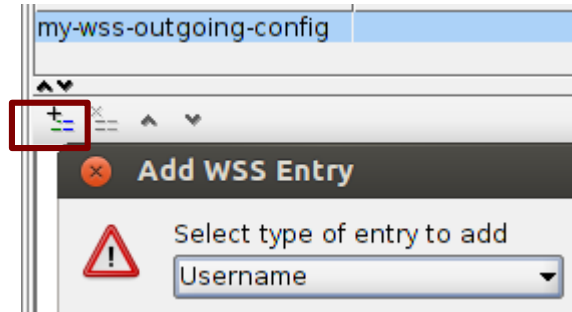
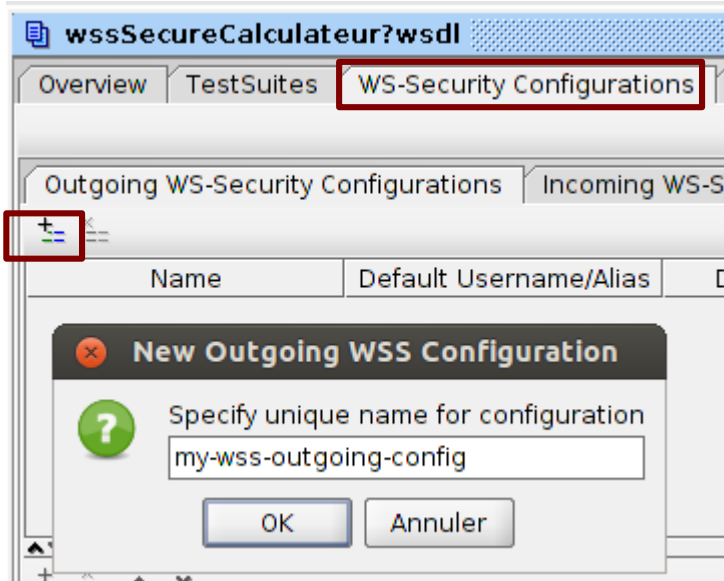
<soap:Fault>

<faultcode xmlns:ns1="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-secext-1.0.xsd">ns1:InvalidSecurity</faultcode>

<faultstring>An error was discovered processing wsse:Security header.</faultstring>

</soap:Fault>

Configuration à préparer sur projet "soap-ui" :



pour un cryptage du mot de passe

Test d'authentification "WS-S" avec soap-ui (2/2)

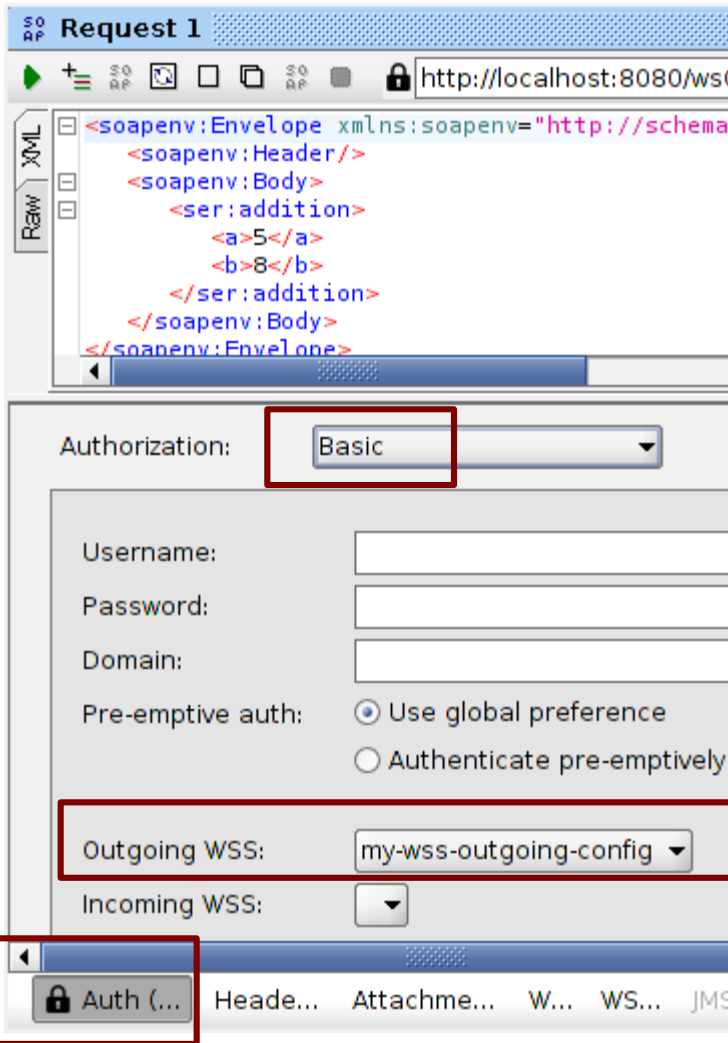
Encodage "ws-s" dans entête soap :

```
<soapenv:Header>
<wsse:Security xmlns:wsse="
  http://docs.oasis-open.org/wss/...." ...>
  <wsse:UsernameToken wsu:Id=
    "UsernameToken-D5DA21F60D....">
    <wsse:Username>user1</wsse:Username>
    <wsse:Password Type="....#PasswordText">
      pwd1</wsse:Password> ...
  </wsse:UsernameToken>
</wsse:Security>
</soapenv:Header>
```

NB : fermer et ré-ouvrir "Request1"
si "my-wss-outgoing-config"
n'apparaît pas !

Avec authentification invalide:

```
<soap:Fault>
  <faultcode>ns1:FailedAuthentication
  </faultcode>
  <faultstring>The security token could not
    be authenticated or authorized</faultstring>
</soap:Fault>
```



Spécification de l'**authentification wss coté client** avec **JAX-WS**

* Beaucoup de lignes de code (assez techniques mais compréhensibles)
à cacher dans des classes utilitaires !!!

* Au moins 2 types de
code possibles :

- avec uniquement le jdk
- avec l'aide de cxf

```
WSSUsernameTokenSecurityHandler  
( implements SOAPHandler<...> )  
  
login  
pwd  
  
.handleMessage (SOAPMessageContext c)
```

```
CxfWSSUsernameTokenSecurityUtil  
  
login  
pwd  
ClientPasswordCallback (classe imbriquée)  
  
void setWssSecurityViaCxf (Object wsProxy)
```

Manipulation
directe (de bas
niveau) de
l'entête soap

En s'appuyant sur `org.apache.cxf.ws.security.wss4j.WSS4JOutInterceptor`

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").

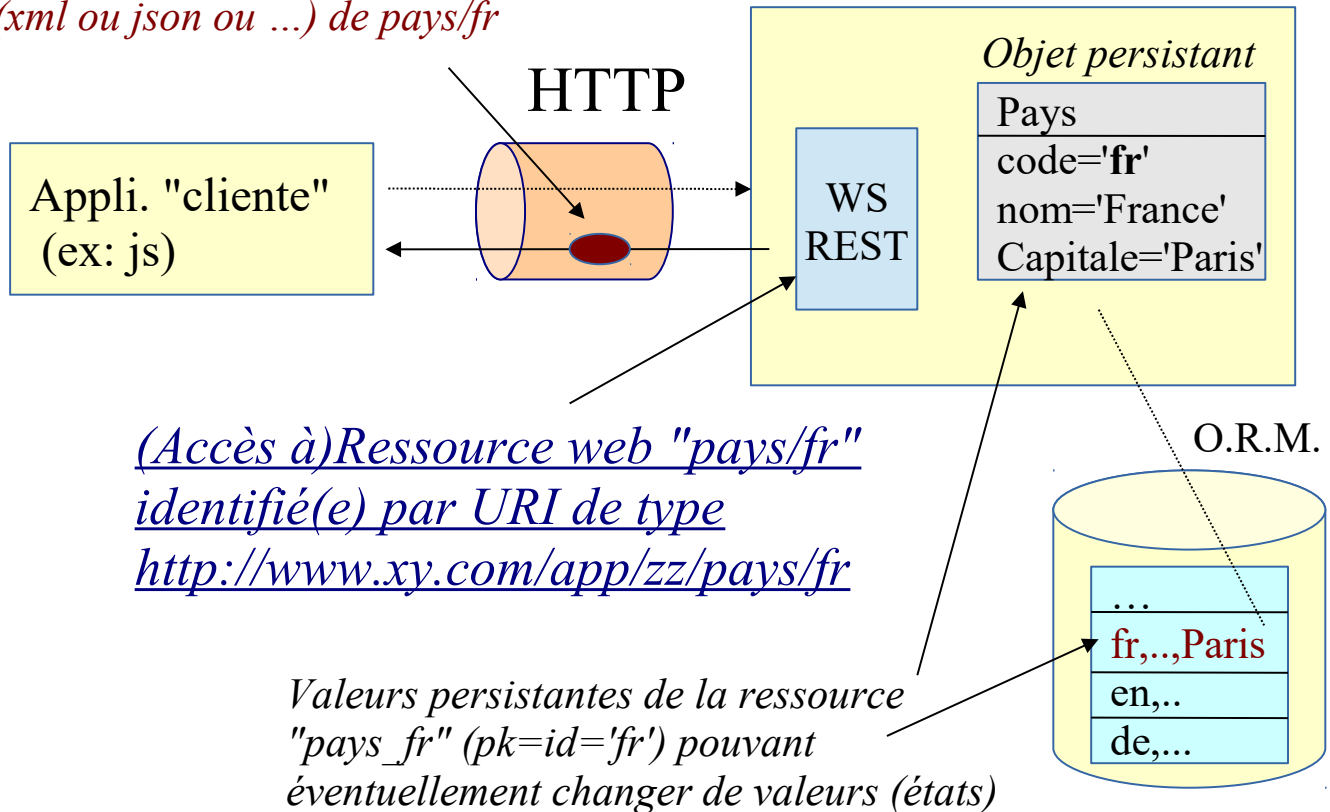
Les composants de l'architecture REST manipulent ces ressources en **transférant à travers le réseau** (via HTTP) des **représentations de ces ressources**.

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*

Appli. "serveur" (ex: java, php)



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

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

Les types de données valables sont :

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

Exemples basiques d'url fonctionnelles

Dans le cas où l'on utilise les conventions "REST" pour déclencher des appels de méthodes à distance on pourra éventuellement utiliser des URLs de ce genre :

<http://localhost:8080/xx/yy/euroToFranc/15>

avec une réponse simple en text/plain *98.39355*

[http://localhost:8080/xx/services/rest/calculateur/addition?a=5&b=6](http://localhost:8080/xx/services/rest/calculateur/<u>addition?a=5&b=6</u>)

retournant *11*

Angular-js (positionnement)

Coté client (navigateur)

Technologie http
quelconque coté serveur
(ex : java, .net , php ,...)

The diagram illustrates the positioning of Angular.js in a client-server architecture. On the left, the 'Coté client (navigateur)' (Client side) is shown, containing a 'Page html téléchargée' (Downloaded HTML page). Inside this page, three components are stacked: 'Vue html/dom+css' (View), 'Automatismes angular-js (lib)' (Angular.js library), and 'Ctrl+Modèle js/json' (Controller/Model). Arrows indicate the flow of data and control between these components. On the right, the 'Technologie http' (HTTP technology) is shown, which connects the client to the 'quelconque coté serveur' (any server side). The server side is represented by a box containing 'WS_REST (json)' (Web Service REST) and 'Accès données (CRUD, ...)' (Data Access). The 'WS_REST' component is connected to the 'Ctrl+Modèle' on the client via an 'http' connection. The 'Accès données' component is connected to a database cylinder. The 'WS_REST' component is also connected to the 'Accès données' component. The 'stateless' label is placed near the database, indicating that the server does not store state. The 'Scope/session en local' label is placed near the client, indicating that the client manages its own state.

Page html téléchargée

Ctrl+Modèle
js/json

Automatismes
angular-js (lib)

Vue
html/dom+css

http

WS_REST
(json)

Accès données
(CRUD , ...)

stateless

Scope/session
en local

Angular-js (positionnement)

Coté client (navigateur)

Technologie http
quelconque coté serveur
(ex : java, .net , php ,...)

The diagram illustrates the positioning of Angular.js in a client-server architecture. On the left, the 'Coté client (navigateur)' (Client side) is shown, containing a 'Page html téléchargée' (Downloaded HTML page). Inside this page, three components are stacked: 'Vue html/dom+css' (View), 'Automatismes angular-js (lib)' (Angular.js library), and 'Ctrl+Modèle js/json' (Controller/Model). Arrows indicate the flow of data and control between these components. On the right, the 'Technologie http' (HTTP technology) is shown, which connects the client to the 'quelconque coté serveur' (any server side). The server side is represented by a box containing 'WS_REST (json)' (Web Service REST) and 'Accès données (CRUD, ...)' (Data Access). The 'WS_REST' component is connected to the 'Ctrl+Modèle' on the client via an 'http' connection. The 'Accès données' component is connected to a database cylinder. The 'WS_REST' component is also connected to the 'Accès données' component. The 'stateless' label is placed near the database, indicating that the server does not store state. The 'Scope/session en local' label is placed near the client, indicating that the client manages its own state.

Page html téléchargée

Ctrl+Modèle
js/json

Automatismes
angular-js (lib)

Vue
html/dom+css

http

WS_REST
(json)

Accès données
(CRUD , ...)

stateless

Scope/session
en local

Angular-js (positionnement)

Coté client (navigateur)

Technologie http
quelconque coté serveur
(ex : java, .net , php ,...)

The diagram illustrates the positioning of Angular.js in a client-server architecture. On the left, the 'Coté client (navigateur)' (Client side) is shown, containing a 'Page html téléchargée' (Downloaded HTML page). Inside this page, three components are stacked: 'Vue html/dom+css' (View), 'Automatismes angular-js (lib)' (Angular.js library), and 'Ctrl+Modèle js/json' (Controller/Model). Arrows indicate the flow of data and control between these components. On the right, the 'Technologie http' (HTTP technology) is shown, which connects the client to the 'quelconque coté serveur' (any server side). The server side is represented by a box containing 'WS_REST (json)' (Web Service REST) and 'Accès données (CRUD, ...)' (Data Access). The 'WS_REST' component is connected to the 'Ctrl+Modèle' on the client via an 'http' connection. The 'Accès données' component is connected to a database cylinder. The 'WS_REST' component is also connected to the 'Accès données' component. The 'stateless' label is placed near the database, indicating that the server does not store state. The 'Scope/session en local' label is placed near the client, indicating that the client manages its own state.

Page html téléchargée

Ctrl+Modèle
js/json

Automatismes
angular-js (lib)

Vue
html/dom+css

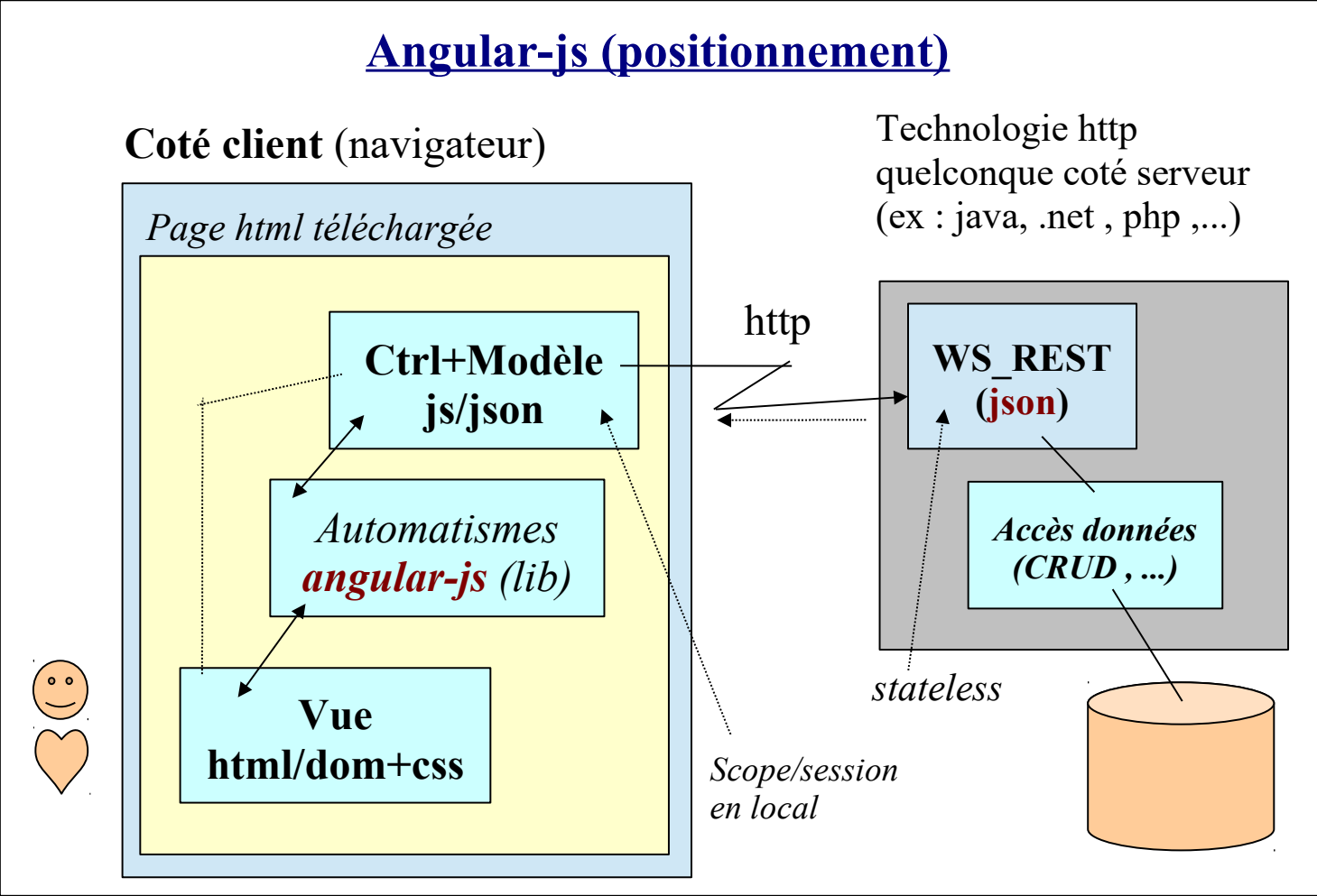
http

WS_REST
(json)

Accès données
(CRUD , ...)

stateless

Scope/session
en local



Angular-js (positionnement)

Coté client (navigateur)

Technologie http
quelconque coté serveur
(ex : java, .net , php ,...)

The diagram illustrates the positioning of Angular.js in a client-server architecture. On the left, the 'Coté client (navigateur)' (Client side) is shown, containing a 'Page html téléchargée' (Downloaded HTML page). Inside this page, three components are stacked: 'Vue html/dom+css' (View), 'Automatismes angular-js (lib)' (Angular.js library), and 'Ctrl+Modèle js/json' (Controller/Model). Arrows indicate the flow of data and control between these components. On the right, the 'Technologie http' (HTTP technology) is shown, which connects the client to the 'quelconque coté serveur' (any server side). The server side is represented by a box containing 'WS_REST (json)' (Web Service REST) and 'Accès données (CRUD, ...)' (Data Access). The 'WS_REST' component is connected to the 'Ctrl+Modèle' on the client via an 'http' connection. The 'Accès données' component is connected to a database cylinder. The 'WS_REST' component is also connected to the 'Accès données' component. The 'stateless' label is placed near the database, indicating that the server does not store state. The 'Scope/session en local' label is placed near the client, indicating that the client manages its own state.

Page html téléchargée

Ctrl+Modèle
js/json

Automatismes
angular-js (lib)

Vue
html/dom+css

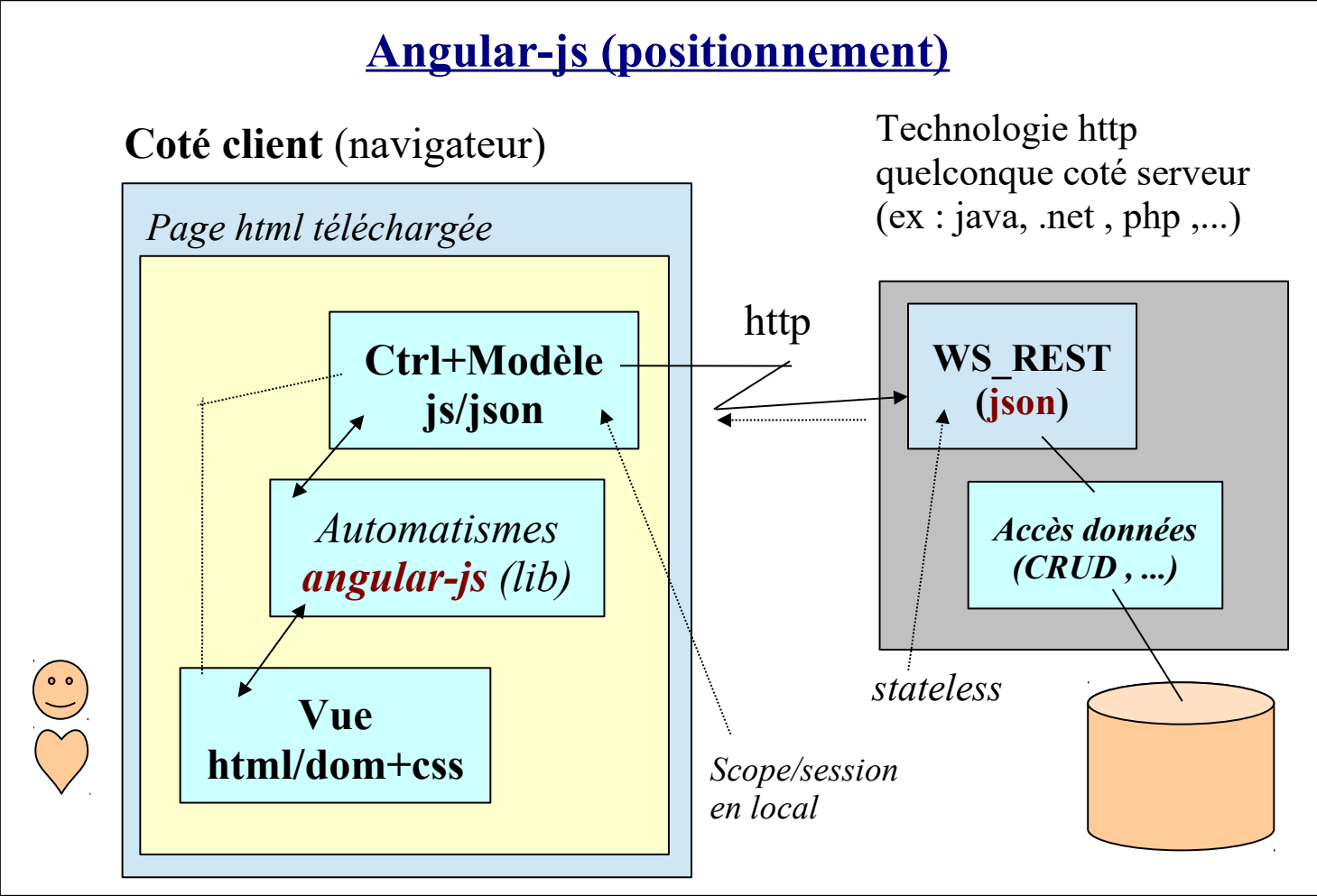
http

WS_REST
(json)

Accès données
(CRUD , ...)

stateless

Scope/session
en local



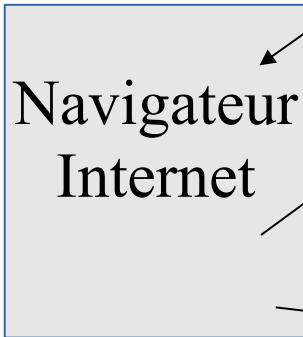
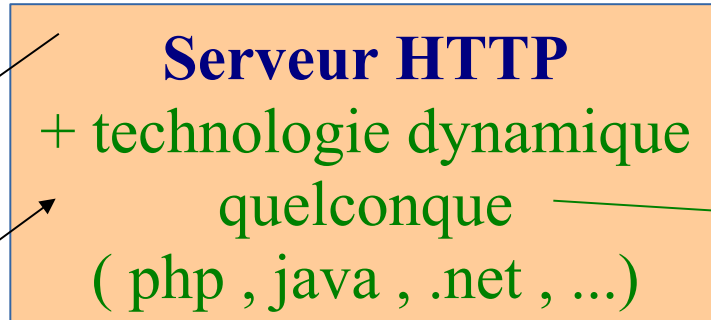
Conventions "angular-js" sur URL des ressources REST

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)
	DELETE			

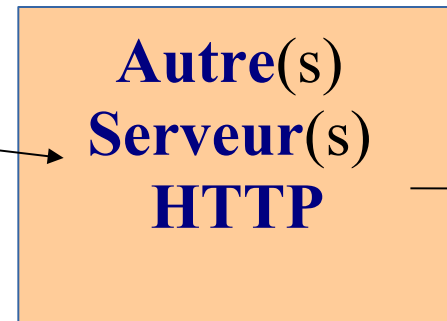
Cadre des appels "ajax" vers services REST

www.xy.com

① téléchargement
pages "angular-js"



② appels de Web-
services REST



Appels Ajax (XmlHttpRequest)
vers autres domaines
par défaut interdits sauf si paramétrage
"Cross-origin resource sharing (CORS)"

www.zz.com

JAX-RS = Api Java normalisée pour services REST

Les principales implémentations de JAX-RS sont :

- **Jersey**, implémentation de référence de SUN (bien : simple/léger et efficace + coté 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** (???)

Principales annotations de JAX-RS

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 du service et/ou sur une de ses méthodes
- **@GET, @PUT, @POST, @DELETE** : définit le mode HTTP (selon logique CRUD des méthodes)
- **@Produces** spécifie le ou les Types MIME de la réponse du service
- **@Consumes** : spécifie le ou les Types MIME acceptés en entrée du service

Classe de données compatible JAX-RS

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

/* pour préciser balise xml englobante et pour jaxb2 */
@XmlRootElement(name = "user")
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 Web-service JAX-RS

```
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") *//partie de l'url (préfixe commun à toutes méthodes)*

@Produces("application/json") *//par défaut pour toutes les méthodes
//de la classe*

//+ éventuel @Named selon contexte (Spring ou CDI/JEE6 ou ...)

```
public class ServiceImpl {
    ...
}
```

pas d'interface obligatoire à obligatoirement implémenter .

implémentation JAX-RS de recherche par id

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

Implémentation JAX-RS de recherches multiples

```
public class ServiceImpl {    ...
```

```
    private Collection<User> getAllUsers() {  
        return users.values();  
    }
```

```
    @GET
```

```
    @Path("users")
```

```
    // 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) { ...; }  
        else { return getAllUsers() ; }  
    }
```

```
    @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) !!!  
    }
```

implémentation JAX-RS d'envoi de données (v1)

```
public class ServiceImpl {    ...  
    @POST    // (REST recommande fortement POST pour des ajouts )  
    @Path("users")  
    // pour 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();  
    }  
  
    @PUT  
    @Path("users")  
    public Response updateUser(@FormParam("name")String newName,  
                                @FormParam("id")Integer id){  
        User user = users.get(id);  
        if(user!=null){  
            user.setName(newName);  
            return Response.status(Status.OK).build();  
        }  
        else return Response.status(Status.NOT_FOUND).build();  
    }  
}
```

Retour en text/plain pour opérations élémentaires

```
@GET
@Path("/euroToFranc/{s}")
@Produces("text/plain")
public double euroToFranc(@PathParam("s")double s){
    return s*6.55957 ;
}
```

http://localhost:8080/xx/yy/euroToFranc/15 ---> 98.39355

@Produces("application/xml") ou **@Produces("application/json")**

Combiné avec code java unique

@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);  
}
```

Pour que le framework JAX-RS
puisse automatiquement générer et retourner

```
<?xml version="1.0" encoding="UTF-8"  
    standalone="yes"?>  
<user>  
  <id>1</id>  
  <name>foo</name>  
</user>
```

Conversion automatique java → xml

```
{  
  "id" : 1,  
  "name" : "foo"  
}
```

*Conversion automatique
java → json*

@Produces et @Consumes en "application/json" (pour Angular-js , ...)

@Path("/json/")

@Produces("application/json")

@Consumes("application/json")

public class **ProductRestJsonService** {

...

private ProductService **prodService** = new ProductService(); *//may be injected by spring*

@POST *//POST --> save/create or saveOrUpdate dans logique "Angular-Js"*

@Path("products/{id}")

//p est passé comme un seul bloc (en mode json) dans le corps/body de la requête

//exemple: { id: 1 , name='xxx' , label='yyy' , ...}

public **Product** **saveOrUpdateProduct**(**@PathParam("id")**Long id,**Product** p){

 if(id!= null){

 prodService.updateProduct(p);

 }

 else{

 Long newId= prodService.addNewProduct(p);


 p.setId(newId); *//auto_incr id/pk*

 }

 return p;

 }

}



@Consumes permet une conversion automatique **json → java** en entrée .

Configuration de JAX-RS intégrée à un projet JEE6/CDI

Lorsque JAX-RS est utilisé au sein d'un projet JEE6/CDI (par exemple avec ou sans "EJB3" pour JBoss7 ou bien pour Tomcat EE) , on peut configurer la partie intermédiaire des URL "rest" et les classes java à prendre en charge via une configuration ressemblant à la suivante :

```
package tp.web.rest; //ou autre
import java.util.HashSet; import java.util.Set;
import javax.ws.rs.ApplicationPath; import javax.ws.rs.core.Application;
```

```
//pour url en http://localhost:8080/myWebApp/services/rest + @Path() java
@ApplicationPath("/services/rest")
public class MyRestApplicationConfig extends Application {
```

```
    @Override
```

```
    public Set<Class<?>> getClasses() {
```

```
        final Set<Class<?>> classes = new HashSet<Class<?>>();
```

```
        // register root resource(s):
```

```
        classes.add(XyServiceRest.class);
```

```
        classes.add(ServiceRest2.class);
```

```
        return classes;
```

```
    }
```

```
//Ceci fonctionne avec des classes java (ex: XyServiceRest) annotées par "@Named"
```

```
// et avec beans.xml présent dans WEB-INF .
```

Configuration de JAX-RS avec CXF intégré dans Spring (1/2)

Beaucoup de versions de CXF utilisent par défaut "jettison" en interne pour générer du "json", ce qui pose des problèmes de compatibilité avec Angular-Js (ou ...).

Il est conseillé de configurer CXF pour qu'il utilise "jackson" à la place de "jettison".

CXF 2.x est compatible avec jackson 1.9 (*org.codehaus.jackson*)

CXF 3.x est plutôt compatible avec jackson 2.x (*com.fasterxml.jackson*)

WEB-INF/web.xml (*spring+cxf*) :

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>WEB-INF/classes/beans.xml</param-value>
    <!-- ou .... , chemin menant à la configuration spring -->
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
    </listener-class> <!-- initialisation de spring lors dès le démarrage webApp -->
</listener>

<servlet>
    <servlet-name>CXFServlet</servlet-name>
    <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
</servlet>    <!-- cxf...jar recherchés dans WEB-INF/lib ou ... -->
<servlet-mapping>
    <servlet-name>CXFServlet</servlet-name>
    <url-pattern>/services/*</url-pattern> <!-- ou autre que services/* →
    <!-- URL WS en "http://localhost:8080/myWebApp/services/..." -->
</servlet-mapping>
```

Configuration de JAX-RS avec CXF intégré dans Spring (2/2)

beans.xml (*spring+cxf*)

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:jaxws="http://cxf.apache.org/jaxws" xmlns:jaxrs="http://cxf.apache.org/jaxrs"
        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
http://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd">
    <import resource="classpath:META-INF/cxf/cxf.xml" />
    <bean id='jacksonJsonProvider' class='com.fasterxml.jackson.jaxrs.json.JacksonJaxbJsonProvider' />
    <bean id='jacksonXmlProvider' class='com.fasterxml.jackson.jaxrs.xml.JacksonJaxbXMLProvider' />

    <!-- url complete de type "http://localhost:8080/mywebapp/services/rest/myservice/users/"
         avec "services" associe à l'url-pattern de CxfServlet dans web.xml
         et myservice/users associé aux valeurs de @Path() de la classe java et des méthodes -->
    <jaxrs:server id="myRestServices" address="/rest">
        <jaxrs:providers>
            <ref bean='jacksonJsonProvider' /> <ref bean='jacksonXmlProvider' />
        </jaxrs:providers>
        <jaxrs:serviceBeans>
            <ref bean="serviceRestImpl" />         <!-- <ref bean="service2Impl" /> -->
        </jaxrs:serviceBeans>
    </jaxrs:server>

    <bean id="serviceRestImpl" class="service.ServiceRestImpl" /> </beans>
```

Configuration maven type pour cxf (dans pom.xml)

```
<properties>
```

```
  <org.springframework.version>4.1.1.RELEASE</org.springframework.version>
```

```
  <org.apache.cxf.version>3.0.2</org.apache.cxf.version>
```

```
</properties>
```

```
<dependencies>
```

```
  <dependency>
```

```
    <groupId>org.apache.cxf</groupId>
```

```
    <artifactId>cxf-api</artifactId>
```

```
    <version>${org.apache.cxf.version}</version>
```

```
  </dependency>
```

```
  <!-- idem pour les artifactId suivants :
```

```
  cxf-rt-frontend-jaxrs , cxf-rt-frontend-jaxws
```

```
  cxf-rt-ws-security ,    cxf-rt-transport-http    -->
```

```
  <dependency>
```

```
    <groupId>com.fasterxml.jackson.jaxrs</groupId>
```

```
    <artifactId>jackson-jaxrs-json-provider</artifactId>
```

```
  <!-- <artifactId>jackson-jaxrs-xml-provider</artifactId> -->
```

```
    <version>2.2.3</version>
```

```
  </dependency>
```

```
</dependencies>
```

Configuration de JAX-RS avec jersey

Dans pom.xml :

```
<dependency>
  <groupId>org.glassfish.jersey.containers</groupId>
  <artifactId>jersey-container-servlet-core</artifactId> <version>2.11</version>
</dependency>
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-json-jackson</artifactId> <version>2.11</version>
</dependency>
```

Dans WEB-INF/web.xml (jersey) :

```
<servlet>
  <servlet-name>Jersey_REST_Service</servlet-name>
  <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
  <init-param>
    <param-name>jersey.config.server.provider.packages</param-name>
    <!-- package java avec classes à prendre en charge -->
    <param-value>tp.service.rest</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>Jersey_REST_Service</servlet-name>
  <url-pattern>/services/rest/*</url-pattern>
</servlet-mapping>
```

Appel de webServices REST en java via l'API "**httpclient**" (1)

Pour le coté client des services web REST en java on peut utiliser une **API** open-source du monde *apache* spécialisée dans les appels http (en modes GET, POST,PUT, DELETE) : **httpclient** (de httpcomponents).

```
<dependency>
  <groupId>org.apache.httpcomponents</groupId>
  <artifactId>httpclient</artifactId><!-- indirectly httpcore -->
  <version>4.2.1</version>
</dependency>
```

```
package client;

import java.net.URI;
import org.apache.http.HttpResponse;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.client.utils.URIBuilder;
import org.apache.http.impl.client.DefaultHttpClient;
import org.apache.http.util.EntityUtils;

public class CalculeurClientRestApp {

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


Appel de webServices REST en java via l'API "**httpClient**" (2)

```
try {
    String restAppPart = "/wsCalculateur/services/rest";
    URIBuilder builder = new URIBuilder();
    builder.setScheme("http").setHost("localhost").setPort(8080)
        .setPath(restAppPart + "/calculateur/addition")
        .setParameter("a", "5")
        .setParameter("b", "6");
    URI uri = builder.build();
    String url5plus6=uri.toString();
    System.out.println("REST GET URL="+url5plus6);

    HttpClient httpClient = new DefaultHttpClient();
    HttpGet httpget = new HttpGet(url5plus6);
    HttpResponse response = httpClient.execute(httpget);

    String res5plus6=EntityUtils.toString(response.getEntity());

    System.out.println("5+6="+res5plus6);
} catch (Exception e) {
    e.printStackTrace();
}
```

Code brut (à encapsuler dans des classes utilitaires) !

Appel de webServices REST en java via l'API "[httpclient](#)" (3)

```
public static void testCreateNewDevise(String name,String change) {
    try {
        ...
        HttpPost httppost = new HttpPost(urlCreateDevises);

        List<NameValuePair> formparams = new ArrayList<NameValuePair>();
        formparams.add(new BasicNameValuePair("name", name));
        formparams.add(new BasicNameValuePair("change", change));
        UrlEncodedFormEntity entity=new UrlEncodedFormEntity(formparams,"UTF-8");
        httppost.setEntity(entity);
        HttpResponse response = httpclient.execute(httppost);
        System.out.println("response:"+response.toString());
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public static void testUpdateDevise(String name,String change) {
    ...
    HttpPut httpput = new HttpPut(urlUpdateDevises);
    ...
    UrlEncodedFormEntity entity =new UrlEncodedFormEntity(formparams,"UTF-8");
    httpput.setEntity(entity);
    HttpResponse response = httpclient.execute(httpput);
    System.out.println("response:"+response.toString());
    ...
}
```

Appel de webServices REST en java via l'API "**httpclient**" (4)

```
public static void testDeleteDevise(String name) {  
    String restAppPart = "/wsCalculateur/services/rest";  
    URIBuilder builder = new URIBuilder();  
    builder.setScheme("http").setHost("localhost").setPort(8080)  
        .setPath(restAppPart + "/devises/delete/" + name);  
    URI uri = builder.build();  
    String urlDeleteDevises=uri.toString();  
    HttpClient httpclient = new DefaultHttpClient();  
    HttpDelete httpdelete = new HttpDelete(urlDeleteDevises);  
    HttpResponse response = httpclient.execute(httpdelete);  
  
    System.out.println("response:"+response.toString());  
    ...  
}
```

Autre possibilité : L'implémentation "**Jersey**" de l'api "jax-rs" offre quelques classes permettant d'invoquer un service REST (coté "**client**").

Orchestration de services

(bpmn, bpel , userTask)

Orchestration ou chorégraphie de services

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

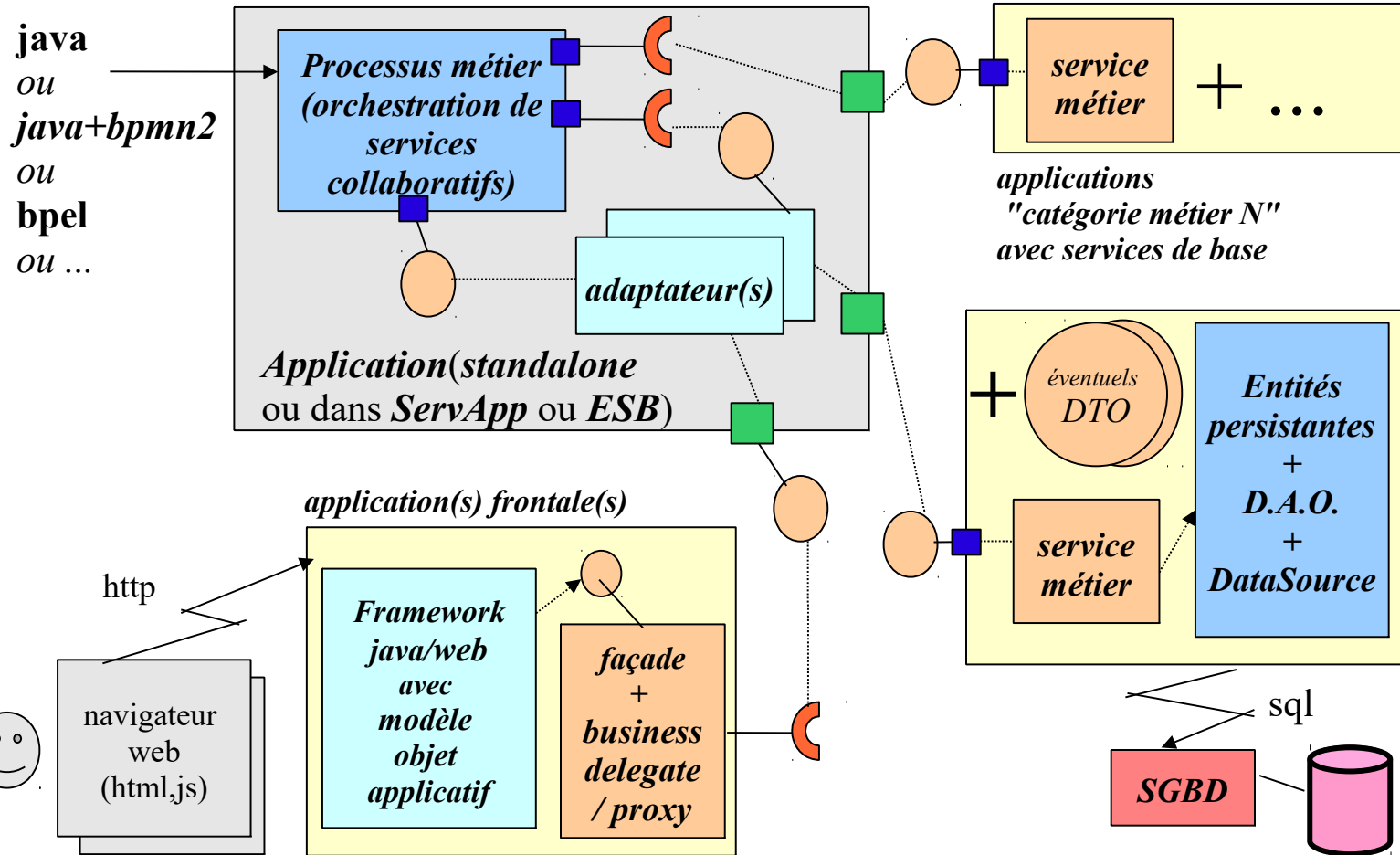
Un service de haut niveau pilote des services de bas niveaux en invoquant des opérations dans un ordre bien établi et/ou selon une logique conditionnelle. Le "chef d'orchestre" peut cependant prendre en compte certains événements.

Le terme de **chorégraphie** (SOA/services) correspond en plus à une **interaction plus étroite/collaborative** où chaque membre **agit de façon plus spontanée (idéalement pas trop dé-synchronisée)**.

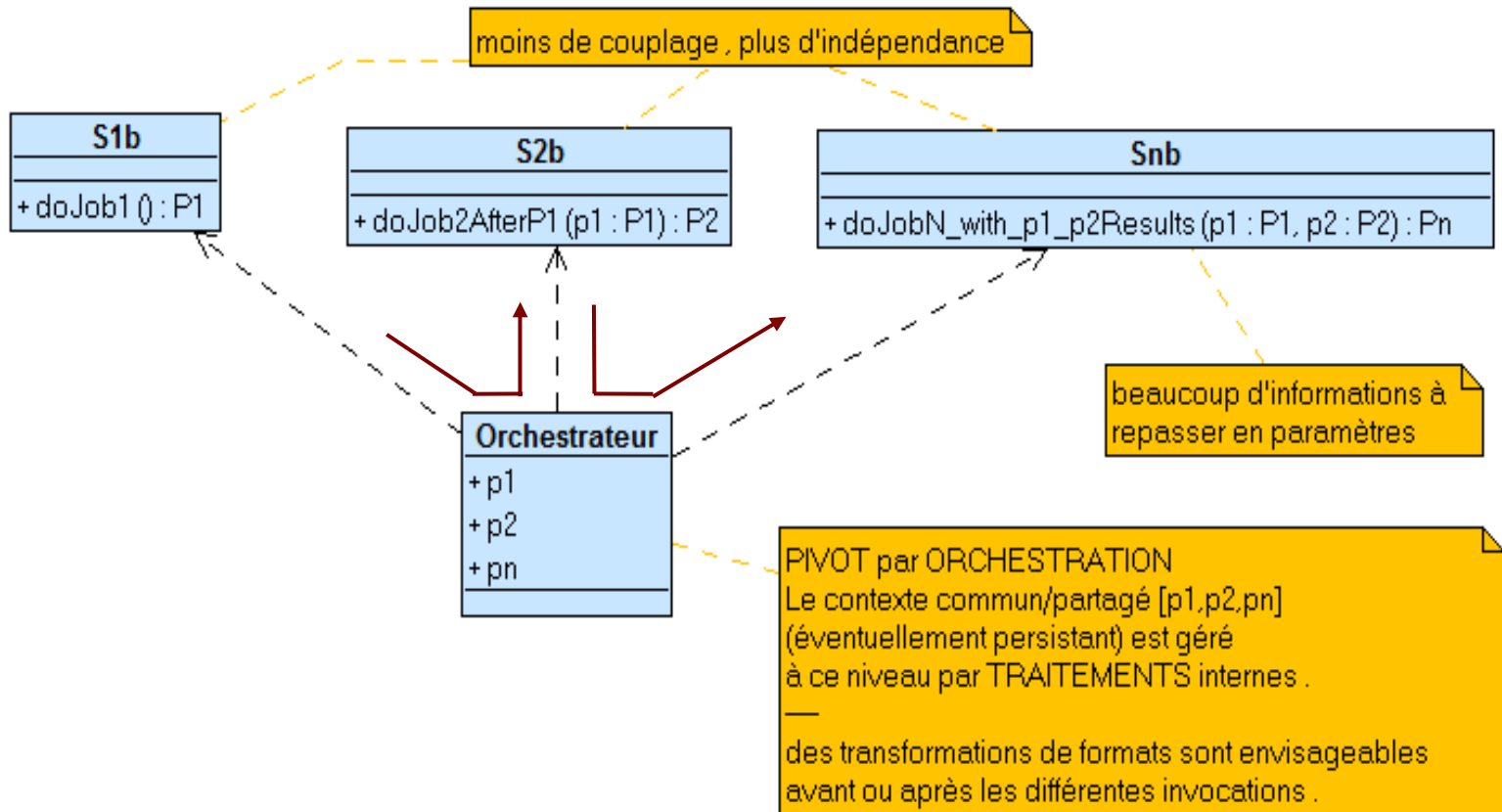
==> Techniquement parlant, dans la plupart des cas , l'orchestration suffit (c'est plus simple à mettre en œuvre et plus fiable). La plupart des technologies existantes (bpel , ...) s'occupent de l'orchestration (et pas de la chorégraphie).

==> Fonctionnellement, on a souvent besoin de modéliser des chorégraphies (avec plusieurs couloirs) pour bien montrer une collaboration de principe entre plusieurs parties prenantes (ex : "client" , "logistique" , "fournisseur") qui partagent un même but (ou sous but) et qui sont quelquefois mutuellement engagées au sein d'une logique de partenariat (quelquefois contractuelle).

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

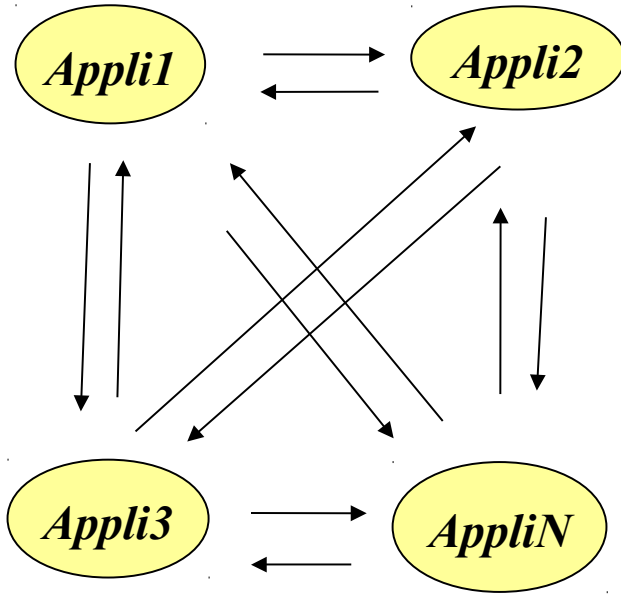


Données "pivot"

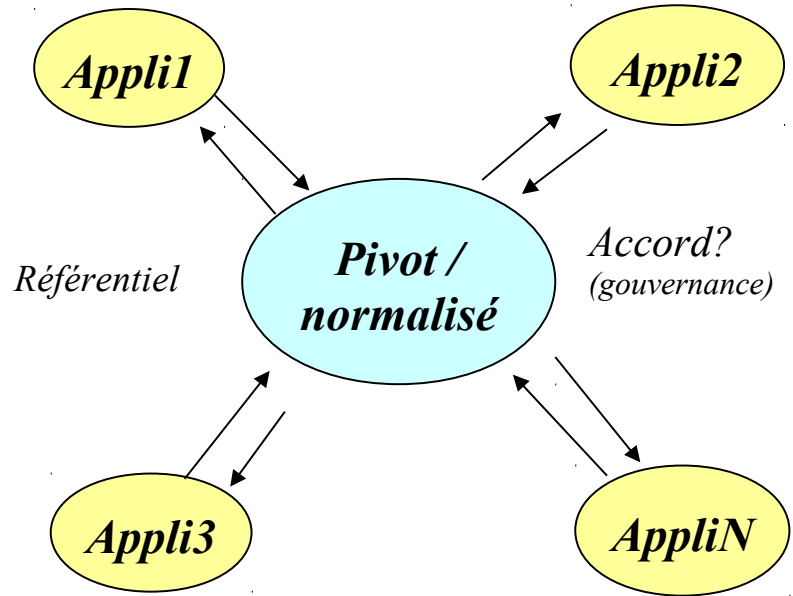


Récupérées comme résultat d'un appel sur un premier service et ré-envoyées en entrée d'un autre service, certaines données pivotent (telles quelles ou retransformées) au niveau de l'orchestrateur.

éventuel format fonctionnel normalisé pour "pivot"



Si transformations
des formats de données
au cas par cas :
--> besoins en $O(n^2)$!



Si transformations avec format
de données intermédiaire
normalisé (pivot) :
--> besoins en $O(n)$!

BPMN et BPMN2

***BPMN** signifie **B**usiness **P**rocess **M**odel and **N**otation*

** 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 bien interprété tel quel via avec certaines extensions.*

** Un **diagramme BPMN** ressemble beaucoup à un diagramme d'activité UML . Les notions exprimées sont à peu près les mêmes.*

** A l'époque BPMN 1 , pas de format de fichier normalisé . Chaque logiciel de modélisation gère son propre format propriétaire. Certains outils exportaient les processus au format "XPDL" ("Xml Process Definition Language")*

** La version 2 de BPMN a normalisé le format de fichier (xml) et donc plus besoin de XPDL. Un fichier "bpmn(2)" généré par un outil 1 peut être repris par un autre logiciel compatible "bpmn(2)".*

Contenu des diagrammes BPMN2

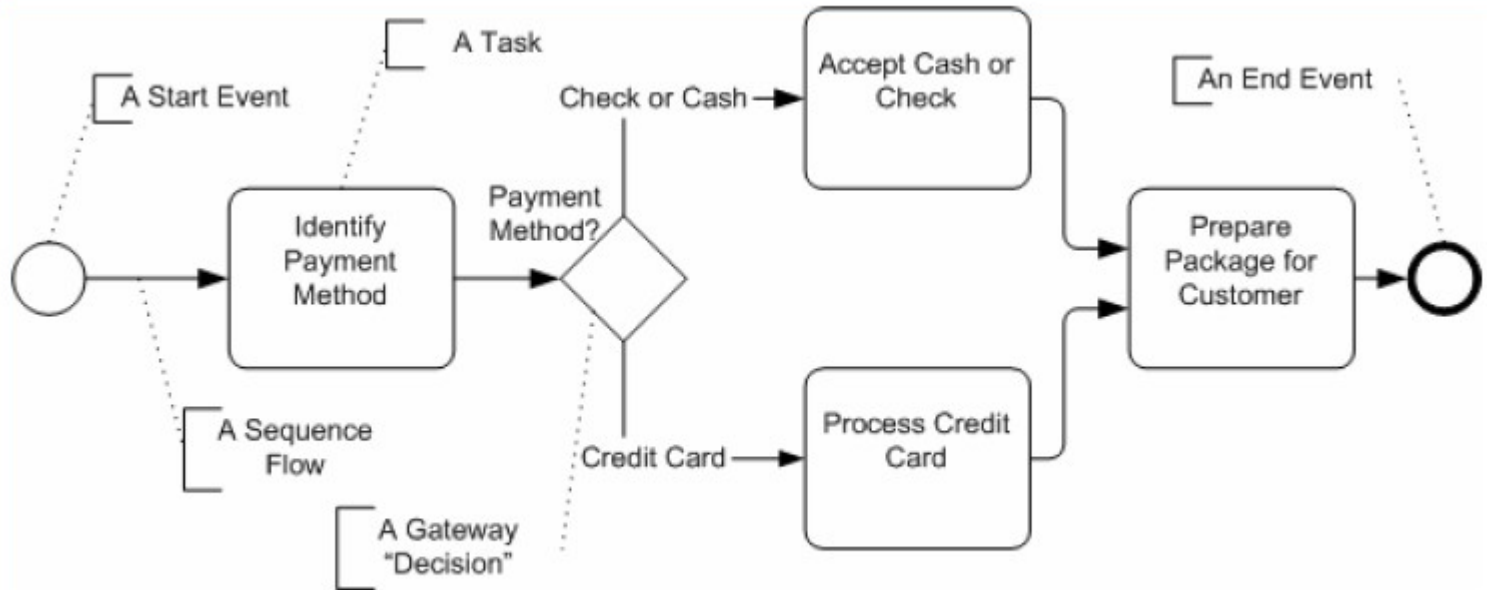





Figure 1: An Example of a Simple Business Process

Un diagramme BPMN est constitué de :

- * **Événement** (Event)
- * **Tâches** (Task)
- * **Gateway** (losange de contrôle de flux, coordination conditionnée)

Types d'événements "BPMN"

-  **Start (début)** – *bordure simple et fine*
-  **Intermediate**
(intermédiaire) – *bordure double*
-  **End (fin)** - *bordure simple épaisse*

Quelques "sous types" (précisions) :

None



Error



Message



Compensation



Timer

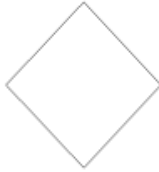


Cancel



Types de "gateway"

**Exclusive
Data-Based**



Alternative exclusive selon condition sur donnée(s) du processus

Event-Based



Branchement événementiel en fonction de la nature de l'événement qui surviendra (ex : accord , refus , ...)

Inclusive



Une branche toujours effectuée et d'éventuelles autres branches facultatives/annexes (selon conditions)

Complex



Sémantique complexe

Parallel



Plusieurs branches effectuées en //

Quelques autres éléments de syntaxe BPMN

Connexions :

* **sequence flow** (séquence ordonnée)



* **message flow** (entre 2 participants/couloirs)



* **association** (entre tâche et données)



Couloirs/partitions d'activités :

* **pool** (*un par participant : Processus ou Partenaire ou ...*)

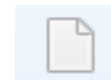


* **lane** (*sous partition*)

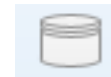


Data Objects :

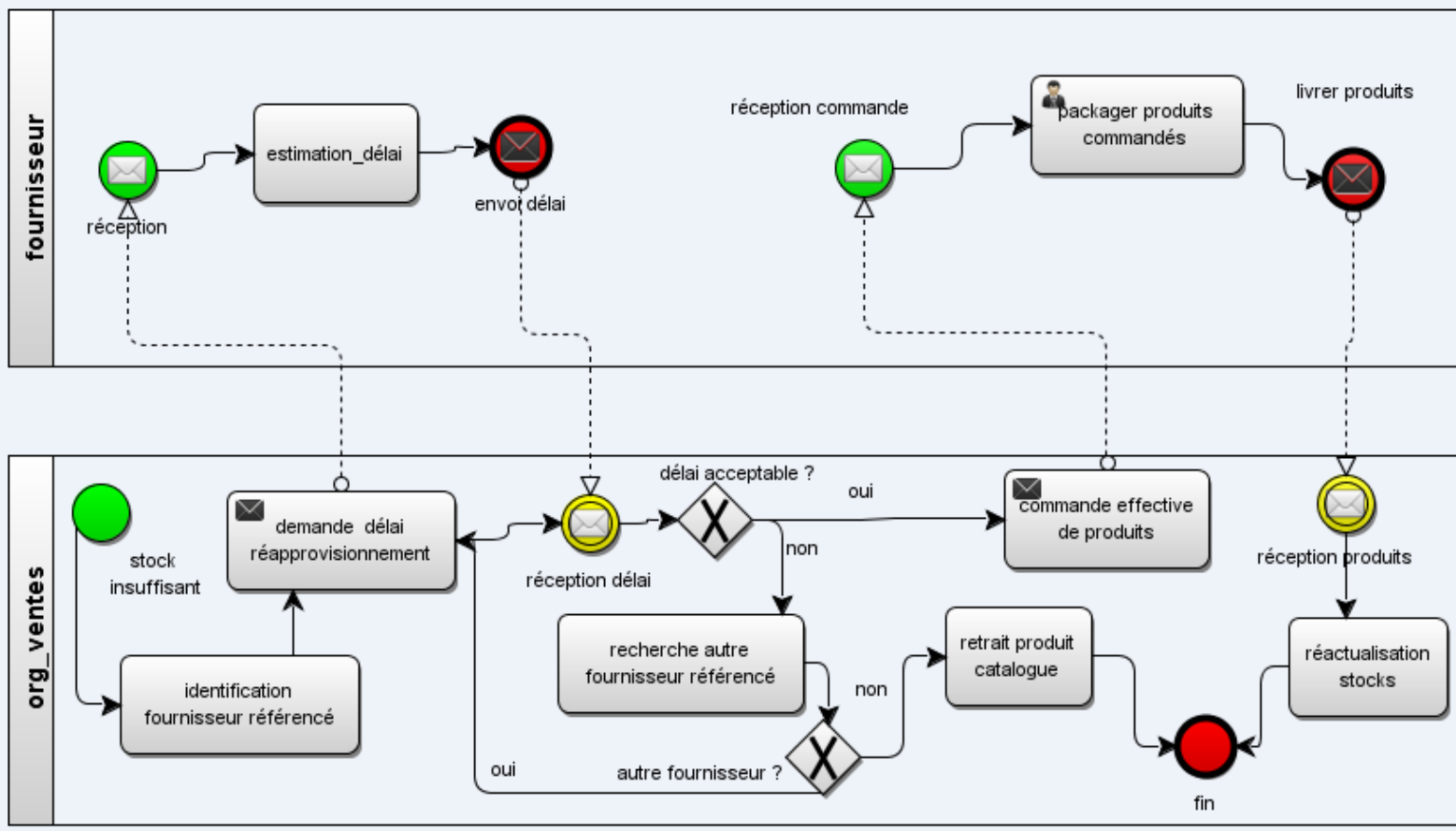
* **Data** (*document xml ,objet en mémoire , ...*)



* **Data Store** (*extraction/persistance en base,...*)



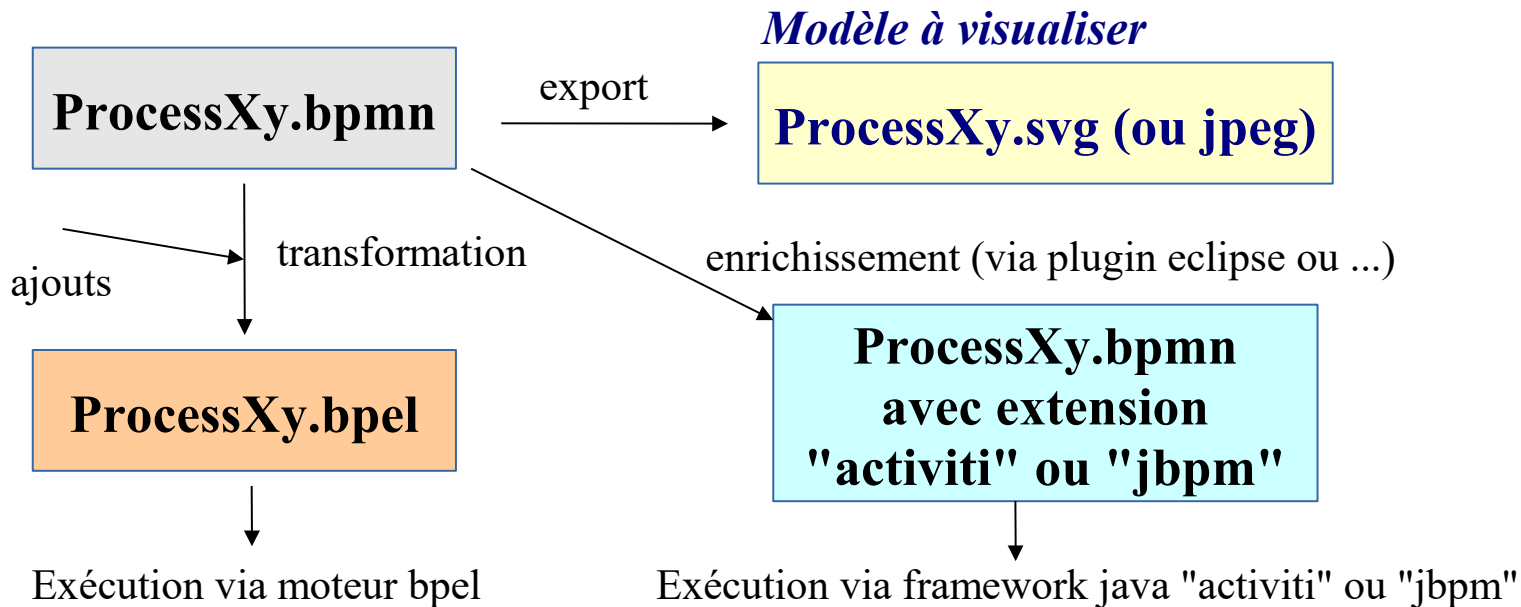
Exemple de diagramme BPMN



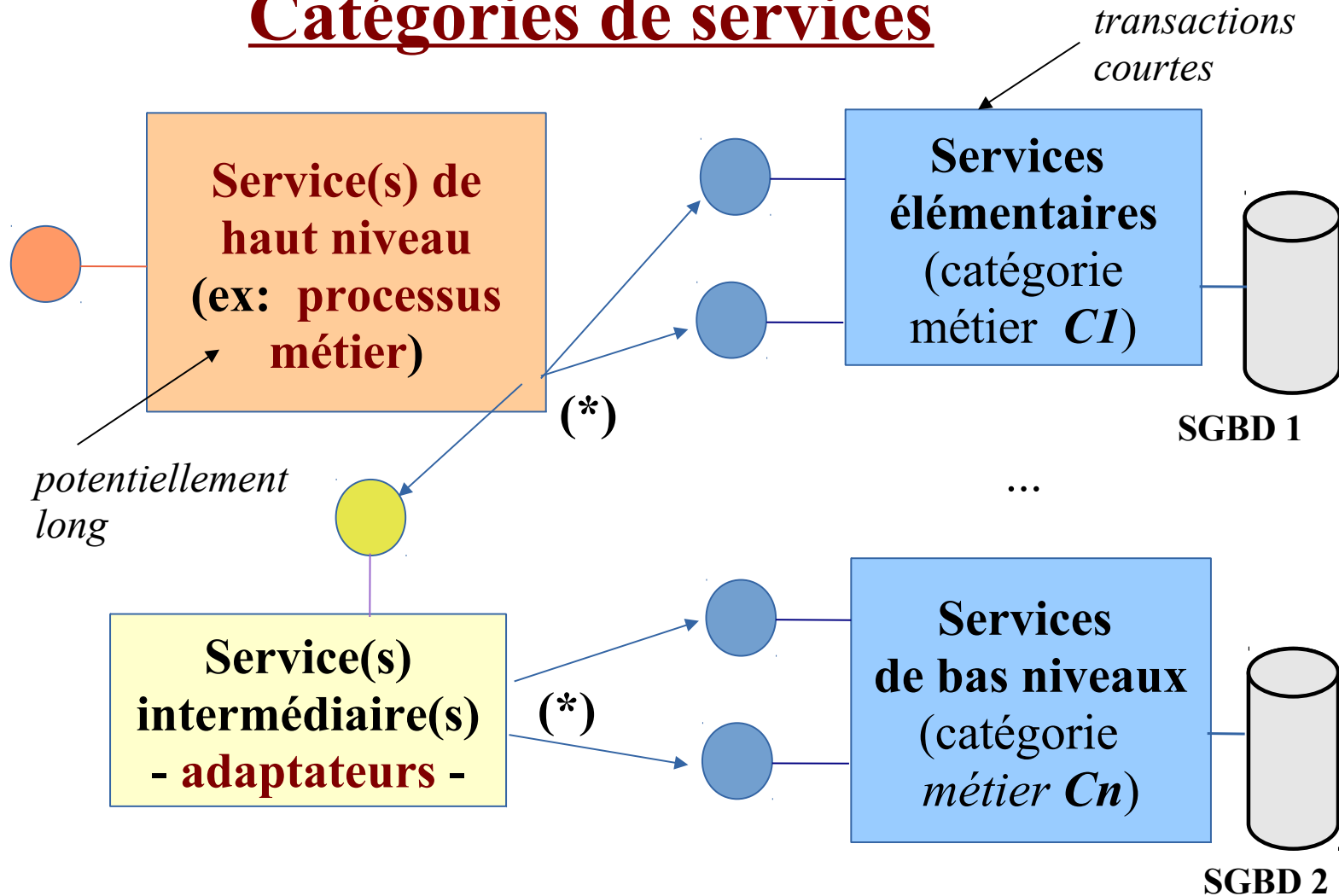
Quelques éditeurs bpmn

- * **Bizagi** Process Modeler
- * **Yaoqiang bpmn(2) editor** (très bien)
- * éditeur sous forme de plugin eclipse (activiti/jbpm)
- * ...

Utilisations possibles d'un fichier ".bpmn"

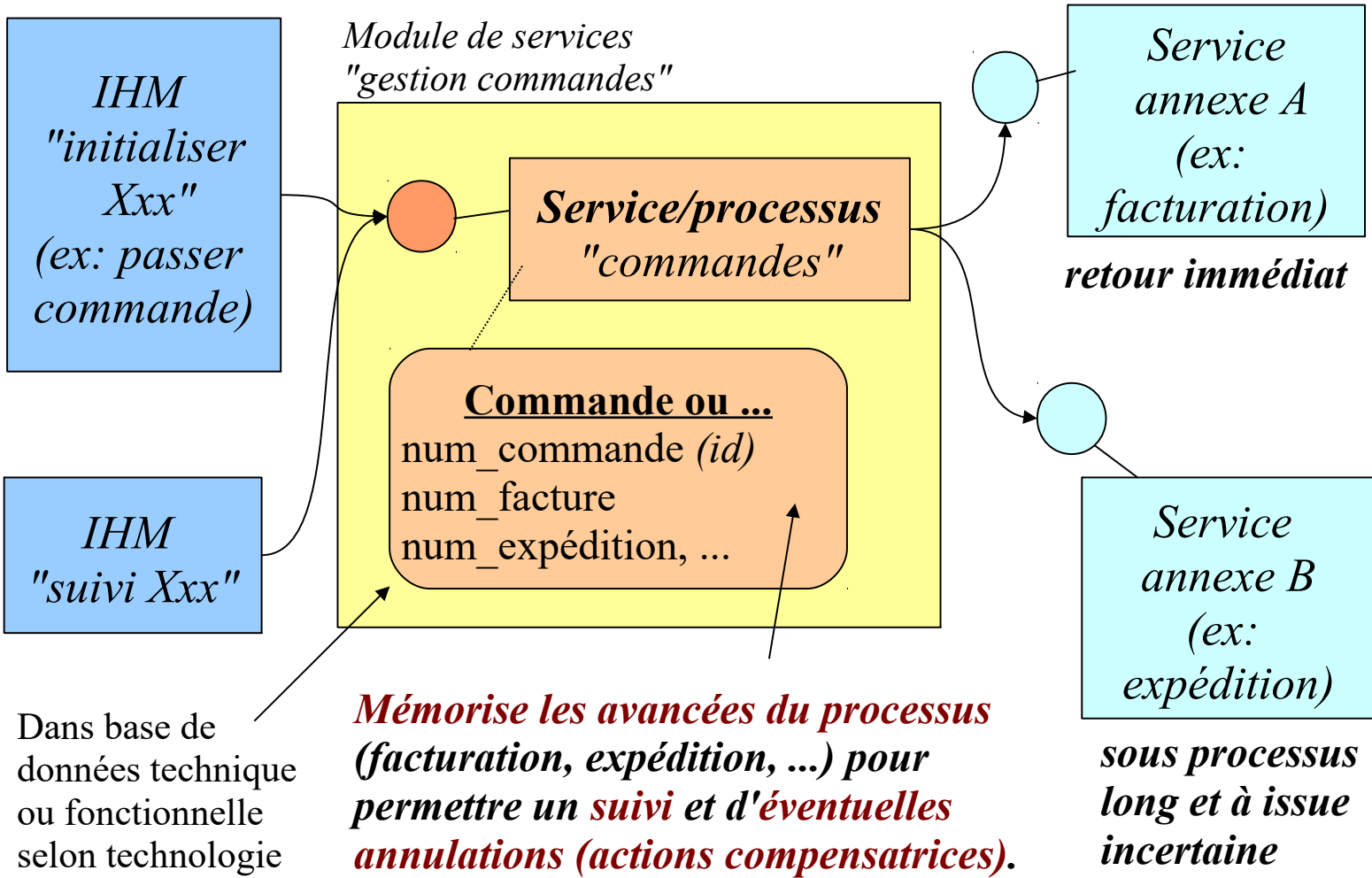


Catégories de services



(*) Orchestration/combinaison potentielle de services .

Etats d'avancement d'un processus long



Plusieurs niveaux/degrés d'orchestration

Orchestration synchrone élémentaire:

- * Invocation d'opérations avec réponses immédiates sur différents "web services"
 - * Enchaînements de tâches conditionnés , adaptateurs sur données récupérées et envoyées
- *du code "java" ordinaire suffit (web service de haut niveau)*

Orchestration asynchrone (processus long):

- * fonctionnalités élémentaires ci-dessus +
 - * communications asynchrones avec corrélations (requêtes , réponses différées à associer aux requêtes préalables)
 - * attente d'événements (issues/résultats différés de traitements longs)
 - * gestion de plusieurs instances d'un processus (avec contexte : état d'avancement + variables + ...)
 - * éventuels déclenchements de tâches humaines , synchronisations, ...
- *une technologie spécifique/sophistiquée est nécessaire (bpel ou activiti ou jbpmm ou ...)*

BPEL (Business Process Execution Language)

- * **BPEL** est une technologie d'orchestration **entièrement basée** sur des syntaxes **XML** (*.bpel + .wsdl + .wsd = tout en xml*).
 - * La technologie BPEL est assez sophistiquée et spécialisée dans l'orchestration de services . BPEL donne tout son potentiel dans un cadre asynchrone (avec plusieurs instances et corrélations).
 - * BPEL se concentre sur l'aspect logique des processus métier et se veut indépendant de la façon exacte de communiquer avec l'extérieur. Un processus BPEL s'exécute donc souvent au sein d'un ESB gérant tout un tas de protocoles pour s'accommoder aux échanges externes (http , jms , smtp , ...).
 - * **BPEL** est à considérer comme un **standard** .
 - * Les principales solutions logicielles "BPEL" sont celles des **grandes marques** (IBM , Oracle , Microsoft , ...).
- ==> La technologie BPEL est rarement "open source" et n'est pas facilement accessible/abordable.
- Il faut généralement s'appuyer sur des IDE spécifiques (avec plein d'assistants) et exécuter le code dans des serveurs spécialisés.

Technologies d'orchestration basées sur "bpmn2 + java"

- * **jBPM_5** et **activiti_5** sont des "**gestionnaires de processus**" ("**moteurs de workflow**" + ...) à intégrer dans des applications **java** (sous forme de ".jar").
 - * *Ces deux technologies (très proches) s'appuient directement sur le standard **BPMN2** (avec quelques extensions interprétées en plus pour le lien avec des variables "java").*
 - * **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** ou **8** ou encore dans **ServiceMix** ou **MuleESB**)*
 - * Bien que nécessitant quelques ajouts de code java (*variables, invocations, ...*), **jBPM5** et **activiti** peuvent être utilisés pour effectuer de l'orchestration de services.
- ==> **Ces technologies "open source"** sont assez facilement **abordables** et ne nécessitent pas d'IDE ou de serveurs spécialisés.

BPEL (***B**usiness **P**rocess **E**xecution **L**anguage*)

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 quelquefois 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 il existe quelques variantes d'interprétations selon le moteur d'exécution (ODE, Orchestra , ...)

BPEL est à considérer comme un **standard** (reconnu par les grandes marques : IBM , Oracle , Microsoft , ...)

Principales fonctionnalités de BPEL:

- * **invoker** 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.

Quelques implémentations : moteurs BPEL

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

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")

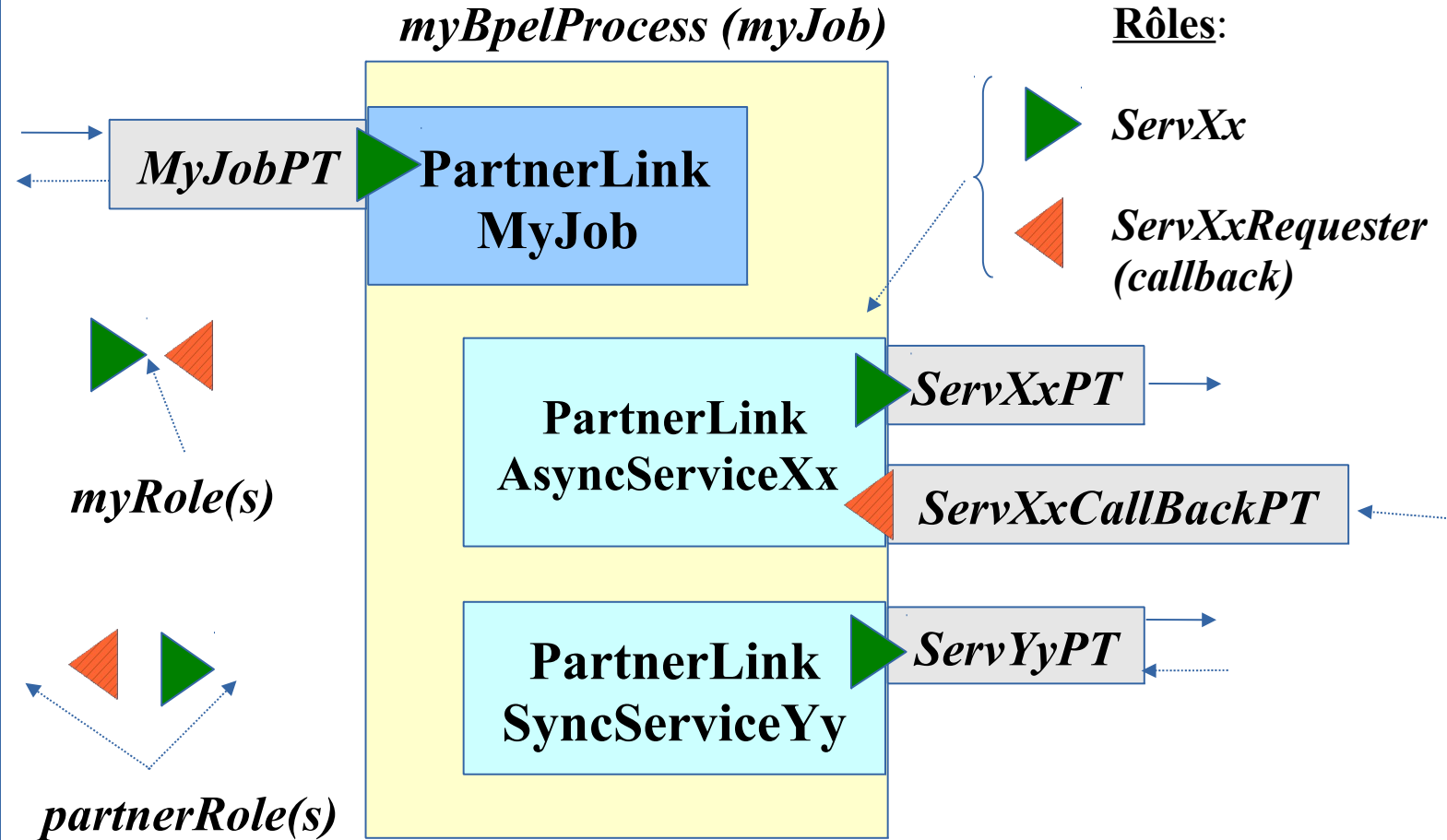
Séquence type d'un processus BPEL:

```
<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:



BPEL - Importation des définitions WSDL

```
<?xml version="1.0" encoding="UTF-8"?>
<process name="..." targetNamespace="..."
  xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="..." xmlns:ns0="..." ...>

  <import namespace="http://yyy/xyz"
    location="xxx.wsdl"
    importType=
      "http://schemas.xmlsoap.org/wsdl/" />

  ....
</process>
```

Eléments WSDL pour BPEL (1/2):

yyy.wsdl

```
<definitions targetNamespace="http://xxx/wsdl/yy"
  xmlns="http://schemas.xmlsoap.org/wsdl/" ...
  xmlns:tns="http://xxx/wsdl/yy"
  xmlns:ns="http://xxx/schema/yyy" >
  <types>
    <xsd:schema targetNamespace="http://xxx/wsdl/yy">
      <xsd:import namespace="http://xxx/schema/yyy"
        schemaLocation="yyy.xsd"/>
    </xsd:schema>
  </types> ....
  <message name="...." >
    <part name="parameters" type="ns:abcType"/>
  </message>
```

...

Elements XSD (via wsdl) pour BPEL:

yyy.xsd

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://xxx/schema/yyy"
  xmlns:tns="http://xxx/schema/yyy" elementFormDefault="qualified">
  <xsd:element name="operation1Req" type="tns:op1ReqType" />
  <xsd:element name="operation1Resp" type="tns:op1RespType" />
  <xsd:complexType name="op1ReqType">
    <xsd:sequence> <xsd:element name="x" type="xsd:int"/>
      <xsd:element name="y" type="xsd:double"/> </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="op1RespType">
    <xsd:sequence> <xsd:element name="a" type="xsd:double"/>
      </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Eléments WSDL pour BPEL (2/2):

yyy.wsdl

```
<definitions ... xmlns:tns="http://xxx/wsdl/yy" ...  
  xmlns:plnk="http://docs.oasis-open.org/wsbpel/2.0/plnktype" >  
  <types>... </types> <message name="..." >... </message> ...  
  <portType name="serviceXyPortType">  
    <operation name="operation1">  
      <input name="input1" message="tns:operation1Request"/>  
      <output name="output1" message="tns:operation1Reply"/>  
    </operation> ... </portType>  
  <binding ....> </binding> ... <service ....> </service>  
  
  <plnk:partnerLinkType name="ServiceXyPartnerLinkType">  
    <plnk:role name="ServiceXyPortTypeRole"  
      portType="tns:serviceXyPortType"/>  
  </plnk:partnerLinkType> </definitions>
```

BPEL – Déclaration des "partnerLink"

```
<process ...> ....  
<partnerLinks>  
  <partnerLink name="ServiceXyPartnerLink"  
    xmlns:ns0="http://xxx/xyz"  
    partnerLinkType="ns0:ServiceXyPartnerLinkType"  
    partnerRole="ServiceXyPortTypeRole"/>  
  <partnerLink name="MyJobPartnerLink"  
    xmlns:tns="http://yyy/myjob"  
    partnerLinkType="tns:MyJobPartnerLinkType"  
    myRole="myJobPortTypeRole"/>  
</partnerLinks> ....  
</process>
```

BPEL – Déclaration des variables

```
<process ...>...  
  <variables>  
    <variable name="operation1ServXYOut"  
      xmlns:ns0="http://xxx/xyz"  
      messageType="ns0:operation1Reply"/>  
    <variable name="operation1ServXYIn"  
      xmlns:ns0="http://xxx/xyz"  
      messageType="ns0:operation1Request"/>  
    <variable ... messageType="xsd:String" />  
  </variables> ....  
</process>
```


BPEL – Liste des principales "*activity*"

receive	Réception de message
reply	Renvoi de la réponse du processus
invoke	Invocation d'une opération sur un service
assign/copy	Affectation de variable(s)
sequence	Suite séquentielle d'activités
flow	Activités concurrentes (en //)

BPEL – receive & reply:

```
<process ...> ....  
  <sequence>  
    <receive operation="operation1"  
      partnerLink="MyJobPartnerLink"  
      variable="operation1ServXyIn"  
      createInstance="yes"  
      xmlns:tns="http:yyy/myjob"  
      portType="tns:MyJobPartnerLinkType" />  
    ...  
    <reply operation="..."  
      partnerLink="MyJobPartnerLink"  
      variable="...Out" portType="..." />  
  </sequence> </process>
```

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

*réponse
globale du
processus
BPEL*

BPEL – assign , copy [from, to]:

<process ...> <sequence> ...

<assign>

<copy>

<from>*\$op1ServYIn.parameters/ns0:montant*</from>

<to>*\$opAServZIn.parameters/ns1:somme*</to>

</copy>

<copy>

<from> (*\$aaIn.parameters/ns0:x* **div**
\$bbIn.parameters/ns0:y) </from>

<to>*\$op1ServYOut.parameters/ns0:a*</to>

</copy>

</assign>

...

BPEL – invoke:

```
<process ...> ....  
  <sequence> ...  
  <invoke operation="operationZ"  
    partnerLink="ServiceXyPartnerLink"  
    inputVariable="...In"  
    outputVariable="...Out"  
    xmlns:ns0="http://xxx/xyz"  
    portType="ns0:ServiceXyPartnerLinkType"/>  
  ...  
</sequence> ...  
</process>
```

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

NB: en mode *réponse asynchrone*,
<invoke operation="callbackXy"
.../> remplace <reply .../>
après opérations longues .

BPEL – test (if/else):

<if>

<condition>*bool-expr***</condition>**

activity

<elseif>*** ←

<condition>*bool-expr***</condition>**

activity

</elseif>

0 à n

<else>*?* ←

activity

</else>

0 ou 1

</if>

BPEL – boucles *while* et *repeatUntil*

<while>

<condition>*bool-expr***</condition>**

activity

</while>

<repeatUntil>

activity

<condition>*bool-expr***</condition>**

</repeatUntil>

exemple: \$var1.xx = 10 ou > , < , >= , <= , != ,
not(...) , ... **and/or** ... (syntaxe *xpath 1.0 / xslt*)

BPEL – boucle *forEach* (de n à m)

```
<forEach counterName="i">  
<startCounterValue>1</startCounterValue>  
<finalCounterValue>10</finalCounterValue>  
<scope>activity*<scope>  
</forEach>
```

*la variable bpel **i** doit être de type
xsd:integer ou équivalent*

*D'autres options sont disponibles pour forEach
==> approfondir si besoin la norme WS-BPEL2*

BPEL *flow* – activités concurrentes (en //)

<flow>

<links>?

<link name="lien_xy"/>+

</links>

activity+

</flow>

liaisons (facultatives)
= dépendances
(avant/après) entre
activités pour exprimer
des synchronisations

activités en //

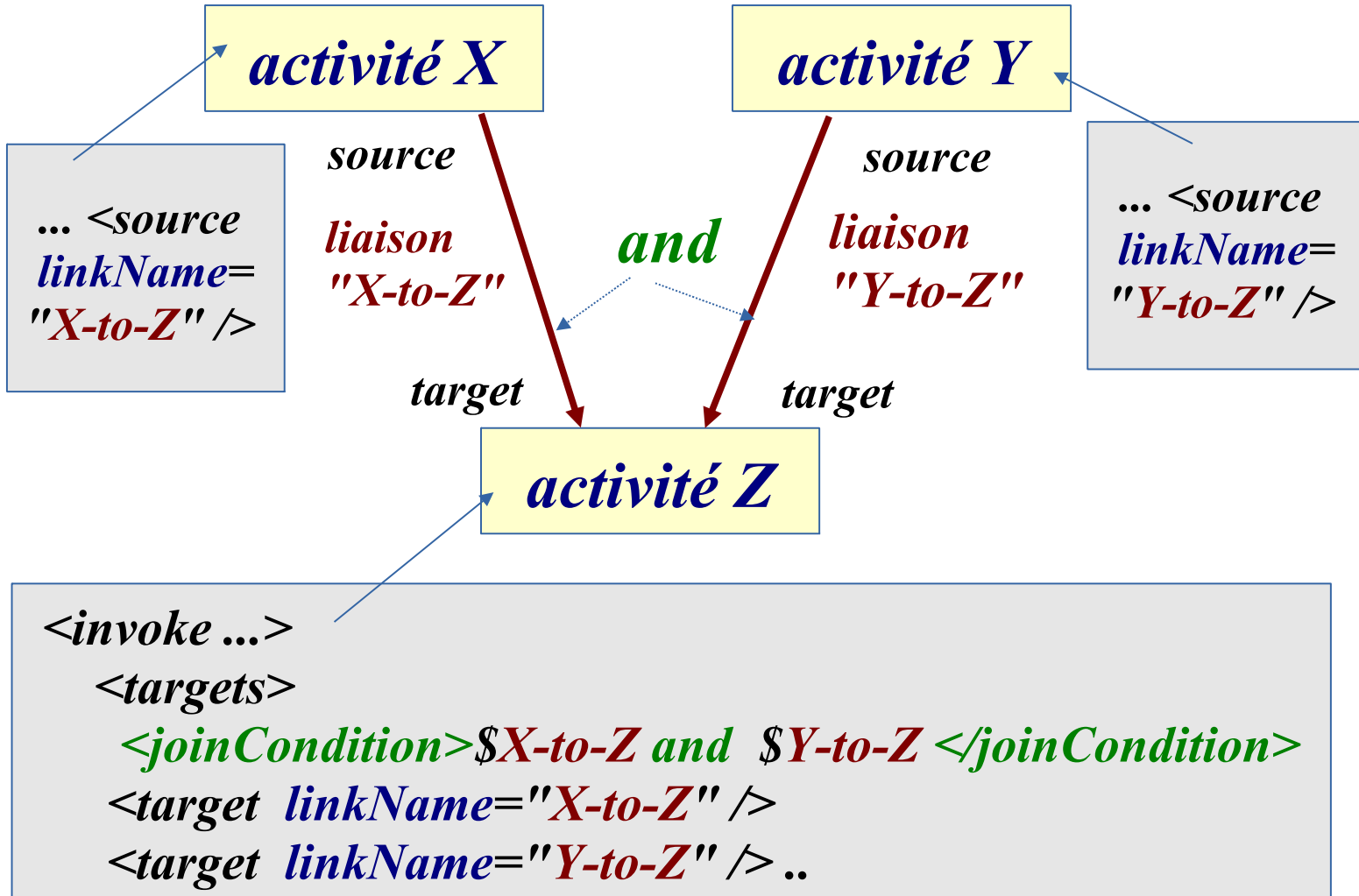
<invoke ou ...> ...

<targets et/ou sources>?

<target ou source linkName="lien_xy"/>+

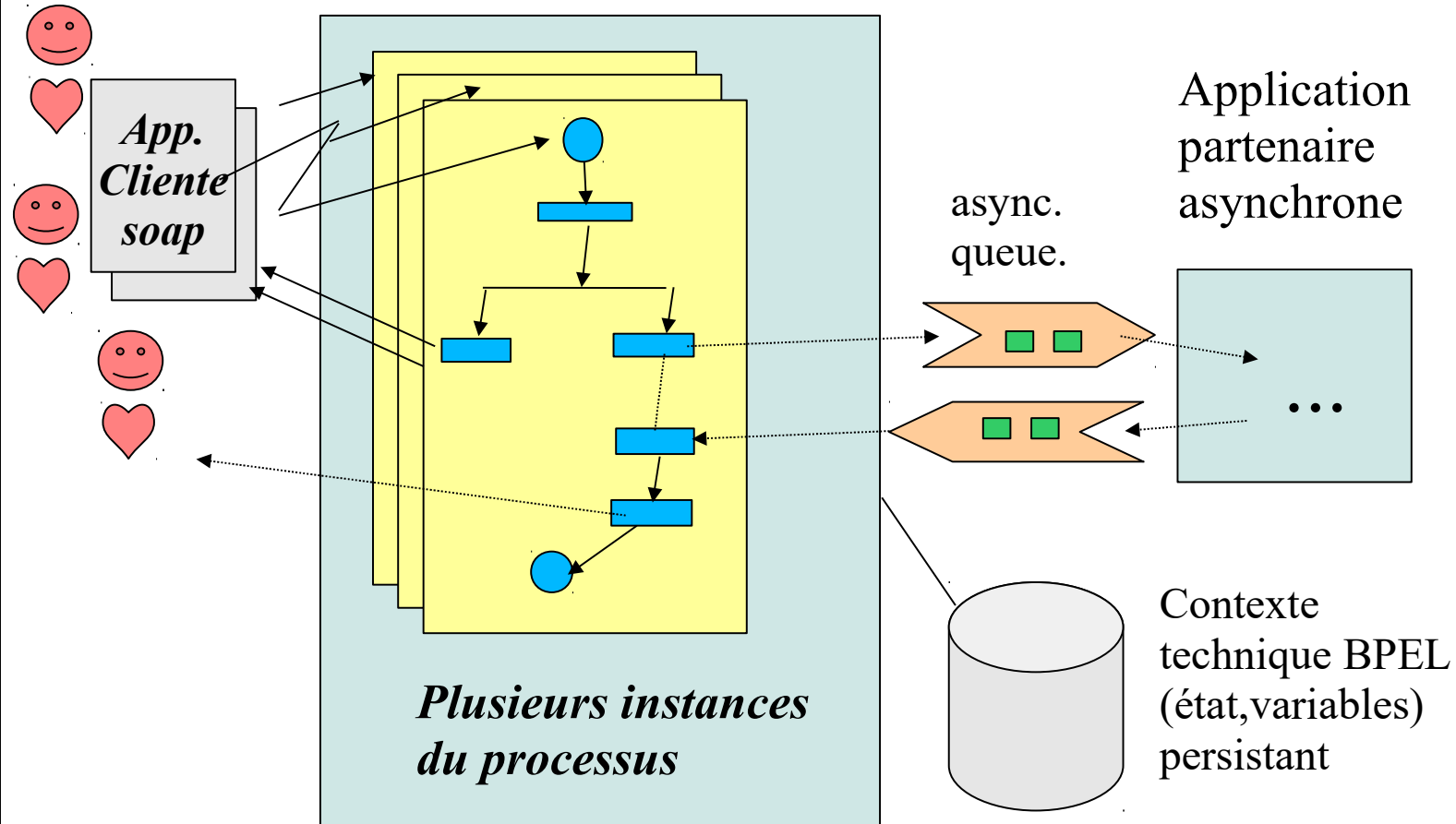
</targets et/ou sources> ... </invoke ou ...>

BPEL flow – link (source , target) , join



BPEL (dans tout son potentiel) en mode asynchrone

Moteur d'orchestration BPEL



BPEL - mode asynchrone et corrélations

```
<invoke operation="xxx" ...>  
<correlations>  
  <correlation set="xxxCorSet"  
    initiate="yes" />  
</correlations> ... </invoke>
```

one-way

request(s)

```
<pick> <onMessage  
  operation="xxxRespCallBack">  
<correlations>  
  <correlation set="xxxCorSet"  
    initiate="no" />  
</correlations> ... </onMessage>
```

one-way

response(s)

BPEL – *pick* (receive avec alternative)

<pick>

<onMessage operation="xxxResponse"

....> <correlations> ...</correlations>

<!-- gérer la réponse positive (acceptation) -->

</onMessage>

<onMessage operation="xxxReject"

....> <correlations> ...</...>

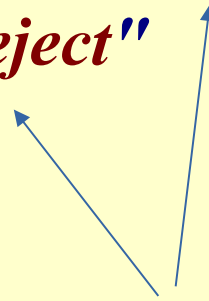
<!-- gérer le refus -->

</onMessage>

...

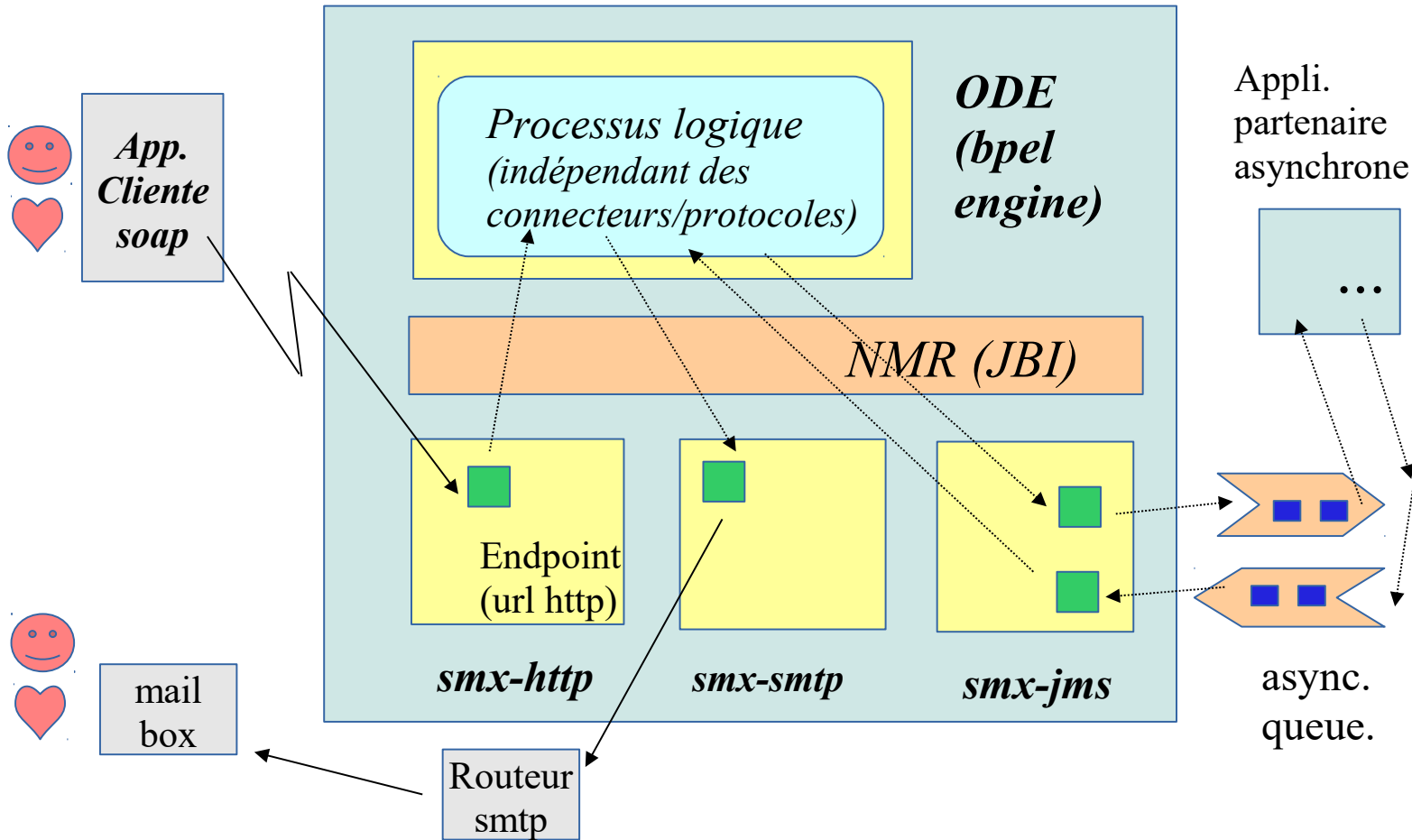
</pick>

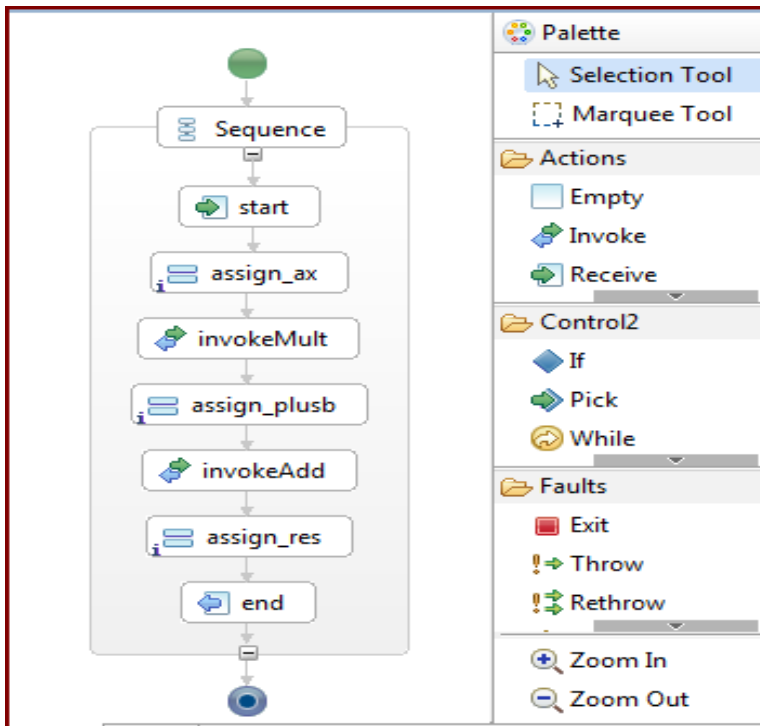
callback



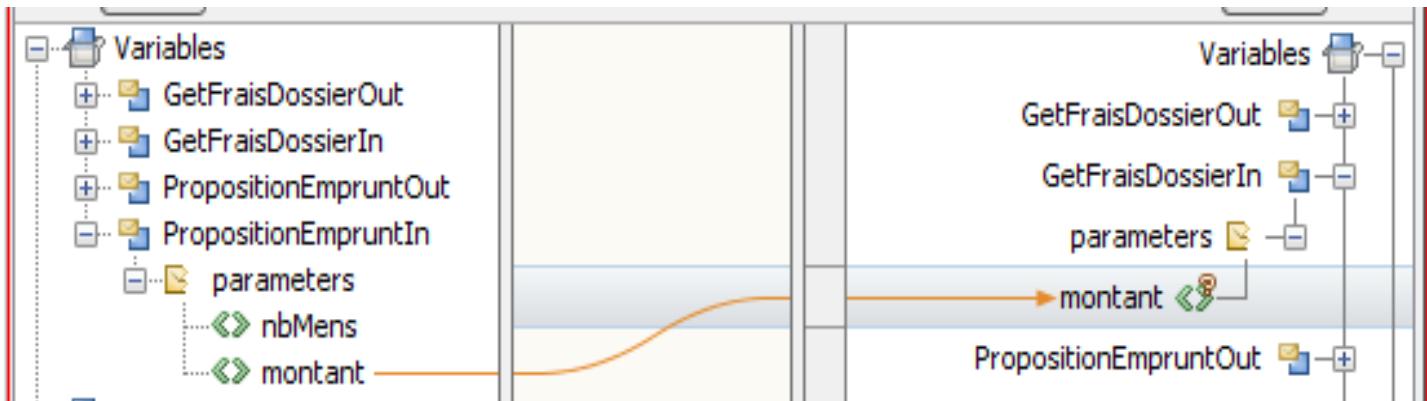
Ex. d'intégration: BPEL asynchrone via ODE dans ancien ServiceMix 4.2

ESB ServiceMix 4.2 (ancienne version)

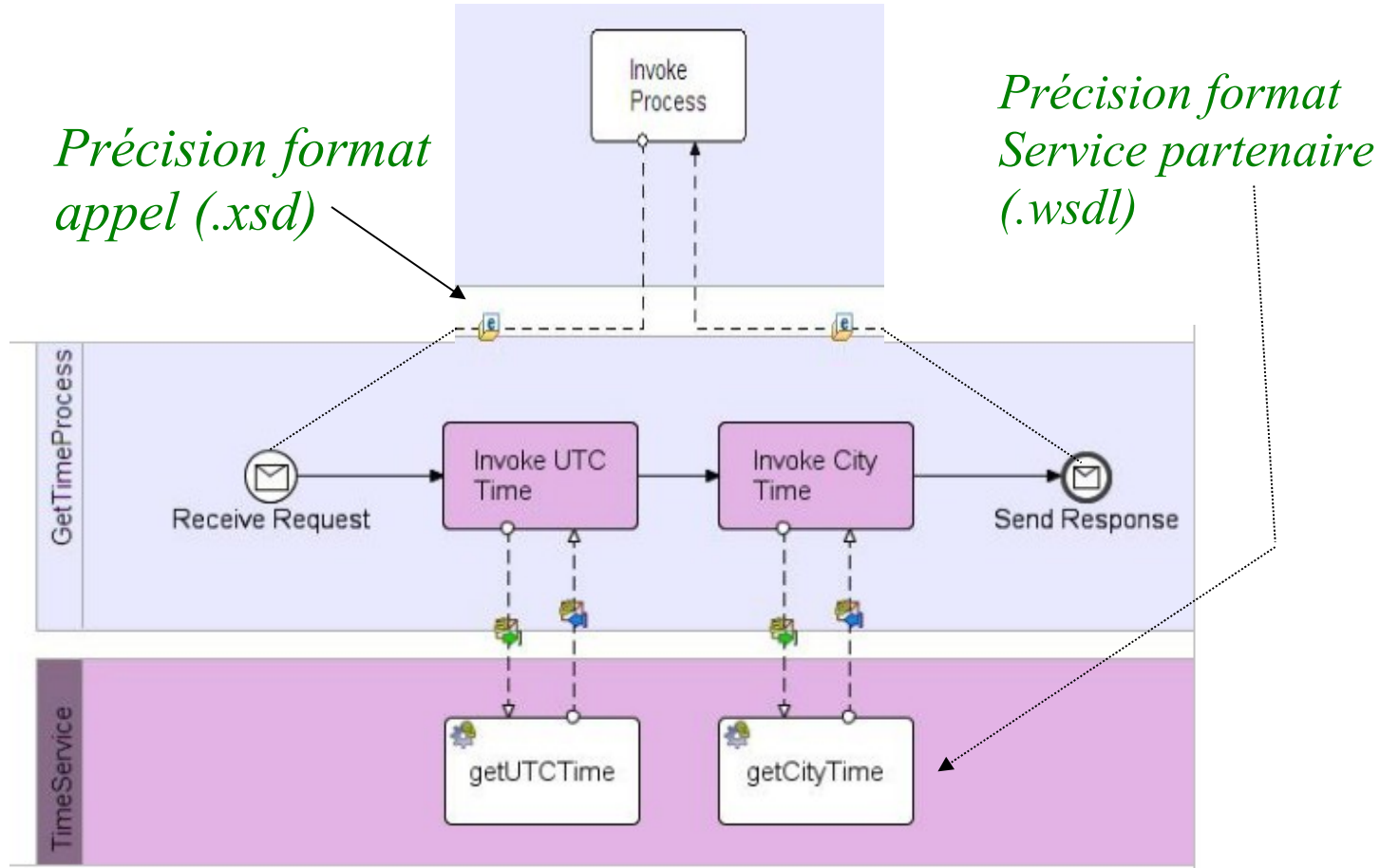




Développement BPEL
direct via *NetBeans*
 ou *Bpel-designer (eclipse)*
 ou ...

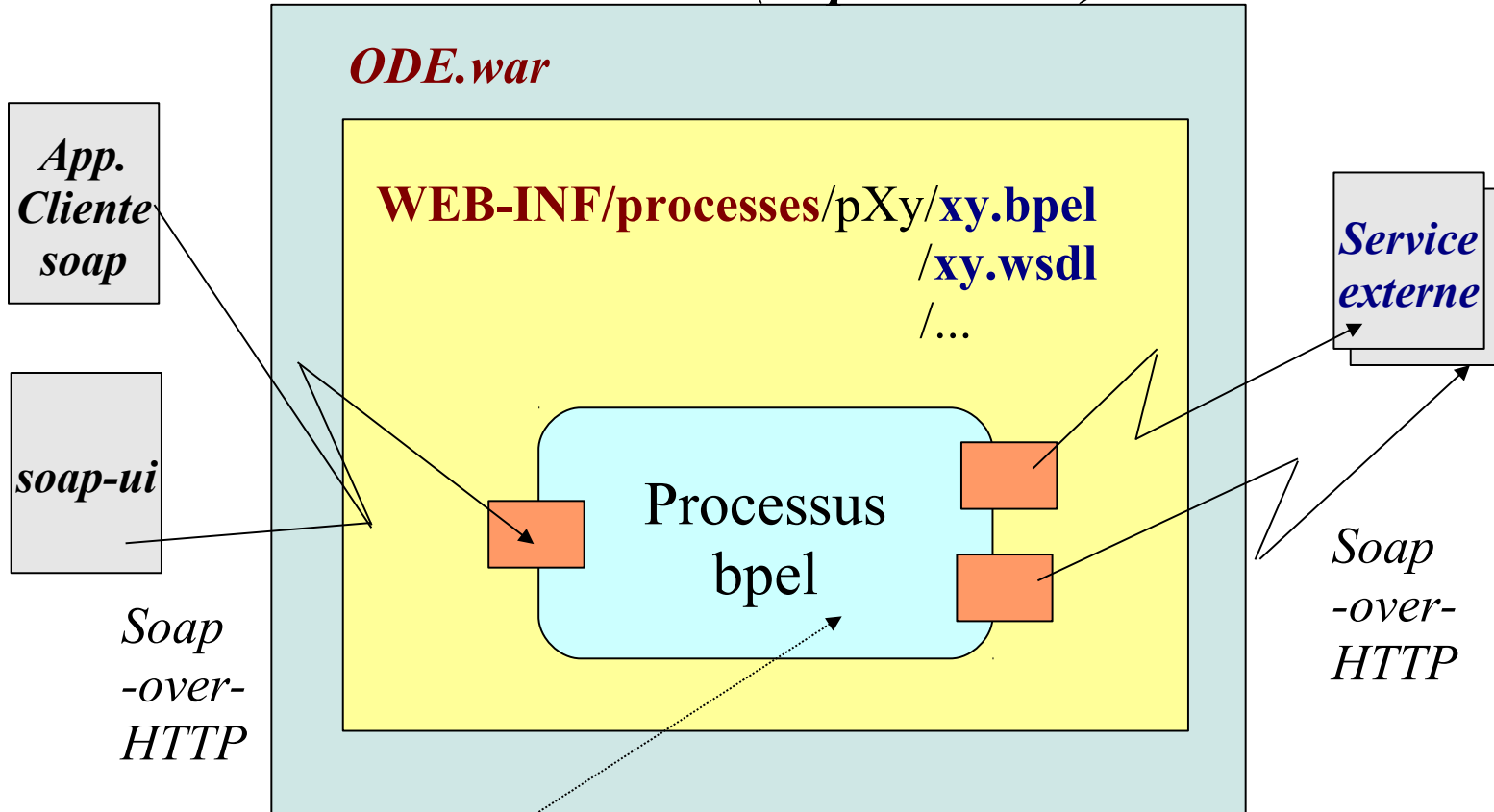


Développement BPEL indirect depuis *Intalio BPMNs Designer* (*modélisation BPMN → code BPEL*)



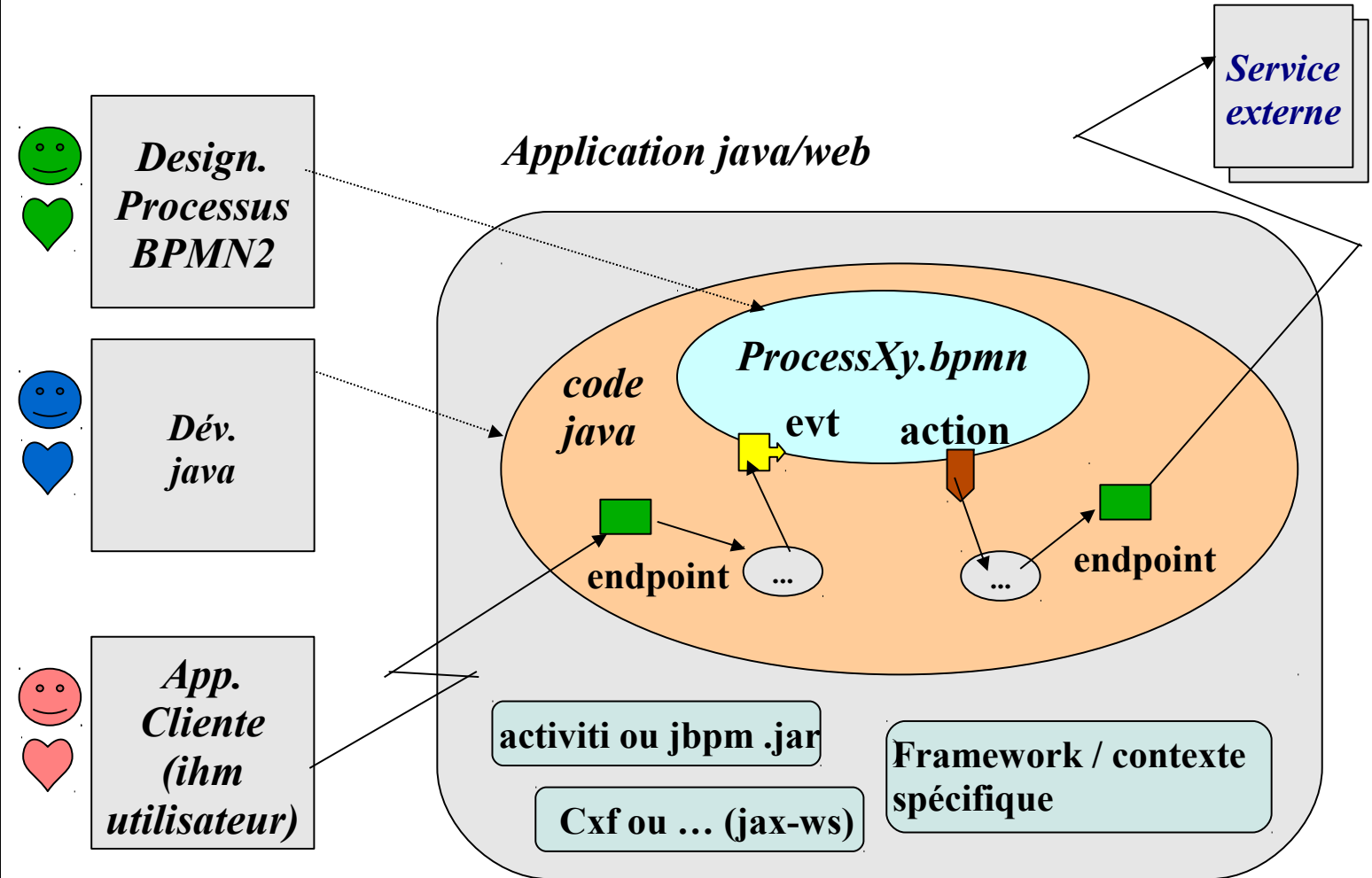
Exemple d'intégration 1: BPEL synchrone via ODE dans Tomcat

Serveur Jee Tomcat (http://...:8080)



() synchrone conditionné*

Socle de fonctionnement de (java/BPM)



Standard bpmn2 (omg)

*Éditable via
yaoqiang bpmn2 editor
Ou autre.*

Fichier **xml**
avec éléments
des diagrammes bpmn
et coordonnées graphiques

**bpmn2 +
extension jbpm_5**

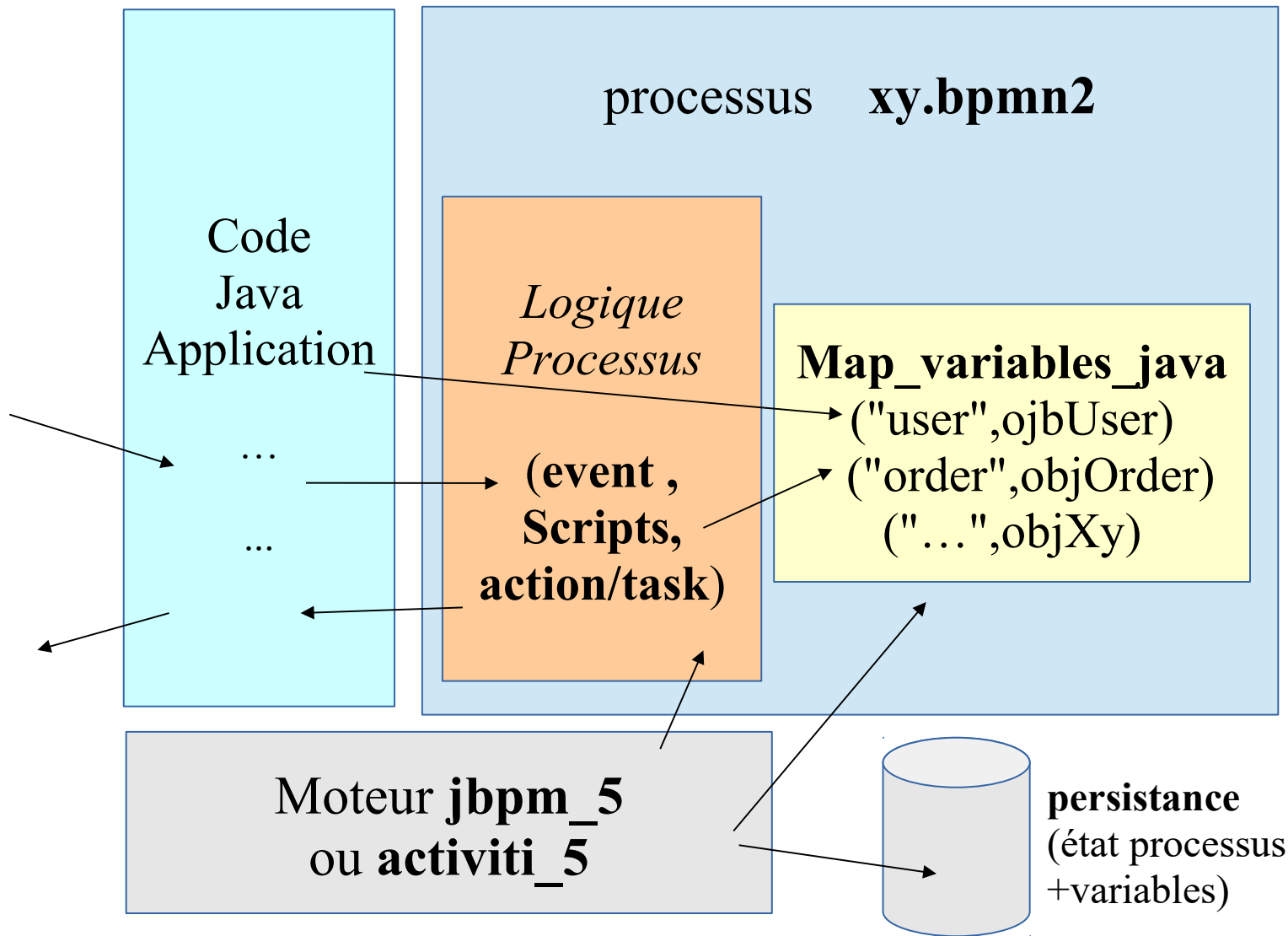
*ou
bien*

**bpmn2 +
extension activiti_5**

Éditeur spécial/plugin eclipse

Autre éditeur spécial/plugin eclipse

*Ajout direct de "variables + paramétrages/scripts" dans
le même fichier ".bpmn2"*

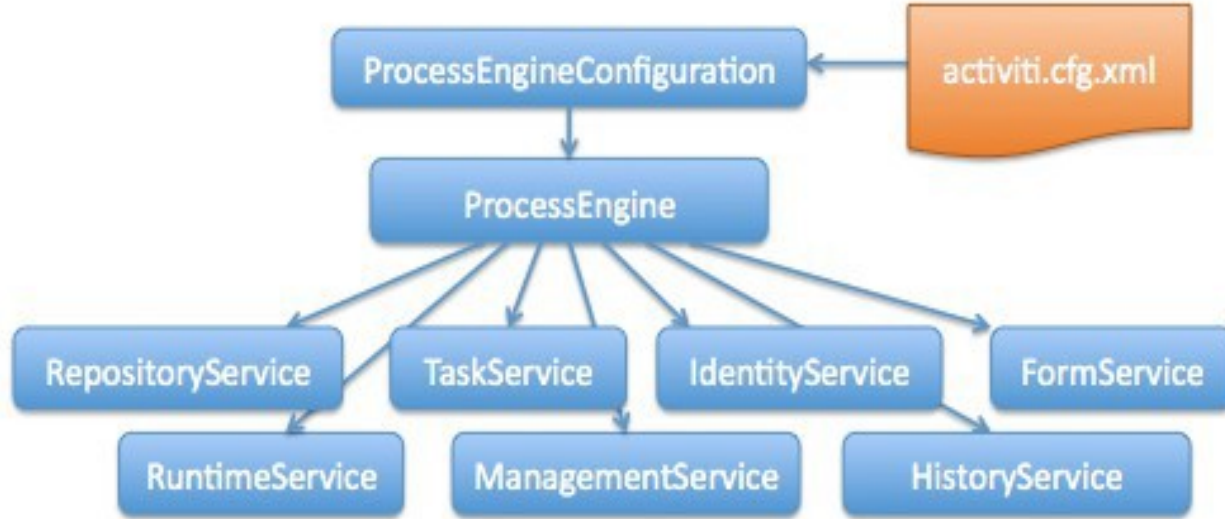


activiti

Activiti peut éventuellement être utilisé seul mais l'**intégration au sein de spring** apporte beaucoup d'avantages. il serait dommage de les ignorer.

Une **documentation détaillée d'activiti** est accessible au bout de l'URL suivante : <http://www.activiti.org/userguide/>

Services techniques d'activiti



Accès aux services techniques d'activiti :

```
ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();
RuntimeService runtimeService = processEngine.getRuntimeService();
RepositoryService repositoryService = processEngine.getRepositoryService();
TaskService taskService = processEngine.getTaskService();
ManagementService managementService = processEngine.getManagementService();
IdentityService identityService = processEngine.getIdentityService();
HistoryService historyService = processEngine.getHistoryService();
FormService formService = processEngine.getFormService();
```

Configuration de la base de données technique pour activiti

Fichier de configuration "**activiti.cfg.xml**"

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="processEngineConfiguration"
    class="org.activiti.engine.impl.cfg.StandaloneProcessEngineConfiguration">
    <property name="jdbcUrl"
      value="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=1000" />
    <property name="jdbcDriver" value="org.h2.Driver" />
    <property name="jdbcUsername" value="sa" />
    <property name="jdbcPassword" value="" />
    <property name="databaseSchemaUpdate" value="true" />
    <property name="jobExecutorActivate" value="false" />
  </bean>
</beans>
```

Ce fichier est par défaut chargé automatiquement.

On pourra éventuellement préférer une configuration "Spring" ordinaire au sein de laquelle une sous partie équivalente sera explicitement utilisée.

Chargement (sans spring) d'un processus activiti et démarrage d'une instance

```
public static void main(String[] args) {  
    try{  
        Map<String,Object> params = new HashMap<String,Object>();  
        //NB: each param/variable (with all sub-objects) must be serializable !!!  
        params.put("username","toto");        params.put("age",new Integer(12));  
        params.put("pers", new Personne("toto",44L));  
  
        // Create Activiti process engine  
        ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();  
  
        // Get Activiti services  
        RepositoryService repositoryService = processEngine.getRepositoryService();  
        RuntimeService activitiRuntimeService = processEngine.getRuntimeService();  
  
        // Deploy the process definition  
        repositoryService.createDeployment()  
        .addClasspathResource("test_activiti_with_props.bpmn").deploy();  
  
        ProcessInstance activitiProcessInstance =  
            activitiRuntimeService.startProcessInstanceByKey("myProcess", params);  
        String processInstanceId=activitiProcessInstance.getId();  
        System.out.println("started new processInstanceId="+processInstanceId);  
    } catch (Exception e) {    e.printStackTrace();  
    }  
}
```

Intégration explicite d'activiti au sein de Spring

NB: le très gros intérêt de la configuration de activiti au sein de spring tient dans le fait que tout composant spring pourra être référencé et utilisé dans un processus activiti (via ServiceTask ou scriptTask) .

Autrement dit , un script (java, javascript , groovy, ...) du processus ".bpmn2" pourra utiliser un composant Spring en tant qu'objet de traitement (exemple : pour appeler un service web , pour envoyer un mail ou encore pour envoyer un message dans une file d'attente JMS) .

service-activiti-spring.xml (exemple) :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
...
```


Configuration "activiti" intégrée à "spring" (suite 1)

```
<bean id="activitiDataSource"
  class="org.springframework.jdbc.datasource.SimpleDriverDataSource">
  <property name="driverClass" value="org.h2.Driver" />
  <property name="url" value="jdbc:h2:mem:activiti;DB_CLOSE_DELAY=1000" />
  <property name="username" value="sa" />
  <property name="password" value="" />
</bean>
```

```
<bean id="activitiTransactionManager"
  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="activitiDataSource" />
</bean>
```

```
<bean id="processEngineConfiguration"
  class="org.activiti.spring.SpringProcessEngineConfiguration">
  <property name="dataSource" ref="activitiDataSource" />
  <property name="transactionManager" ref="activitiTransactionManager" />
  <property name="databaseSchemaUpdate" value="true" />
  <property name="jobExecutorActivate" value="false" />
</bean>
```

<!-- tout ce qui précède est un équivalent spring de activiti.cfg.xml →

...

Configuration "activiti" intégrée à "spring" (suite 2)

```
<bean id="processEngine" class="org.activiti.spring.ProcessEngineFactoryBean">  
  <property name="processEngineConfiguration" ref="processEngineConfiguration" />  
</bean>
```

```
<bean id="repositoryService" factory-bean="processEngine"  
      factory-method="getRepositoryService" />  
<bean id="runtimeService" factory-bean="processEngine"  
      factory-method="getRuntimeService" />  
<bean id="taskService" factory-bean="processEngine" factory-method="getTaskService" />  
<bean id="formService" factory-bean="processEngine" factory-method="getFormService" />  
<bean id="historyService" factory-bean="processEngine" factory-method="getHistoryService" />  
<bean id="managementService" factory-bean="processEngine"  
      factory-method="getManagementService" />
```

<!-- autres beans accessibles depuis le processus (.bpmn): -->

```
<bean name="dynSoapClient" class="generic.ws.util.client.DynReflectSoapClient" />  
  <!-- utilisation: dynSoapClient.dynSoapCall(wsUrl, serviceInterfaceName,  
                                              methodName, soapArgs); -->  
  
<!-- et aussi  
<bean id="javaJmsClient" class="generic.async.jms.util.GenericJavaJmsClient" >  
  de jms-spring.xml -->  
</beans>
```

Accès aux services techniques d'activiti via spring

```
public static void main(String[] args) {  
    try {  
        ClassPathXmlApplicationContext applicationContext = new  
            ClassPathXmlApplicationContext("service-activiti-spring.xml");  
        RepositoryService repositoryService = (RepositoryService)  
            applicationContext.getBean("repositoryService");  
        RuntimeService activitiRuntimeService = (RuntimeService)  
            applicationContext.getBean("runtimeService");  
        ...  
    }  
}
```

* Le service technique "**repositoryService**" sert à gérer la persistance (base de données d'activiti) et sert à charger la définition d'un processus (fichier .bpmn2) en mémoire.

* Le service "**runtimeService**" sert à démarrer et contrôler une instance d'un certain processus.

Configuration "maven" pour "activiti"

pom.xml

```
... <repositories> <repository>
  <id>Alfresco_Maven_Repository</id>
  <url>http://maven.alfresco.com/nexus/content/groups/public/</url>
</repository> </repositories>
<dependencies> <dependency>
  <groupId>org.activiti</groupId> <artifactId>activiti-engine</artifactId>
  <version>5.12</version>
</dependency>
<dependency>
  <groupId>org.activiti</groupId> <artifactId>activiti-spring</artifactId>
  <version>5.12</version>
</dependency>
<dependency>
  <groupId>org.codehaus.groovy</groupId>
  <artifactId>groovy-all</artifactId> <version>2.1.3</version>
</dependency>
<dependency>
  <groupId>com.h2database</groupId> <artifactId>h2</artifactId>
  <version>1.3.170</version>
</dependency>....</dependencies>...
```

Chargement de la définition d'un processus bpmn

```
String deploymentId = repositoryService  
    .createDeployment()  
    .addClasspathResource("my_process.bpmn")  
    .deploy().getId();
```

// si le fichier "my_process.bpmn" comporte en interne

// l'id "myProcess", une instance pourra ensuite être démarrée

// via activitiRuntimeService.startProcessInstanceByKey("myProcess", ...) ;

Démarrage d'une instance d'un processus activiti

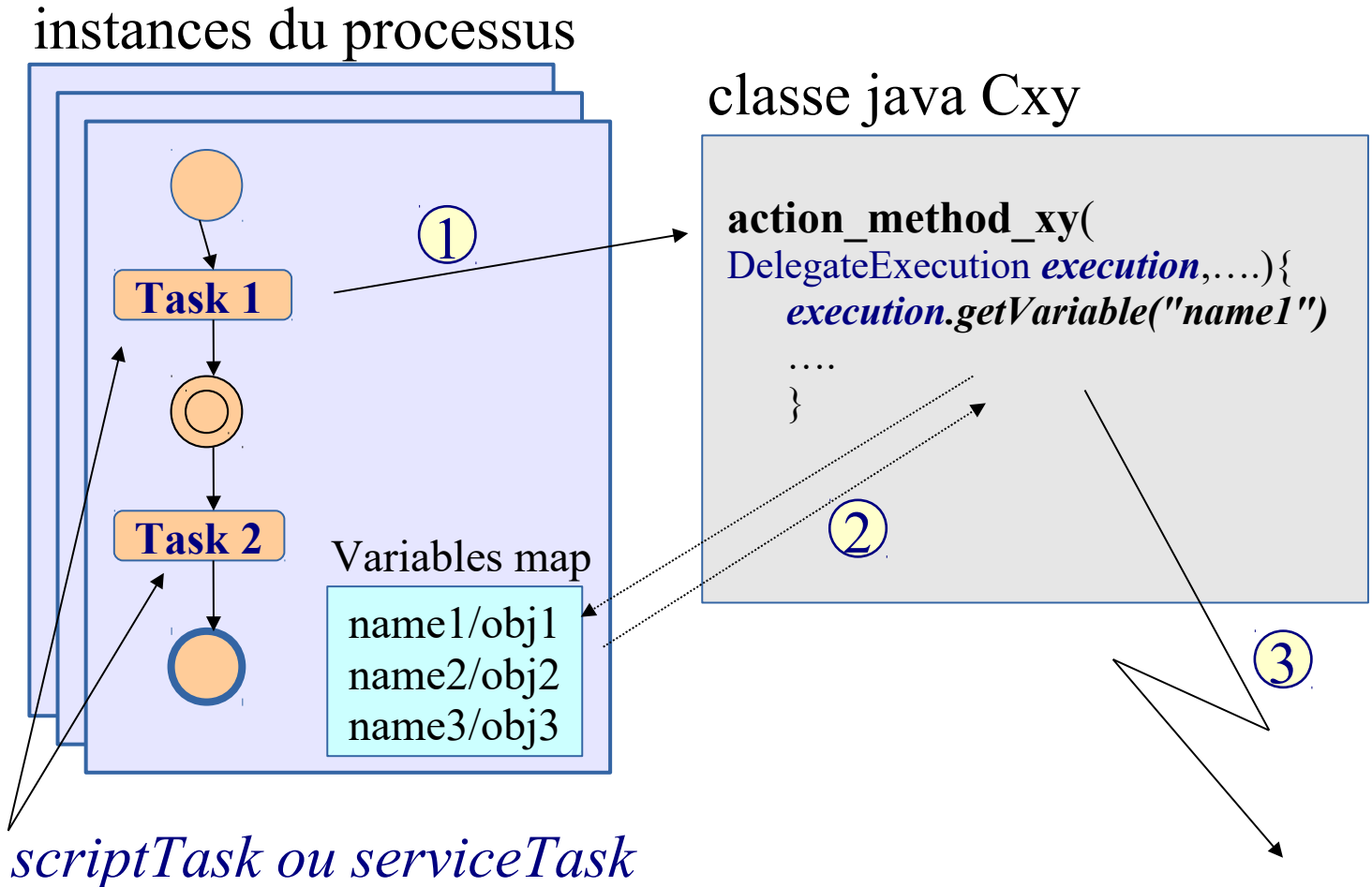
```
Map<String,Object> initParams = new HashMap<String,Object>();  
params.put("pers", new Personne("toto",44L));
```

```
ProcessInstance activitiProcessInstance =  
    activitiRuntimeService.startProcessInstanceByKey(  
        "myProcess", initParams);
```

```
// dès le démarrage ---> initialisation dans éventuel  
// ActivitiStartProcessListener si défini dans .bpmn
```

```
String processInstanceId=activitiProcessInstance.getId();  
System.out.println("started (with spring) new processInstanceId="  
    +processInstanceId);
```

Programmation d'actions (vers l'extérieur)



Délégation java (pour actions)

```
public class ActivitiBpmContextWithDataAccess {  
    ...  
    public Object doAction(DelegateExecution execution,  
                           String actionName){  
        Object result=null;  
        logger.debug("activiti-do- actionName=" + actionName +  
                     " , instanceId : " + execution.getProcessInstanceId());  
  
        // code java (au cas par cas) de l'action à externaliser  
  
        // varXy = (ClasseXY) execution.getVariable("varXy");  
        // execution.setVariable(varName, value);  
        return result;  
    } ...}
```

Cette classe de traitement peut être vue comme un composant spring de nom "**ctx**" ou autre. Et dans ce cas une tâche de type java/javascript/groovy pourra simplement effectuer une délégation d'action java via une syntaxe du type `ctx.doAction(execution, "actionXY");`

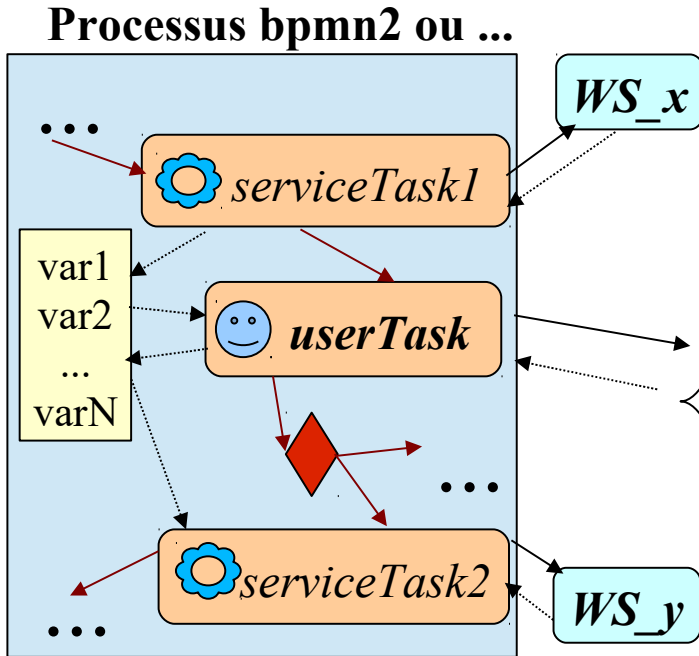
Logique conditionnelle (flow condition)

$\$ \{ \textit{pers.age} < 18 \}$

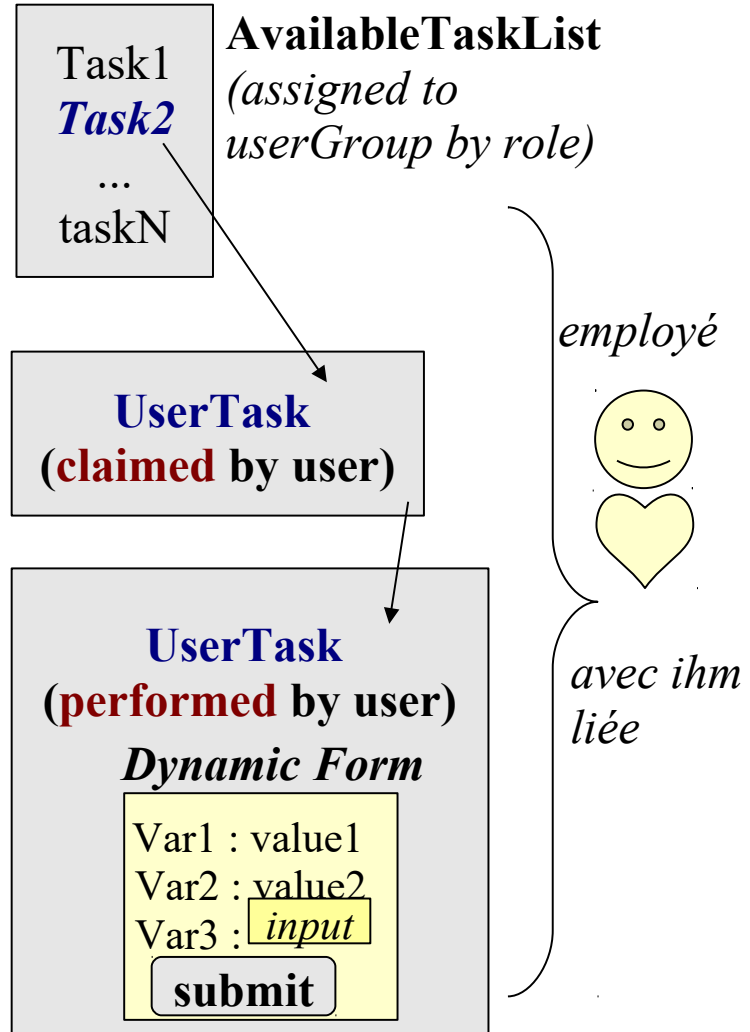
$\$ \{ \textit{statut} \neq \textit{"ok"} \}$

...

UserTask (bpmn,...)



*UserTask en exécution
(potentiellement longue)
tant que pas finie ou annulée*



Sous processus logique d'une "UserTask"

Une "UserTask" peut être associée à un groupe d'individus et dans ce cas :

chaque membre du groupe aura l'occasion de "prendre en charge" cette tâche (**claim**: la réclamer) en choisissant celle ci parmi les tâches disponibles

une fois réclamée , cette tâche sera affectée plus précisément à un certain utilisateur qui pourra alors :

- * remplir un formulaire (avec des infos en lecture et des champs à renseigner)

- * soumettre ce formulaire et ainsi compléter cette "UserTask"

Si dès le départ , une "UserTask" est affectée précisément à un individu , il n'y a pas besoin de réclamer la tâche.

Paramétrage d'une UserTask au sein du processus bpmn

→ pour définir la structure du formulaire généré dynamiquement .

```
<userTask id="usertask1" name="UserTaskSaisirAge"
activiti:candidateUsers="kermit"
activiti:candidateGroups="management">
  <documentation>test userTask</documentation>
  <extensionElements>
    <activiti:formProperty id="age" name="age" type="long"
      expression="#{pers.age}"></activiti:formProperty>
    <activiti:formProperty id="username" name="username"
      expression="#{pers.username}"
      writable="false"></activiti:formProperty>
  </extensionElements>
</userTask>
```

La partie `expression="#{objName.subPart}"` permet de coder des associations avec certaines variables du processus bpmn et les noms/id seront exploités coté "formulaire de saisie" .

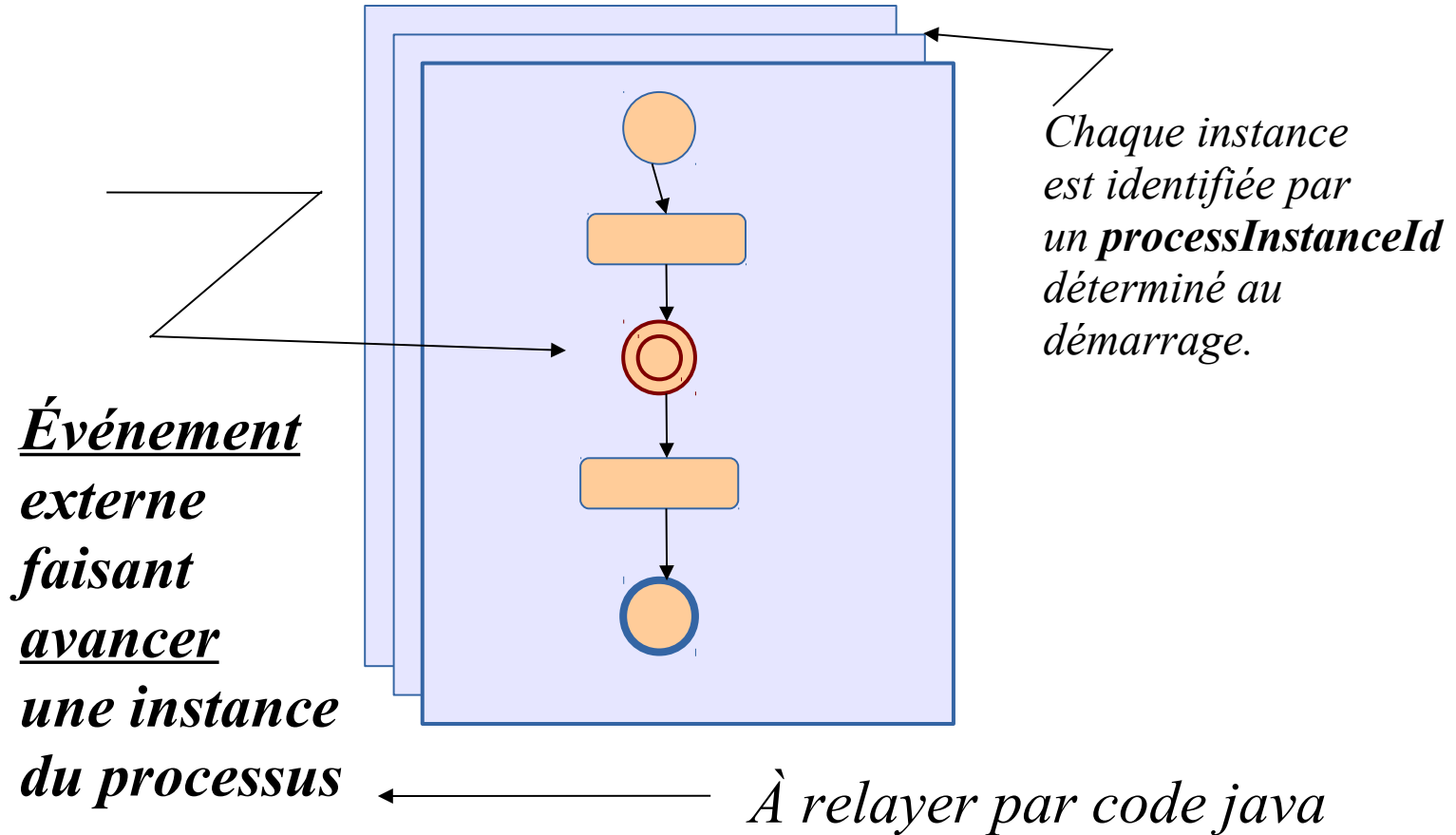
Activiti offre le moyen de personnaliser l'interface graphique du sous processus "UserTask".

On peut ainsi bâtir une IHM avec Struts , JSF ou GWT .

Le code java de l'interface graphique doit s'appuyer sur les api "**taskService**" et "**formService**" d'activiti .

Événements pour faire avancer un processus

instances d'un processus xy.bpmn



Evénements (bpmn / déclenchement via activiti)

2 types d'événements "bpmn" :

- * **signal** : sans paramètre
- * **message** : avec paramètre(s)

Exemple de définition de message dans un processus bpmn :

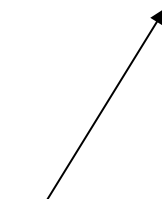
```
<definitions...>
  <message id="reponseConges" name="reponseConges"></message>
  <process id="myAsyncProcess" name="myAsyncProcess"
    isExecutable="true">
    ...
    <intermediateCatchEvent id="messageintermediatecatchevent1"
      name="MessageCatchEvent">
      <messageEventDefinition messageRef="reponseConges" />
    </intermediateCatchEvent>
    ....
  </process>
</definition>
```

Déclencher un événement "signal"

```
public class ActivitiBpmEventManager /* classe utilitaire */ {  
  
    private RuntimeService activitiRuntimeService;  
  
    public void setActivitiRuntimeService(RuntimeService activitiRuntimeService) {  
        this.activitiRuntimeService = activitiRuntimeService;  
    }  
  
    public void signalEvent(String processName /* Id */ , String processInstanceId ,  
        String signalName ){  
        try {  
            Execution execution = activitiRuntimeService.createExecutionQuery()  
                .processDefinitionKey(processName /* Id */)   
                .processInstanceId(processInstanceId)  
                .signalEventSubscriptionName(signalName).singleResult();  
            activitiRuntimeService.signalEventReceived(signalName,execution.getId());  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```


Déclencher un événement "message"

```
public class ActivitiBpmEventManager /* classe utilitaire */ {  
    ...  
  
    public void messageEvent(String processName , String processInstanceId ,  
                               String eventName ,  
                               Map<String,Object> newProcessVariablesParams){  
        try {  
            Execution execution = activitiRuntimeService.createExecutionQuery()  
                .processDefinitionKey(processName)  
                .processInstanceId(processInstanceId)  
                .messageEventSubscriptionName(eventName).singleResult();  
            activitiRuntimeService.messageEventReceived(eventName,execution.getId(),  
                                                         newProcessVariablesParams);  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```



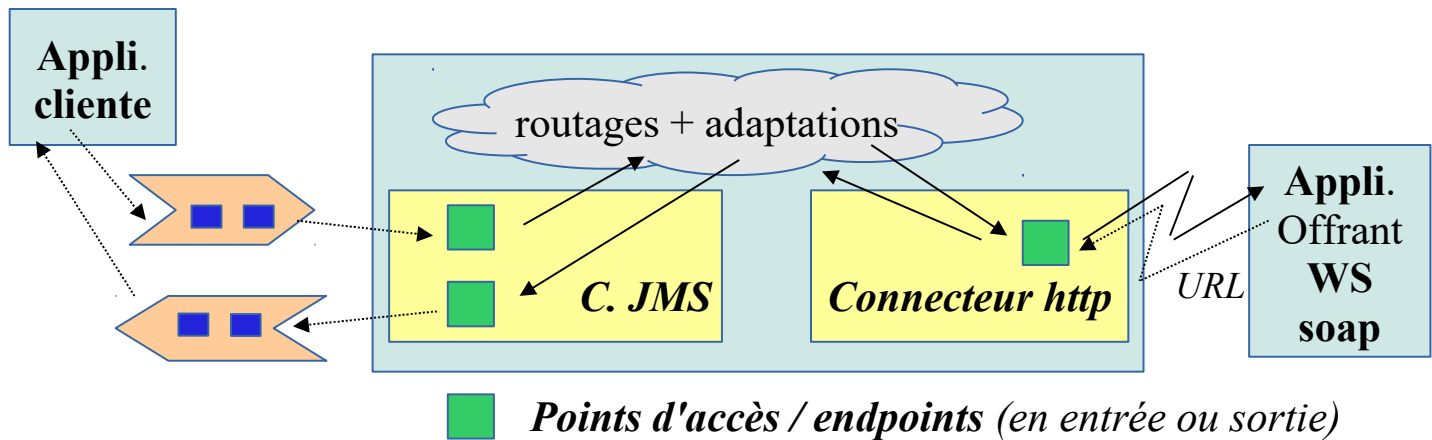
*Paramètres du message
(nouvelles variables du processus)*

ESB (présentation)

- 1) concepts EAI / ESB – diapo 203
- 2) présentation de serviceMix et MuleESB – diapo 243

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*)

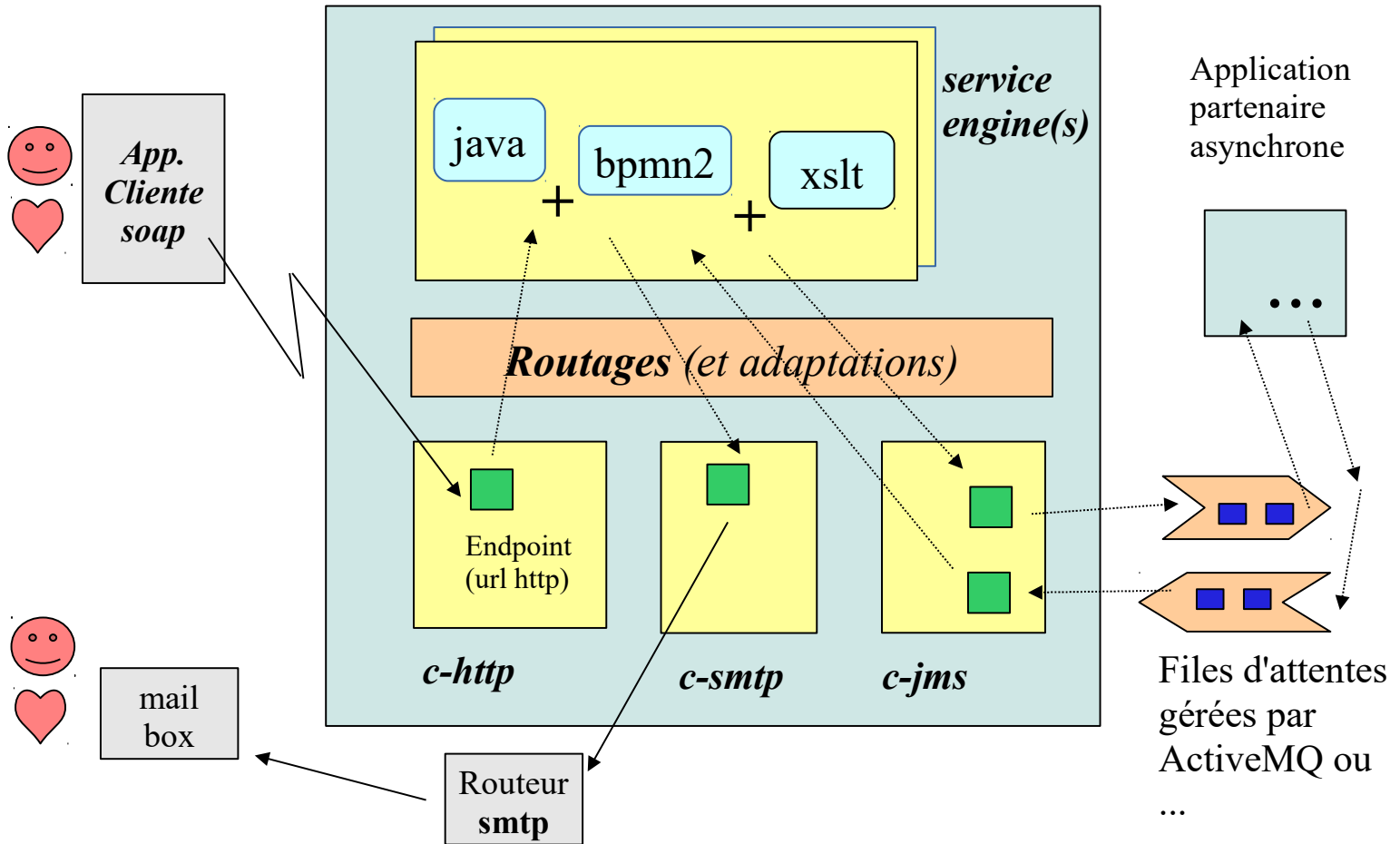
essentielle

*éventuelle
intégration
de bpm ou bpmn2*

*potentiellement
serveur d'applications
(comme jee)*

Exemple d'intégration et de communications autour d'un ESB

ESB (ServiceMix/Camel , Mule , ...)



ESB et framework d'intégration

Taille / complexité / prix

Ex: suite *WebMethod* , ...

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

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

framework
d'intégration

ESB
(*serveur*)

Suite
d'intégration
Complète

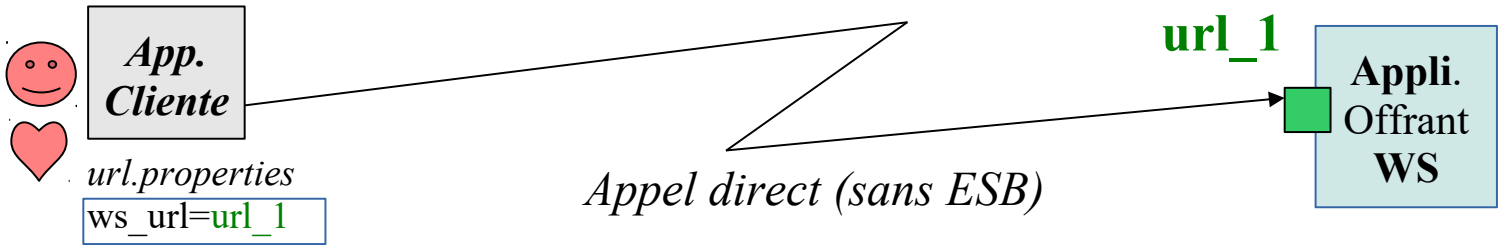
(ESB +
annuaire UDDI
+ sécurité
SAML + ...)

includ

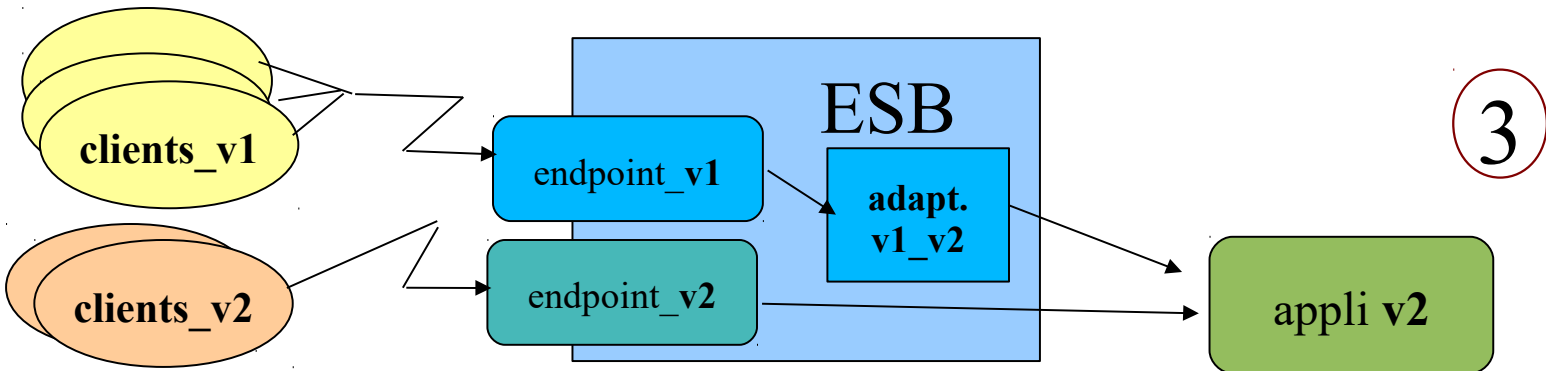
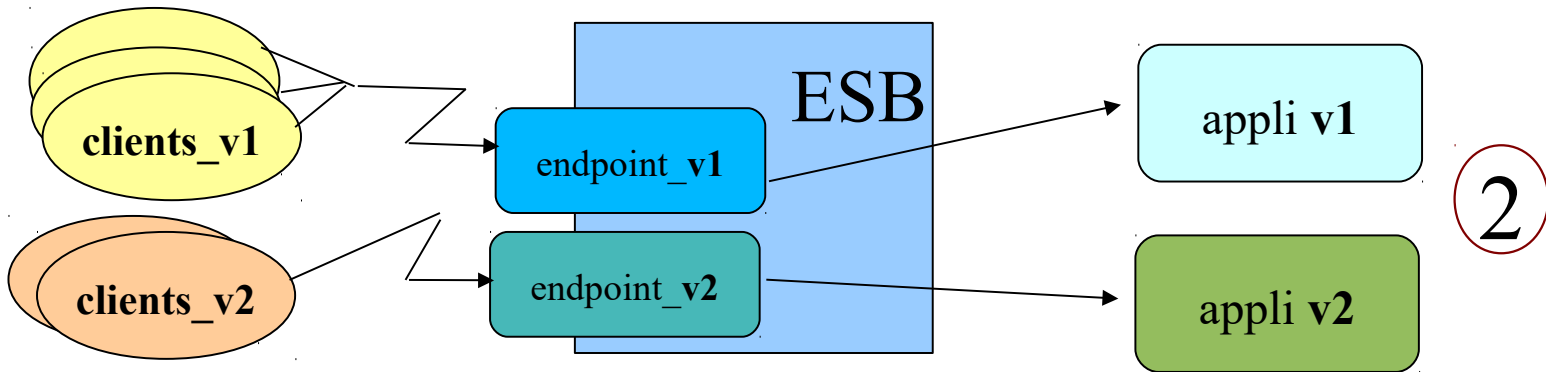
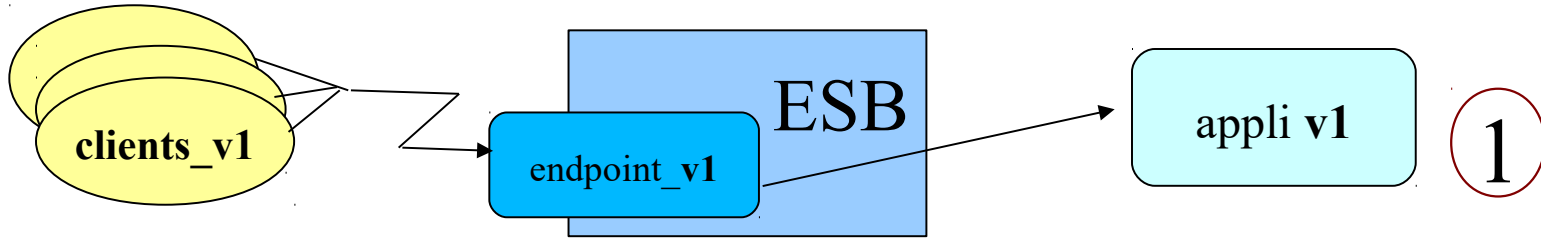
includ

Degré
d'intégration

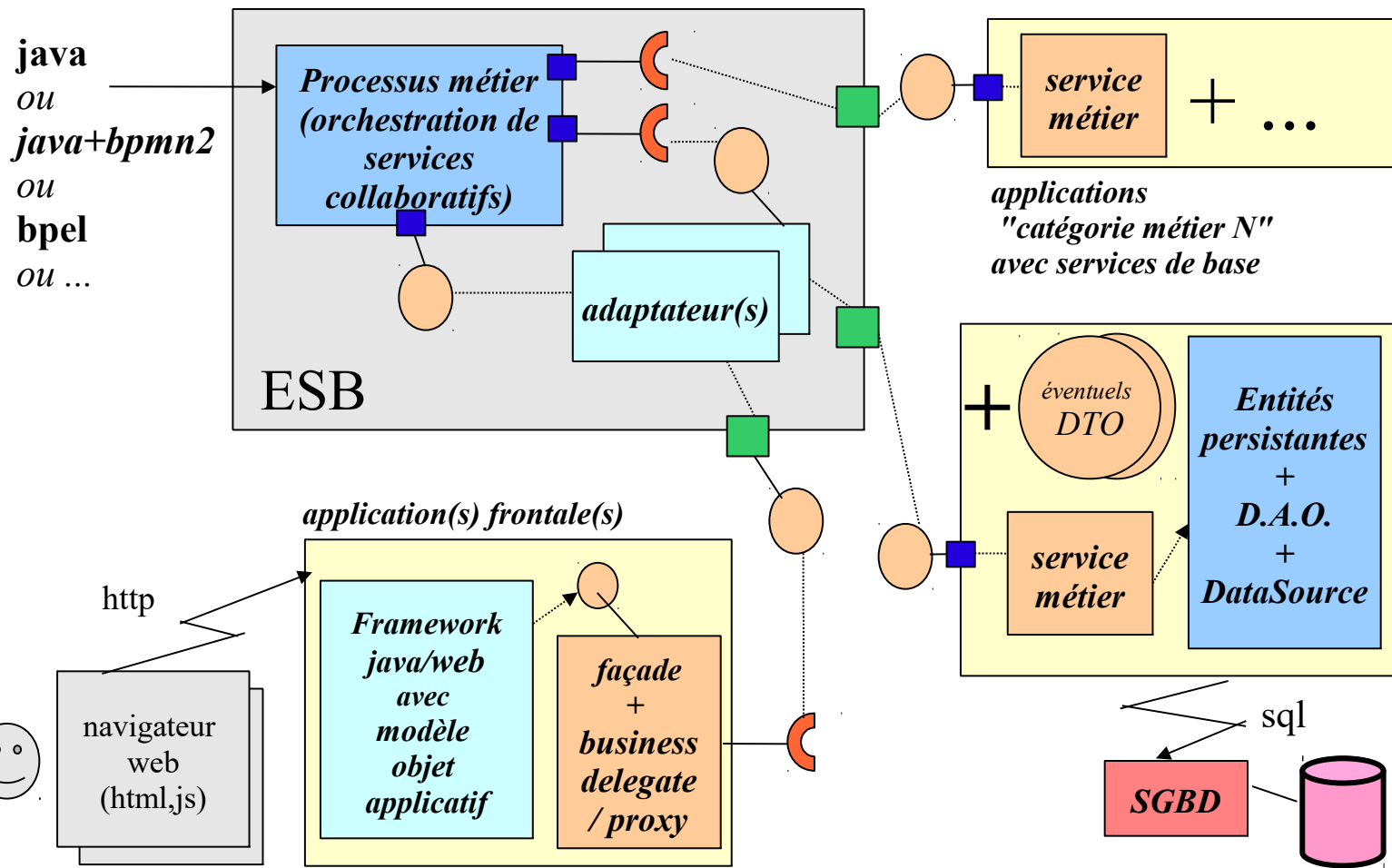
Trois variantes fonctionnellement identiques :



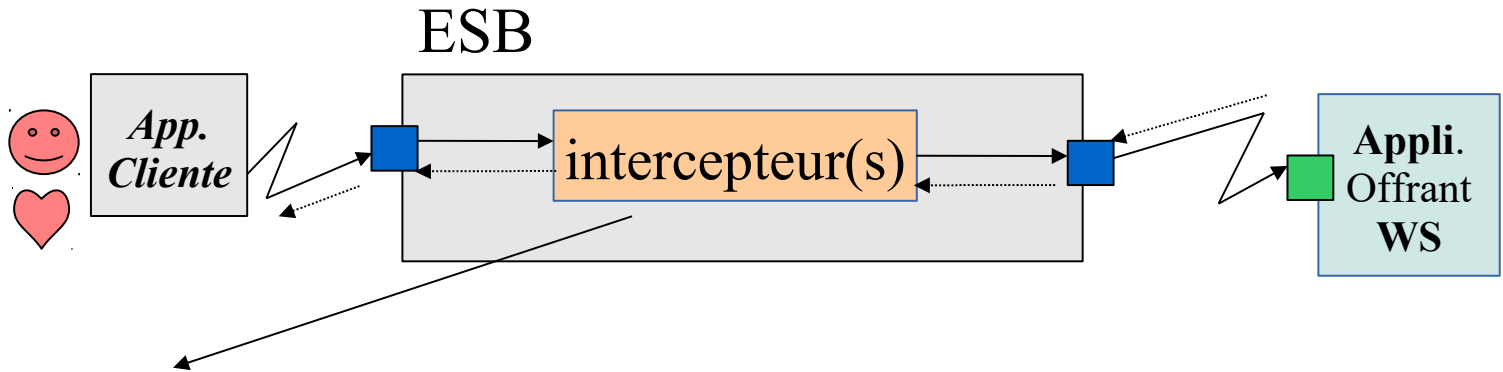
Exemple d'évolution de versions en douceur



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**= **K**ey **P**erformance **I**ndicator)

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) .*

EAI, ESB

E.A.I. = Enterprise Application Integration :

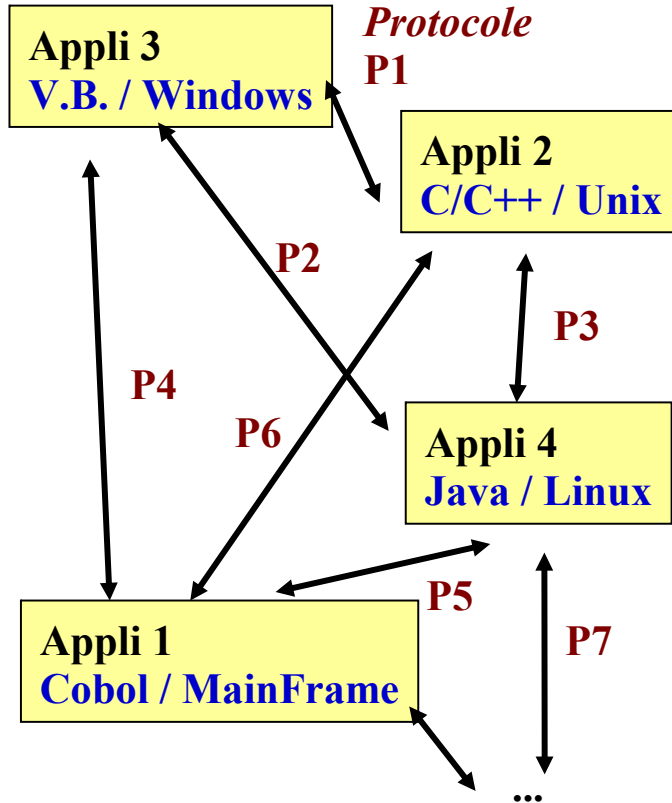
- Un **E.A.I.** est essentiellement un concept qui vise à fédérer (à un certain niveau du S.I.) la plupart des communications entre les différentes applications de l'entreprise.
- Plus de connexions directes mais un système intermédiaire logiquement centralisé qui re-route (et transforme si besoin) les messages.

E.S.B. = Enterprise Service Bus :

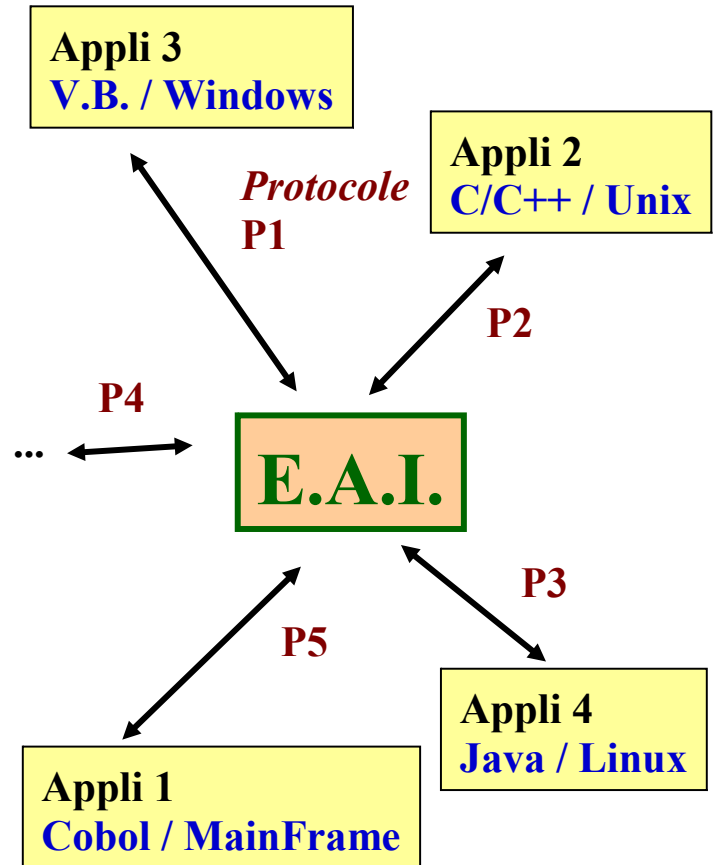
Un E.S.B. est (entre autres) une implémentation d'E.A.I. qui est:

- * compatible avec les technos Services Web (WSDL, SOAP)
- * structurée sous forme de BUS (interopérable avec d'autres Bus de type "ESB")

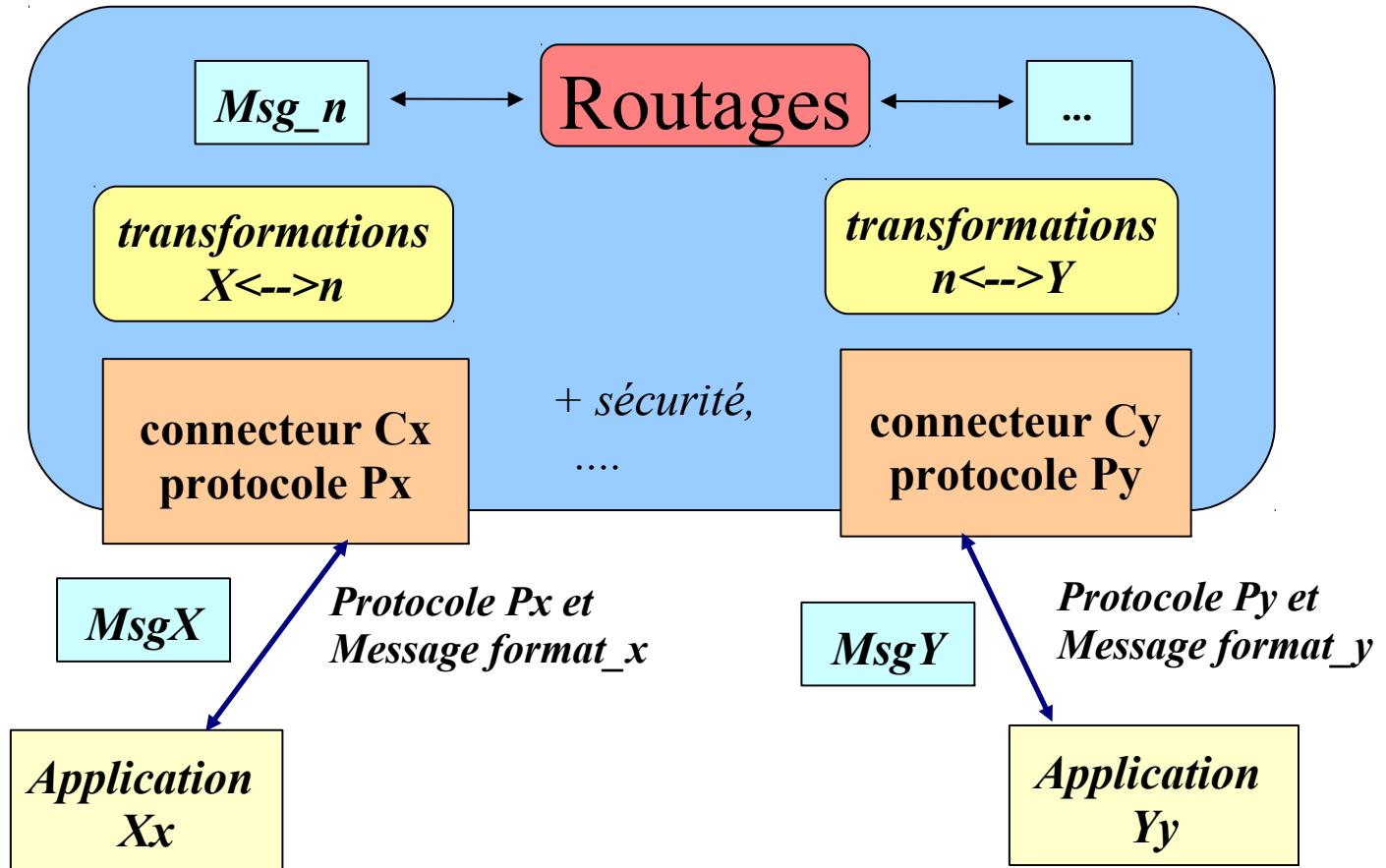
Sans E.A.I.



Avec E.A.I.



Fonctionnalités d'un EAI:

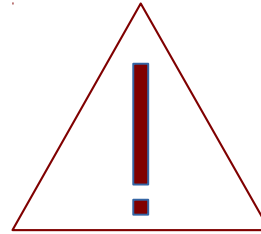
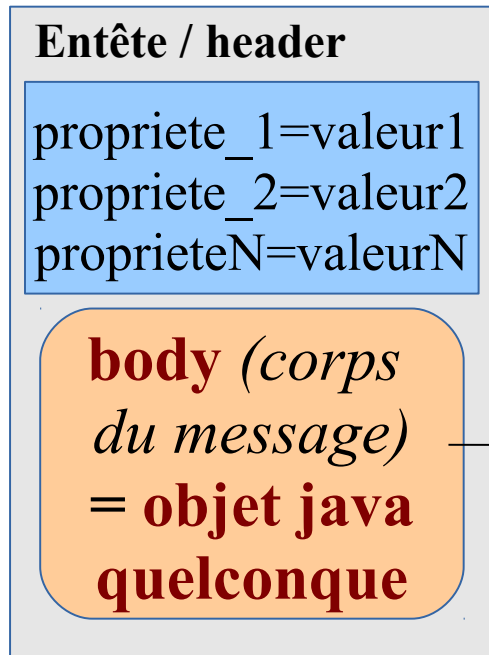


Transformations techniques internes à l'EAI/ESB

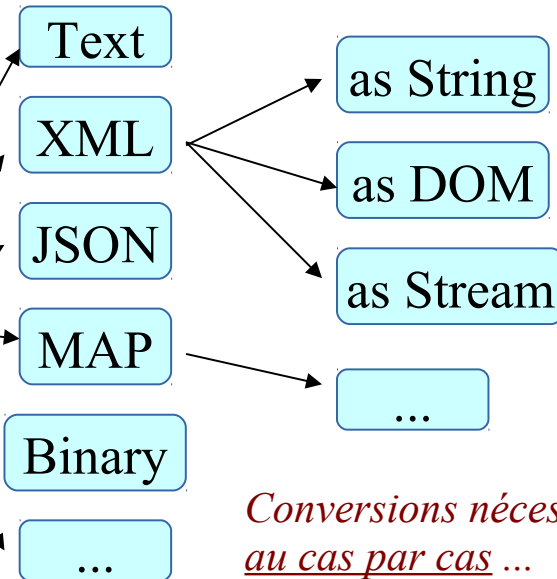
- * Si au cas par cas : potentiellement **$O(n^2)$**
transformations directes
[complexe , performant]
- * Si format interne "*neutre*" ou "*normalisé*"
ou "*canonique*" ($x \leftrightarrow n \leftrightarrow y$)
→ **$2*n$** *doubles transformations*
[simple , moins performant]

Cas fréquent des ESB basés sur java :

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



Pas de format
uniforme / canonique
pour la partie "body"



*Conversions nécessaires
au cas par cas ...*

Grands traits des premiers EAI:

- Xml quelquefois (souvent) 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 , ...).

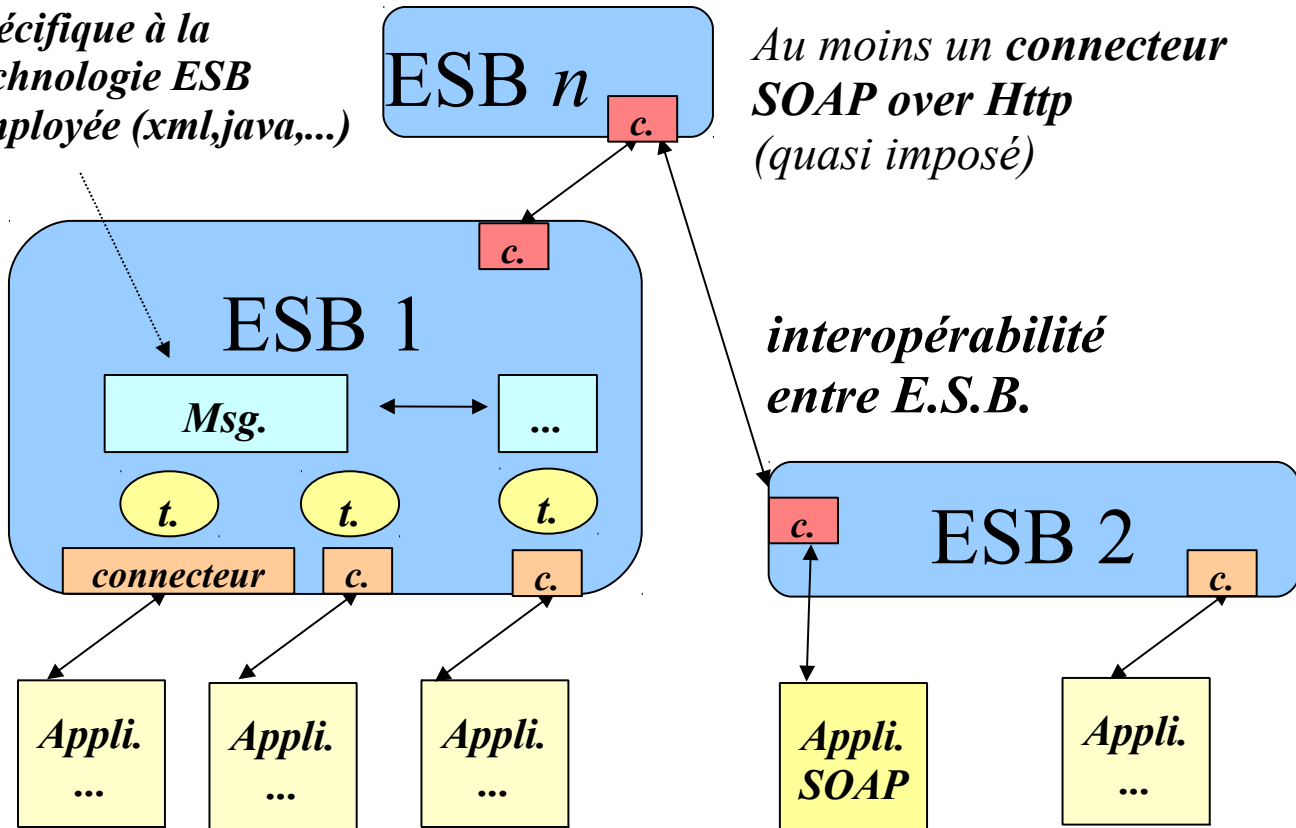
ESB (Enterprise Service Bus)

*Logiquement
décentralisé !*

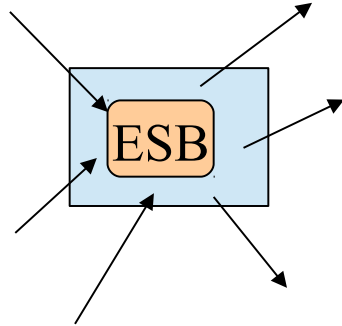
*Format interne
spécifique à la
technologie ESB
employée (xml,java,...)*

*Au moins un **connecteur**
SOAP over Http
(quasi imposé)*

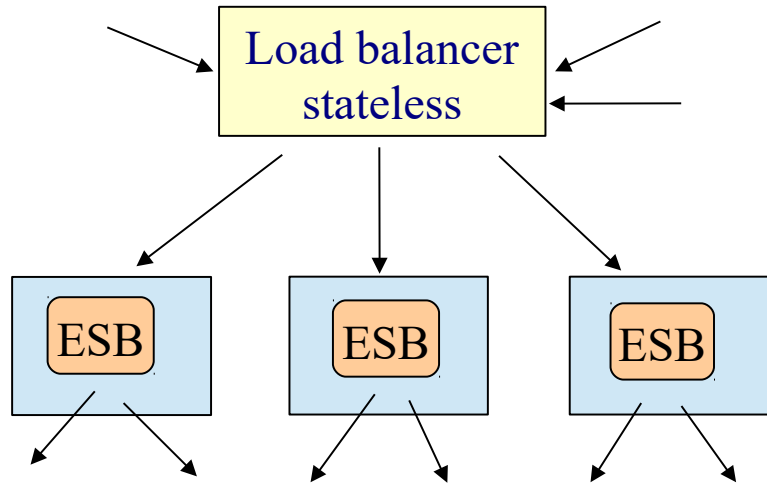
*interopérabilité
entre E.S.B.*



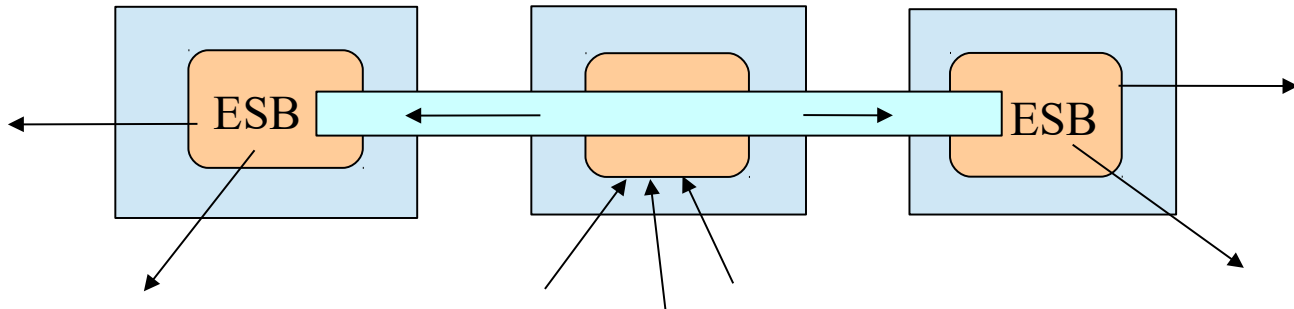
Topologies / architectures possibles



Charge limitée



Via canal inter-machine (ex : JMS)



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/paramétrer**:

- * 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

Référentiel interne des "endpoints" (url), routes et définition de services (interfaces java ● — , wsdl, ...) pris en charge par l'ESB

Yy.wsdl

**inbound
endpoint
(url soap)**

Connecteur cxf

Soap over http

*Application cliente qui appelle via soap le service **Yy?wsdl** (ou service transformé)*

*ajustements
internes*

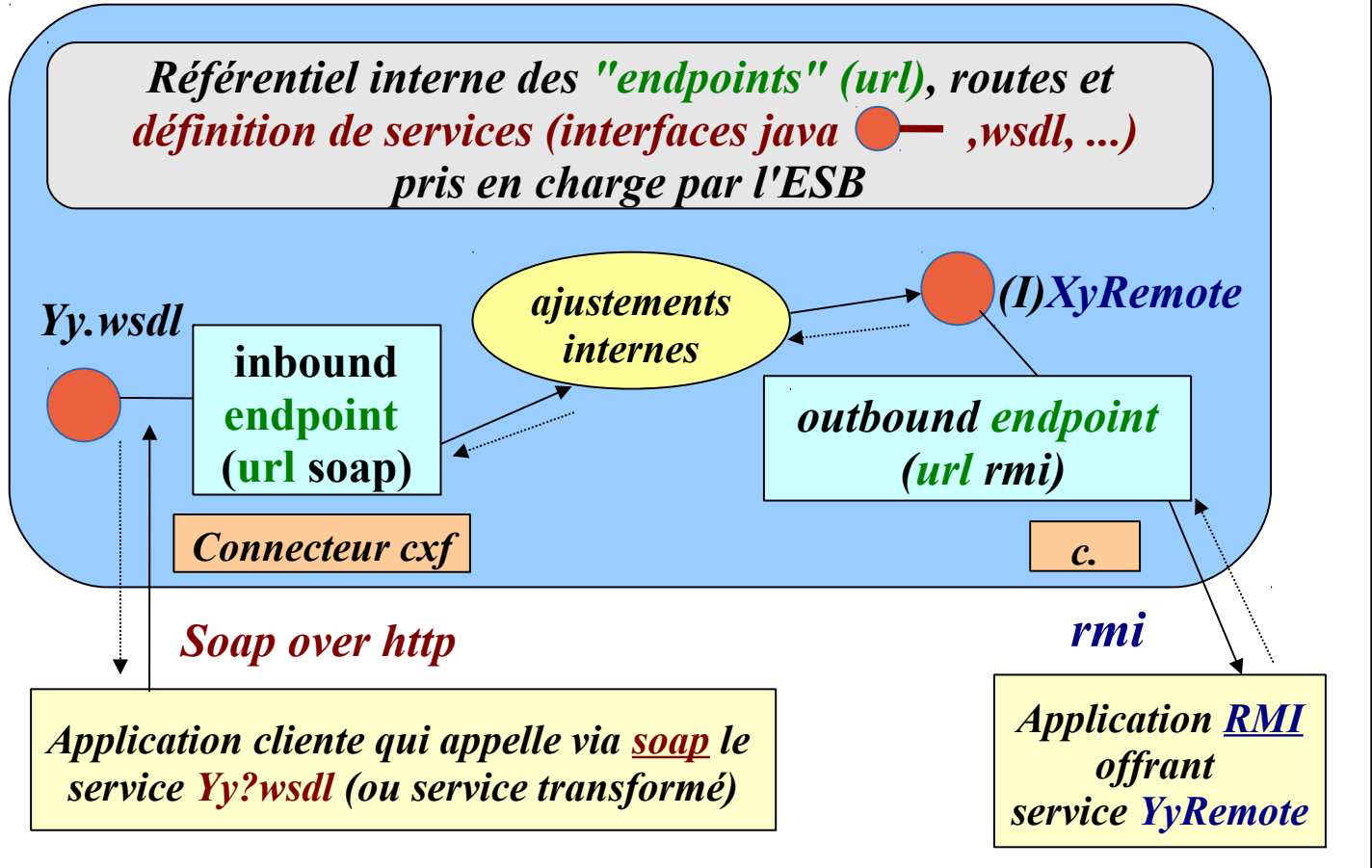
(I)XyRemote

**outbound endpoint
(url rmi)**

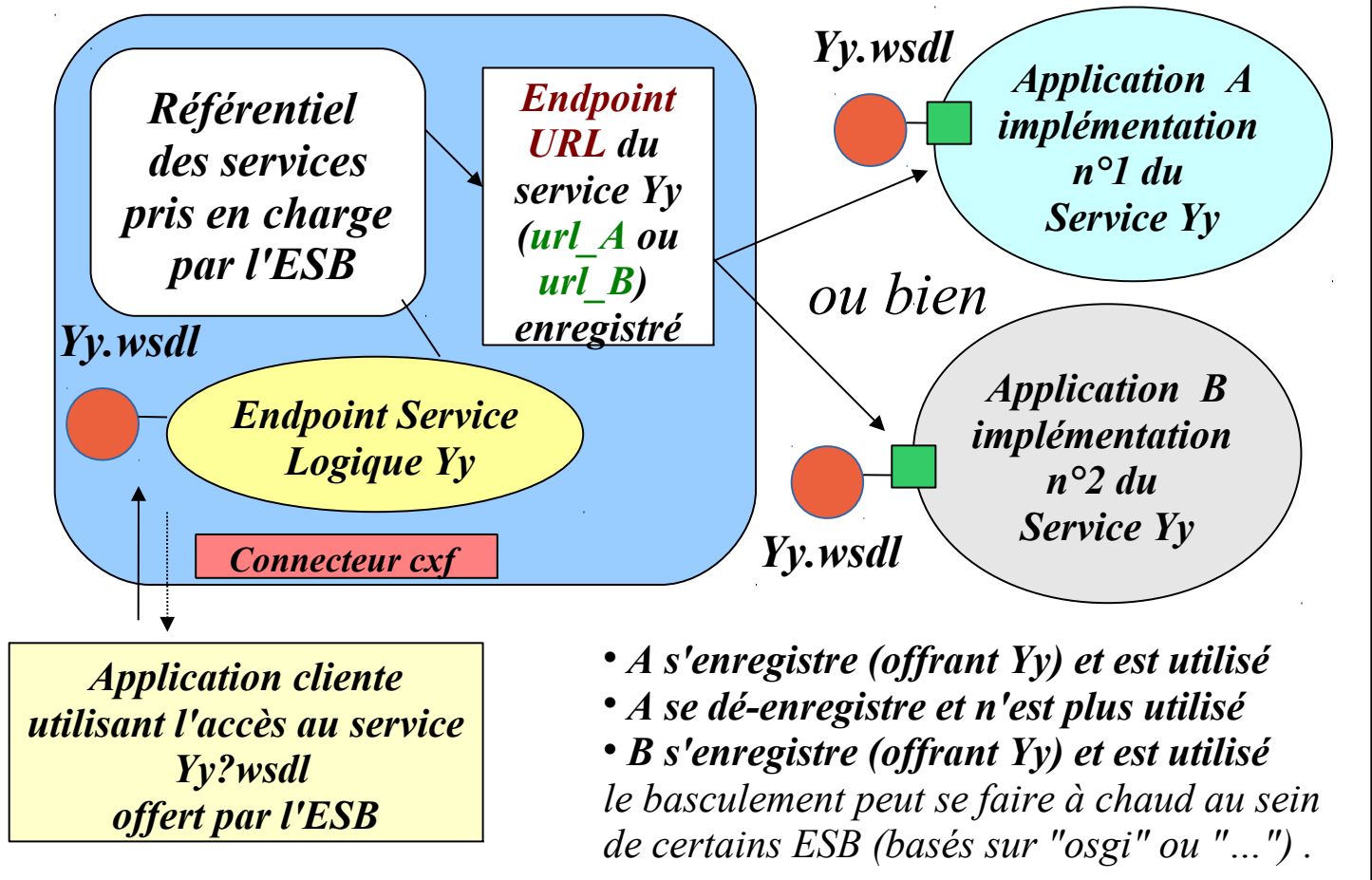
c.

rmi

*Application RMI
offrant
service **YyRemote***



ESB: basculement transparent d'implémentation



Moteurs de services (*exécution/interprétation de code, intégration de technologie*)

Moteur de Services BPEL

Service Xxx.bpel

*(instructions BPEL
à interpréter)*

Moteur de Services Java

Service Yyy.java

*(annotations jax-ws
à interpréter)*

Moteur ... technologieN

(code à interpréter)

- * Un **ESB** peut éventuellement **héberger des services internes (lignes de code)** au lieu de déléguer les appels vers des services externes
- * La sous-partie d'un ESB qui est capable d'interpréter des instructions dans une technologie donnée est appelée "**moteur de service**"
- * Lorsque c'est possible on tentera de séparer au sein de l'ESB l'accès au service rendu et son implémentation (via techno N) et en configurant des routes flexibles

Quelques utilités des "moteurs de services :

* 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

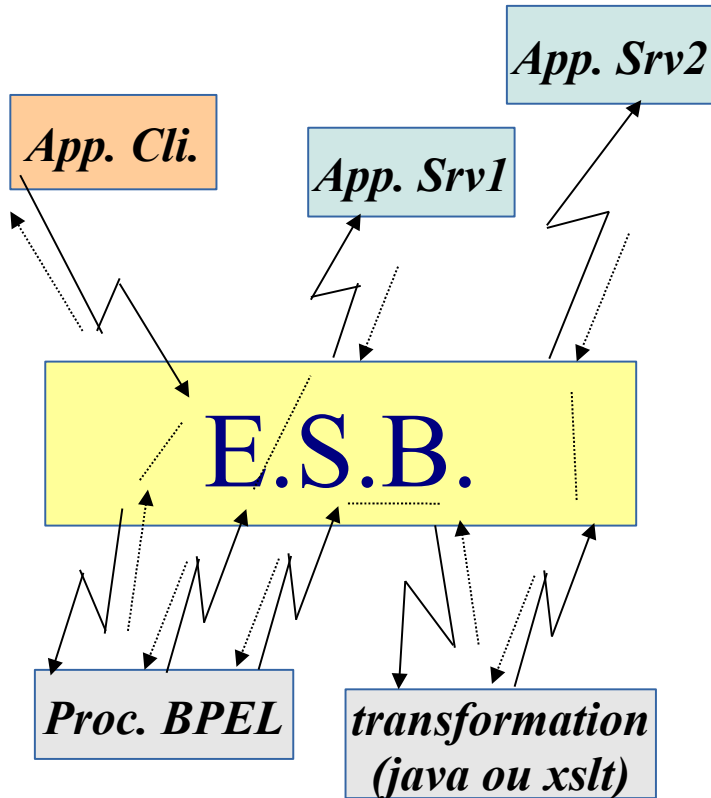
Les technologies "*BPEL*" et "*java/bpm2 (jbpm , activiti)*" nécessitent des moteurs d'exécutions (interpréteurs) spécifiques pour faire fonctionner des services d'orchestration (*invoquant des services élémentaires selon un algorithme/processus convenu*)

* Mise en forme de messages textes

(application d'un template velocity , ...)

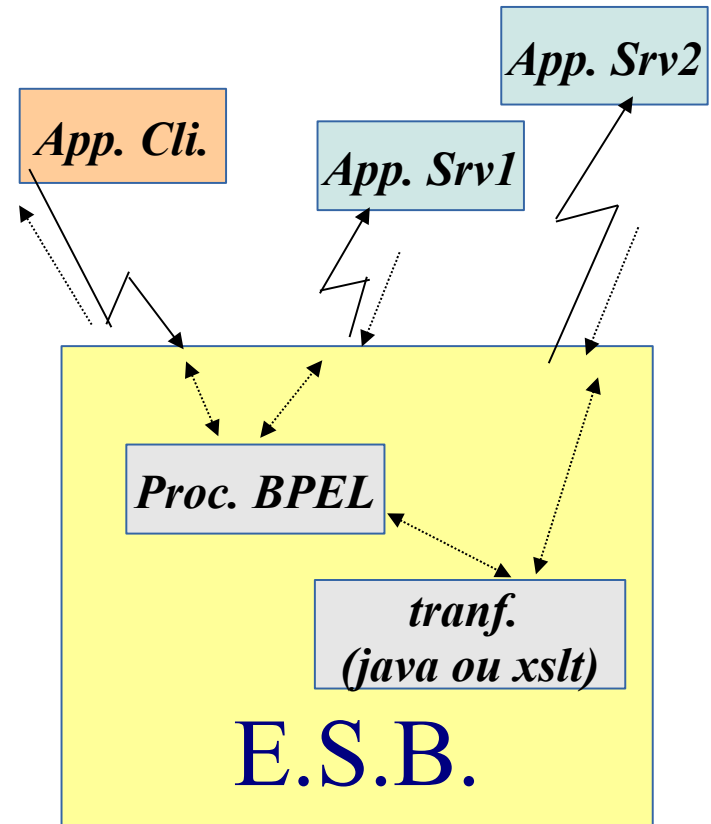
* Divers autres traitements (extractions/insertions sql , ...)

Avec moteurs de services *externes*



**Trop de traversées du réseau
/ mauvaises performances**

Avec moteurs de services *internes* (ESB)



**Meilleure intégration
/ performances améliorées**

Limitation actuelle importante (à connaître) des ESB
"open source" *ServiceMix* et *MuleESB* :

Ces 2 ESB (basés sur une architecture java légère)
ne permettent pas d'intégrer un lourd moteur BPEL.

Pour effectuer de l'orchestration de service au sein de
ces ESB (sans trop de traversées réseaux),
il vaut mieux s'orienter vers les technologies "**jbpm**" ou
bien "**activiti**".

*NB : Les anciennes versions de serviceMix (3.x , 4.1) étaient capables
d'intégrer ODE_bpel via la complexe technologie JBI (aujourd'hui
considérée comme "has been"). Ceci n'est plus possible avec les nouvelles
versions de servicemix.*

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/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, ...)

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

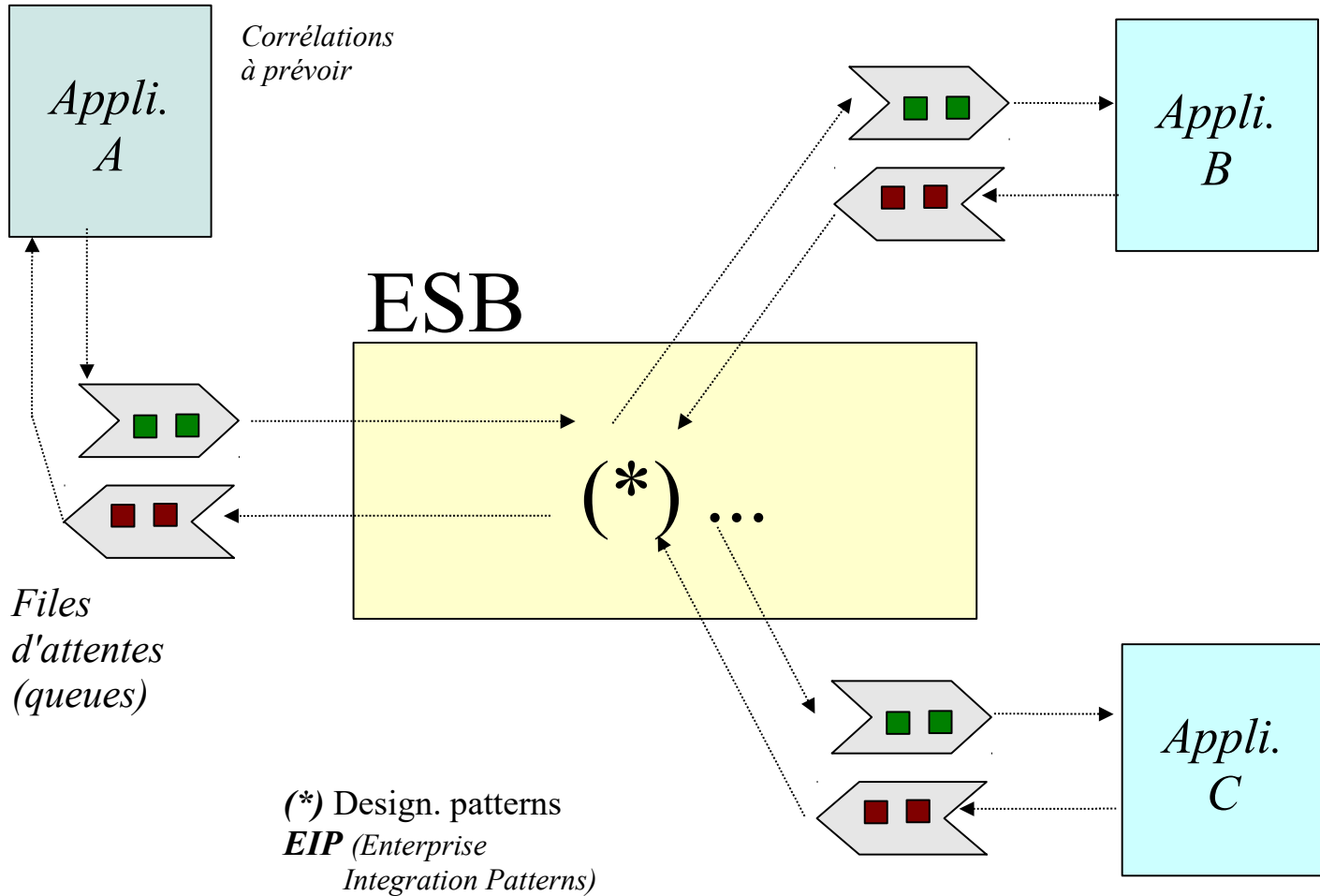
Cette structure peut être véhiculée par :

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

==> passerelles techniques envisageables entre :

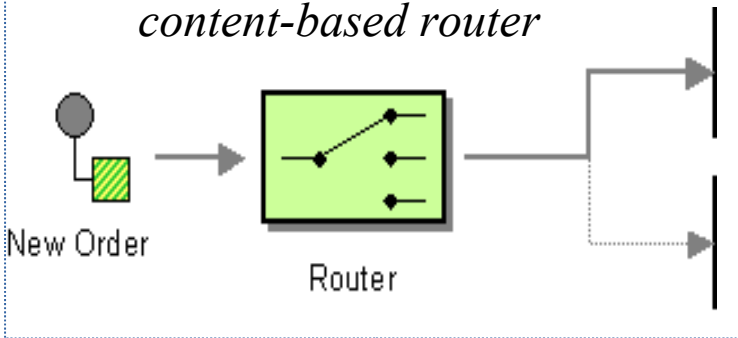
- * *synchrones* et *asynchrones*
- * *document* et *RPC* (Remote Procedure Call) .

ESB en mode asynchrone

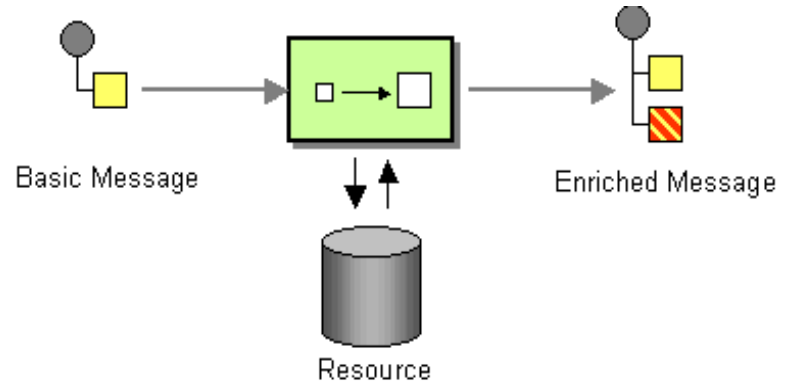


Quelques "design patterns" EIP (*Enterprise Integration Patterns*)

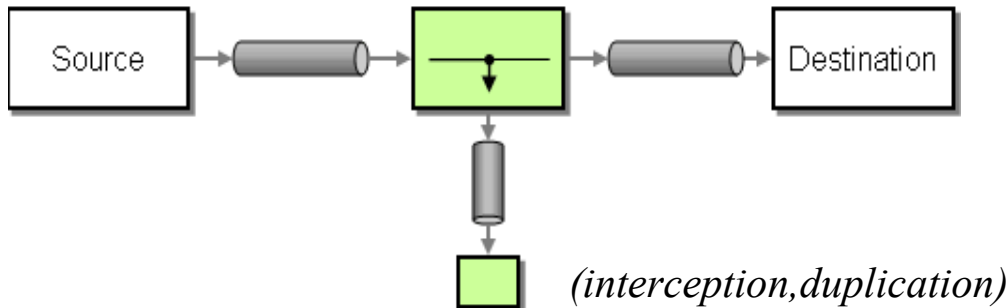
content-based router



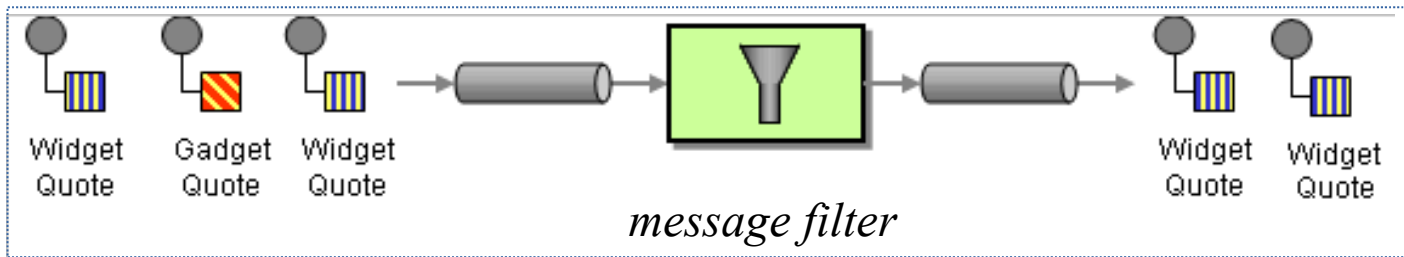
content enricher



wire tap



message filter



Quelques catégories d'ESB (1)

Open source

Commerciaux

Mixte (selon support / extensions)

ESB / Serveur complet
(autonome ,
quelquefois lourd ,)

Websphere ESB (IBM)

Tibco ESB (Tibco)

Oracle ESB (Oracle)

WebMethod (...), ***Talend ESB*** , ...

Petals ESB (OW2)

ServiceMix (Apache)

... (...)

Mule ESB
(MuleSoft)

Spring integration (...)

Camel (...)

Framework d'intégration
(ou sous partie d'ESB) facilement
ré-intégré au sein
d'autres serveurs (JEE ou ...)

Principaux ESB commerciaux

<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>). Architecture assez propriétaire.
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 de grandes marques (à ne pas confondre avec les ESB open-source)

Principaux ESB "open source" (ou mixte)

ESB	Editeur	Caractéristiques
OpenESB, "has been" ServiceMix [3.x , 4.3] Petals	SUN, Apache, OW2	JB/I et "has been"
ServiceMix >= 4.4 (camel en interne)	Apache	OSGi (quasiment plus JB/I), plus d'intégration possible de ODE/BPEL, mais intégration possible de activités
FuseESB	FuseSource puis 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

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.

Quelques catégories d'ESB (2)

Routages via *invocations explicites* (consumer/provider) entre "*endpoints internes*" (ex: JBI/SCA/...)

Spécifications
S.C.A.

Websphere
ESB (IBM)

Tuscany (Apache)

... (...)

Spécifications
J.B.I.

Petals ESB (OW2)

anciens
ServiceMix (Apache)
... (...)

...

Spécifications
WCF

.net
(MICROSOFT)

Routages via *flux de messages* (avec séquençement contrôlé de transformations, interceptions, duplications, ...)

...


Mule ESB
(MuleSoft)

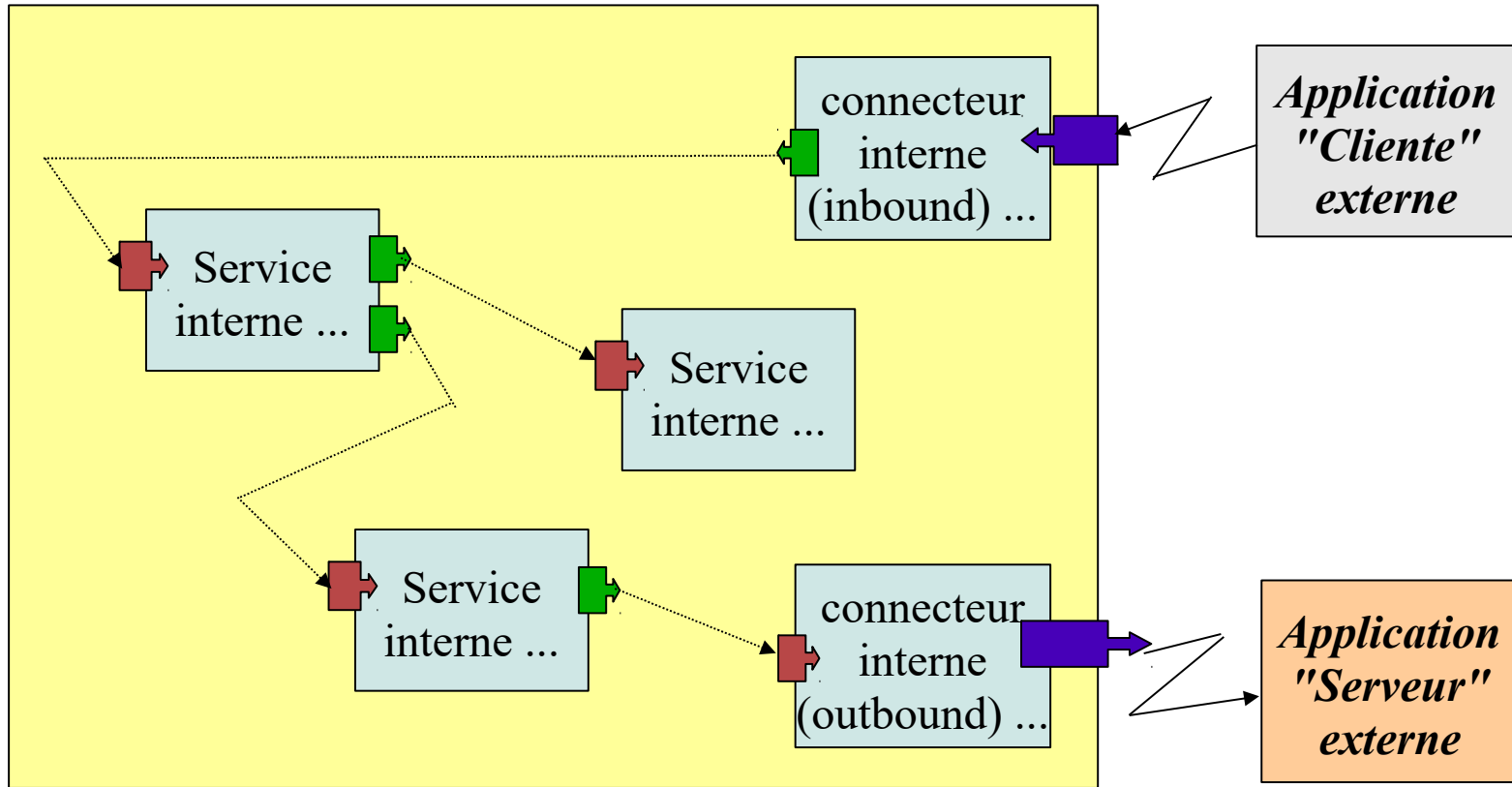
Camel (route)
(ServiceMix)

...

... (...)

"Endpoints" internes à un ESB et connexions explicites

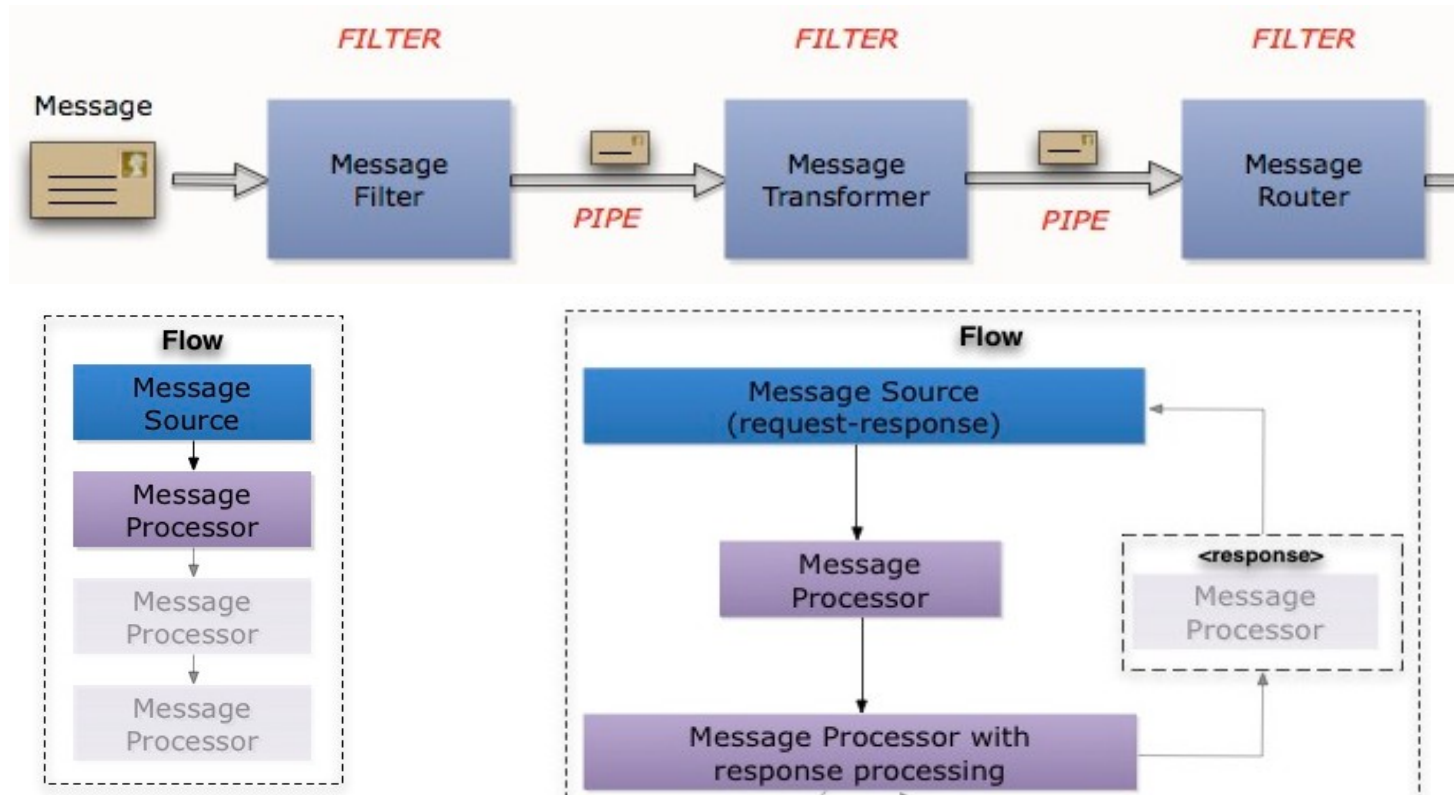
 = *point d'accès (endpoint) interne à l'esb*



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*) !!!!

SCA (Service Component Architecture)

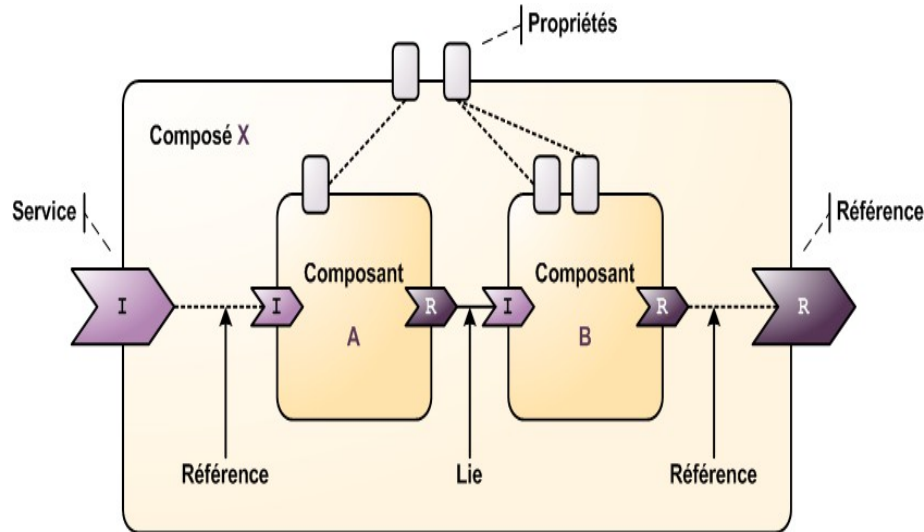
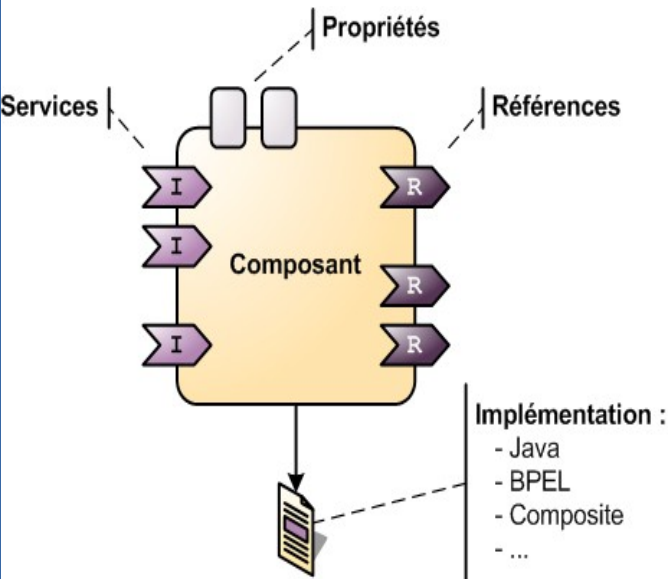
Les spécifications **S.C.A.** (d'origine *IBM*) visent à *structurer des combinaisons/assemblages de services* sur le mode "*consommateurs/fournisseurs*".

S.C.A. se veut être *indépendant des langages de programmation* (java , c++ , c# , bpel ,) .

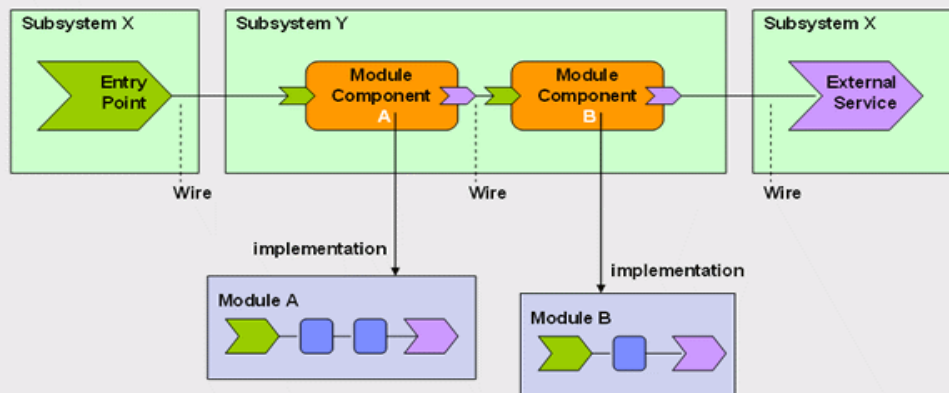
Principales implémentations : *WebSphere ESB* (IBM)
et *Tuscany* (Apache)

Contrairement à JBI , SCA n'impose pas un environnement d'exécution (type de serveur hôte) précis . SCA est plutôt à considérer comme un modèle de structuration d'une unité de services (paramétrages/programmation/...) .

Illustrations sur l'architecture SCA



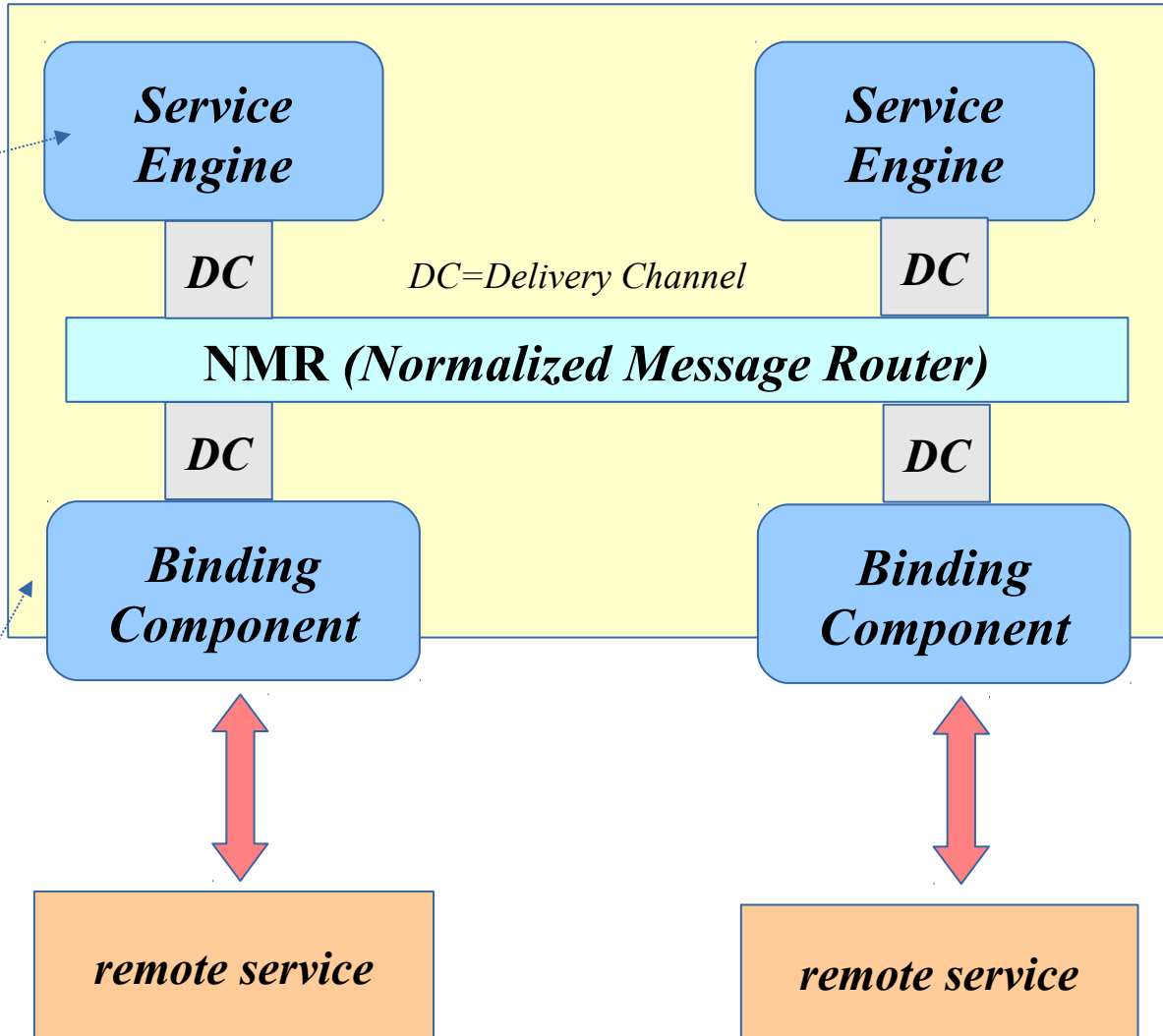
System



Environnement JBI

Has been !!!

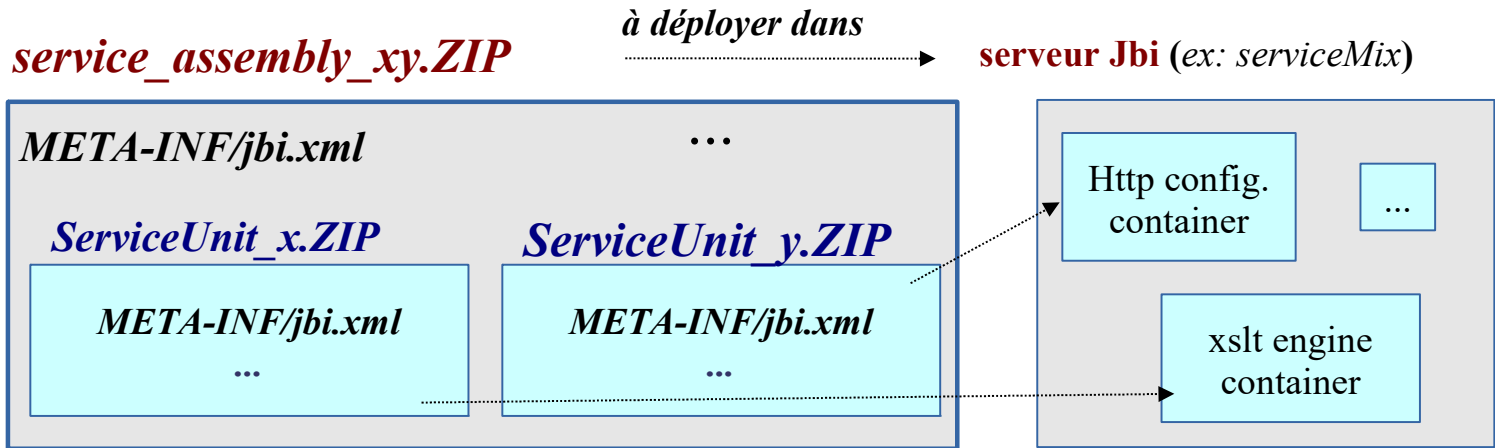
*intégration
des services
internes
/ locaux*



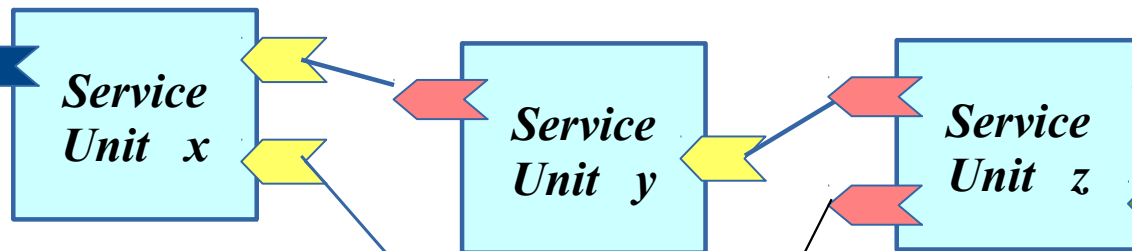
*Gestion de
protocoles
+ proxy
intégrés vers
services
externes*

JBService Assembly (et déploiement jbi)

Has been !!!



assemblage de services (vue logique / fonctionnelle)



Analogie : *appliXy.ear* contenant *webXy.war* et *ejbXy.jar*

Importance toute relative des normes autour des ESB

- La norme **JB**I (Java Business Integration) est aujourd'hui jugée *"has been" / "tombée à l'eau"* car trop complexe.
- La norme "**WCF**" n'est utilisée que chez **Microsoft**.
- La norme "**SCA**" n'est utilisée que chez **IBM**

Donc , au final , ces normes ont une portée très limitée (implémentation/organisation interne à un type d'ESB) et une *assez faible importance (vis à vis des protocoles de communications)* .

Fonctionnalités/spécificités de serviceMix

- **ServiceMix** est un **ESB Open source** codé en **java** (fondation Apache).
- C'est un assez bon et sérieux ESB mais un peu technique et au premier abord un peu moins accessible que son populaire concurrent MuleESB. ServiceMix n'est pas accompagné d'un plugin eclipse pour faciliter le développement ---> *configuration sans assistant !!!*
- ServiceMix est un ESB de type "serveur autonome complet" maintenant basé sur "**osgi**" ce qui permet d'effectuer des déploiements à chaud (sans arrêter le serveur).
- Variantes :
 - * Esb "**ServiceMix**" = version de base (gratuite , sans support).
 - * **FuseESB** → version améliorée avec support / services / documentation enrichie , ... (groupe **redHat**)

Évolution de serviceMix (JBI ou OSGi selon versions)

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

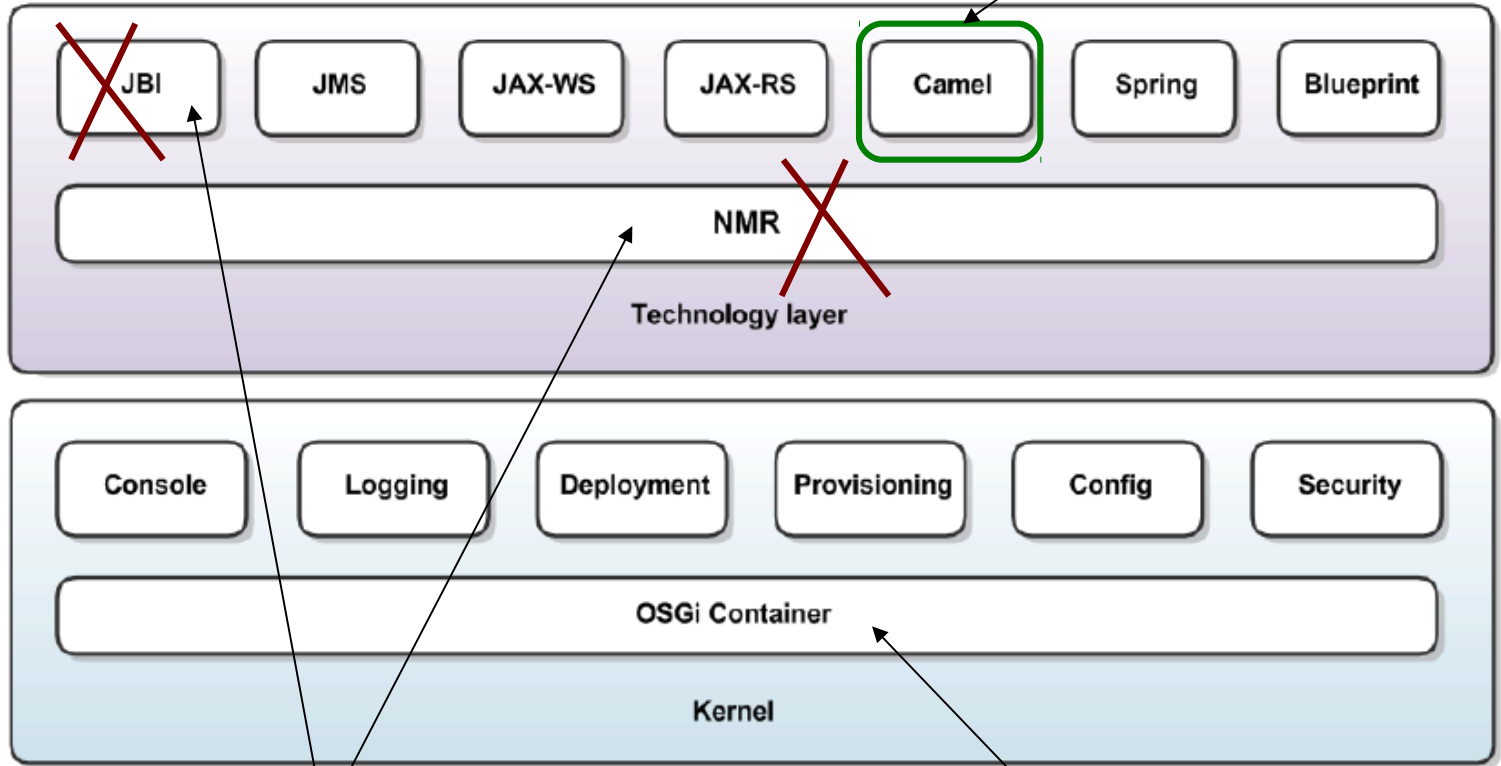
A partir de la **version 4**, la structure de l'ESB **ServiceMix** a été entièrement refondue sur un **cœur OSGi**. ServiceMix ≥ 4 s'appuie en interne sur le container OSGi "**Karaf**" de la marque "Apache"

Les spécifications OSGi en version 4.2 ont 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)

Versions	Technologies	Caractéristiques
3.x	JBI (pas osgi)	Époque où l'on croyait en JBI
4.1 , 4.2 , 4.3	JBI sur Osgi ou Osgi sans JBI (ODE-BPEL intégré via JBI)	Transition "jbi" / "osgi"
4.4	À fond Osgi + cxf + camel	ODE BPEL devenu incompatible

Structure de l'ESB serviceMix

Framework d'intégration fondamentale



*JBI et NMR = anciennes technologies
petit à petit optionnelles et abandonnées .*

solide cœur osgi

Présentation de OSGi



OSGi (*Open Services Gateway initiative*) est une **spécification d'environnement d'exécution java modulaire** .

OSGi est le fruit d'une **alliance** (collaboration) entre les grands éditeurs du monde java (IBM, Oracle , Jboss , ...) et est en soit une spécification indépendante de telle ou telle marque.

OSGi s'appuie sur une **JVM java** et complète celle ci en apportant une très bonne modularité via une structure globale découpée en modules (bundles) relativement indépendants .

A l'origine prévu pour les (petits) **systèmes embarqués** , OSGi est maintenant également utilisé au niveau des (gros) **serveurs d'applications**.

Quelques implémentations d'osgi :

- * **felix**/karaf (apache)
- * **equinox** (eclipse)
- * ...

Quelques produits qui utilisent osgi :

- * Jboss A.S. 7.1 (redHat)
- * **android** (google)
- * ...

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.
-

Modèle structurel en couches :

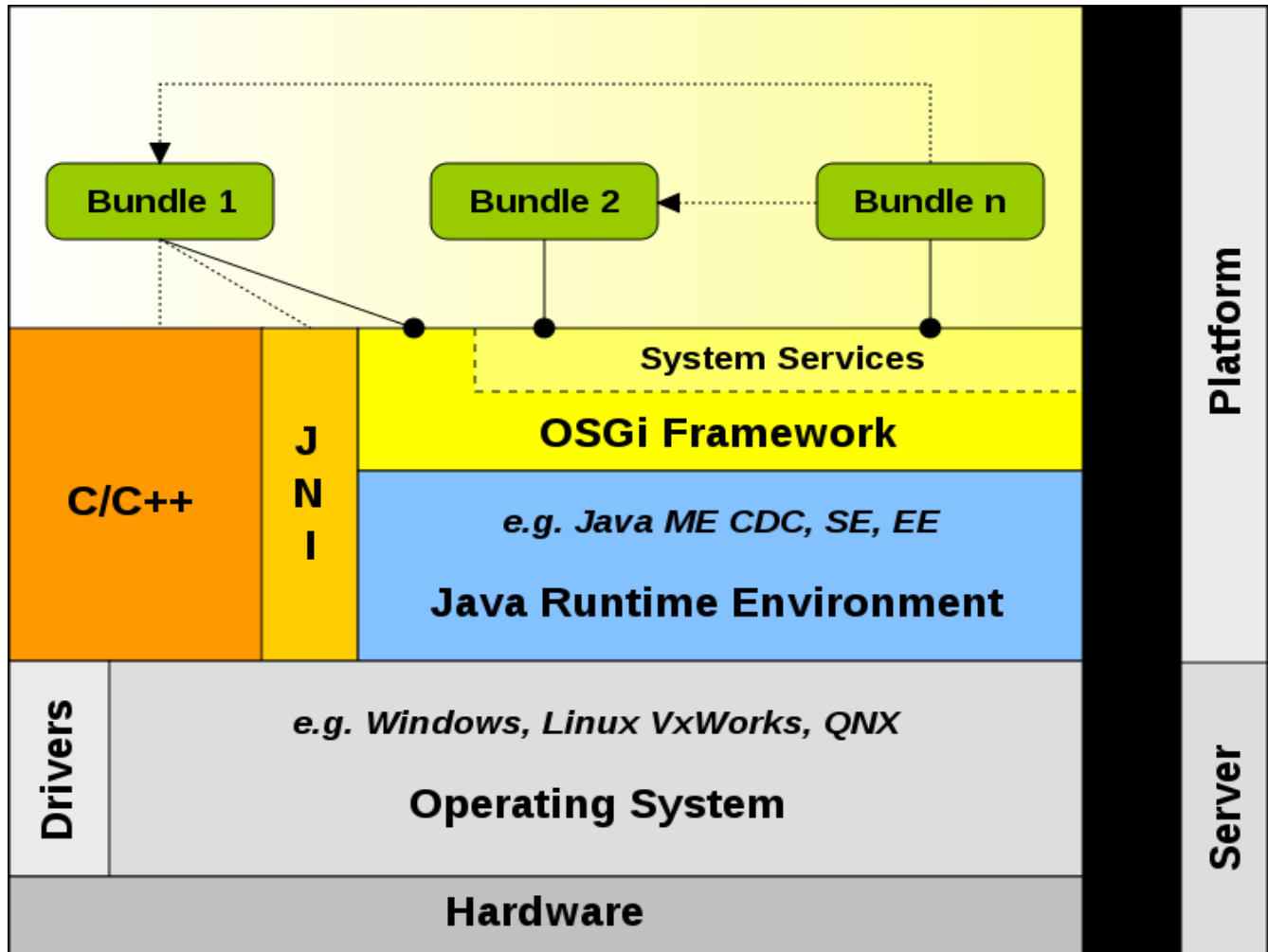
services (publication/souscription)

cycle de vie (activator)

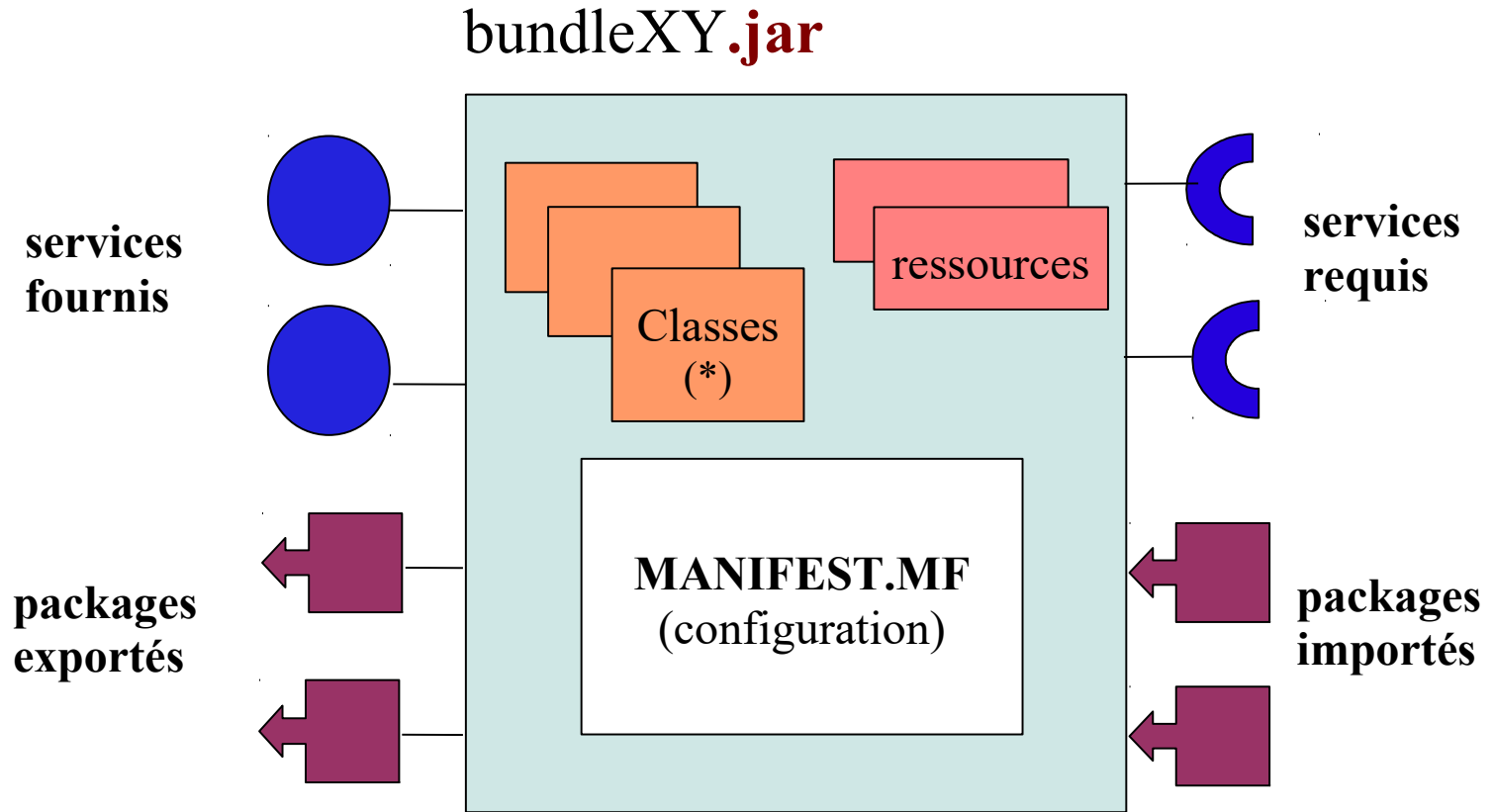
modules/bundles avec "classloader" (avec import/export des packages)

JVM

Architecture osgi (sur-couche pour JVM)



Anatomie d'un bundle OSGi



() classes internes : Activateur + implémentations des services + ...*

Présentation de Camel

Camel est une **technologie d'intégration java** et *open source* de l'éditeur "Apache" .

Camel apporte les principales fonctionnalités suivantes :

- **intégration** des principales **technologies asynchrones** (*JMS* , SMTP , File)
- intégration de quelques technologies synchrones (Http, SOAP via cxf , ...)
- **design patterns d'intégration "EIP"** (filtrage , composition et enrichissement de message, routage conditionné , interception , logs , ...)
- **gestion** simple (syntaxiquement compacte) des **routes** (pour les *messages* véhiculés)
- **configuration "java DSL" et/ou "xml** proche spring" .

En d'autres termes , camel permet de facilement "intégrer" et "piloter" tout un tas de technologies qui tournent autour de la gestion des messages dans un environnement middleware .

Camel est souvent associé à *ActiveMQ* (middleware JMS) ou à l'ESB *ServiceMix* .

Configurations "Camel" en "xml" ou bien "java/DSL"

```
<camelContext xmlns="http://camel.apache.org/schema/blueprint">
  <!-- ou bien xmlns="http://camel.apache.org/schema/spring" -->
  <endpoint id="dossierIn" uri="file:in"></endpoint>
  <endpoint id="dossierOutDefault" uri="file:outDefault"></endpoint>

  <route>
    <from ref="dossierIn" />
    <to ref="dossierOutDefault">
  </route>
</camelContext>
```

xml

```
CamelContext context = new DefaultCamelContext();

context.addRoutes(new RouteBuilder() {
  public void configure() {
    from("test-jms:queue:test.queue")
    .to("file://test");
  }
});
```

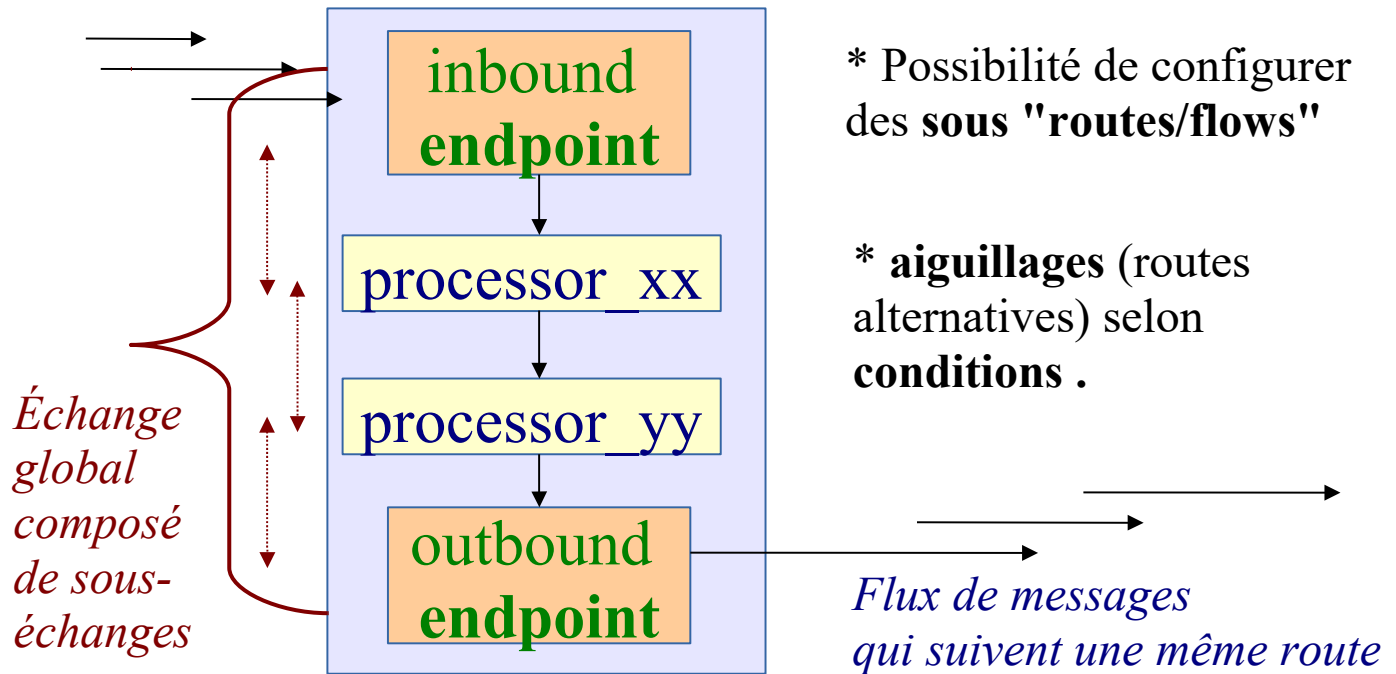
java DSL

Combinaison possible avec d'autres technologies dans même xml

Variantes et héritages de config. possibles

route (alias "message **flow**")

- MuleESB utilise le vocabulaire "message **flow**" (flux de message).
- Camel (intégré à servicemix) se configure à base de "**route**" pour les message qui circulent.



Exchange / MEP

Entre deux éléments d'une route (endpoint , processeur , ...) , des messages sont **échangés** (**produits/fournis** ou **recupérés/consommés**) selon une certaine logique dépendant essentiellement du mode de communication (**synchrone** ou **asynchrone**).

MEP signifie "**M**essage **E**xchange **P**attern" et correspond à un type récurrent d'échange de messages entre deux éléments.

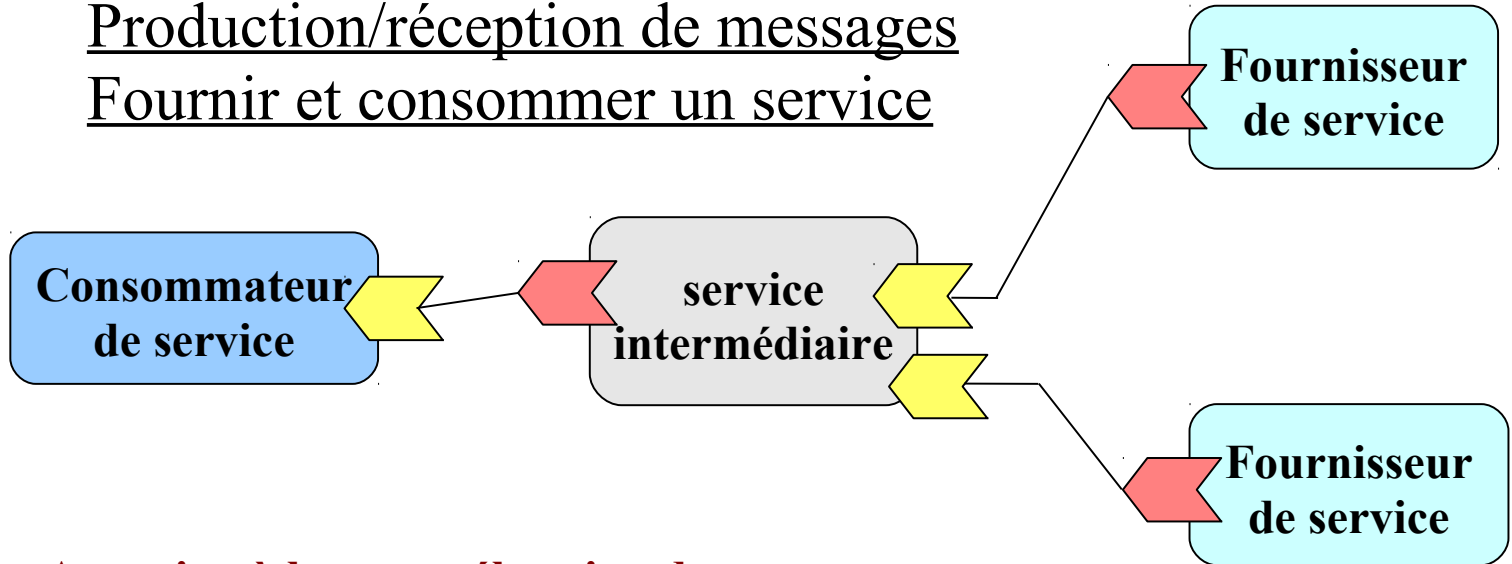
Les 2 principaux "MEP" sont "**InOnly**" (one-way) et "**InOut**".

Un grand échange global est souvent décomposé en de multiples sous-échanges .

Une transaction sur un échange global aura comme portée "tous les sous-échanges impliqués". Une erreur de traitement à un niveau pourra ainsi déclencher un "rollback global" et annuler la consommation / prise en compte d'un message à l'entrée de la route (ou flux) à problème.

==> cette **logique transactionnelle** permet d'obtenir des **échanges fiables** !

Production/réception de messages Fournir et consommer un service



Attention à la compréhension des sens :

"Client" = **Consommateur de service** (= souvent **producteur de message de requête** et éventuellement *receveur du message de réponse*) .

"Serveur" = **Fournisseur de service** (= souvent **receveur du message de requête** et éventuellement *producteur du message de réponse*) .

Quelques solutions "ESB open source"

ESB **ServiceMix**
(serveur OSGi)

camel-xy

cxp (ws)

Développement (*sans assistant*) de configuration
avec
spring/blueprint
+ camel
sous **eclipse** standard
et avec "*maven*"

Mule-ESB
(serveur standalone)

cxp (ws)

mule-xy

Développement (**avec assistant graphique**) de
configuration avec
spring/mule
sous **Mule/AnyPointStudio**
et avec ou sans "*maven*"

Mode "embarqué" (léger)

"Appli java" ou
"serv JEE / tomcat"

camel-xy

cxfr (ws)

Développement
qui se base
uniquement sur "**camel**"
servicemix

"Appli java" ou
"serv JEE / tomcat"

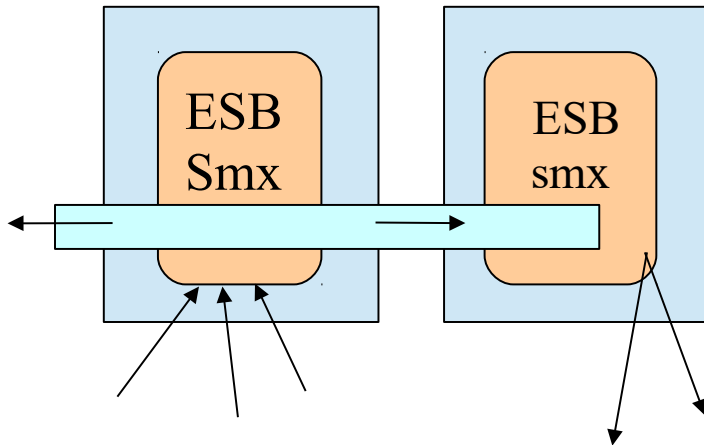
cxfr (ws)

mule-xy

Développement (avec
assistant graphique) de
configuration avec
spring/mule
sous **Mule/AnyPointStudio**
et avec ou sans "*maven*"

Mode "tenue en charge" (cluster solide)

*Via canal/bus "JMS"
inter-machine*



smx = servicemix

