

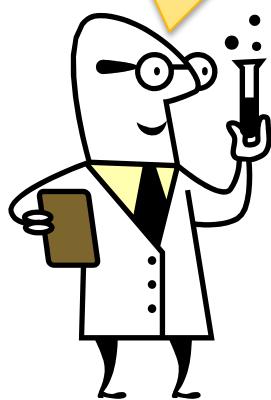
# Services Web - SOAP

Stéphanie CHOLLET

Yoann MAUREL

# Les services Web, plusieurs visions...

hétérogénéité,  
dynamisme, découverte,  
qualité de service,  
changer le monde, ...



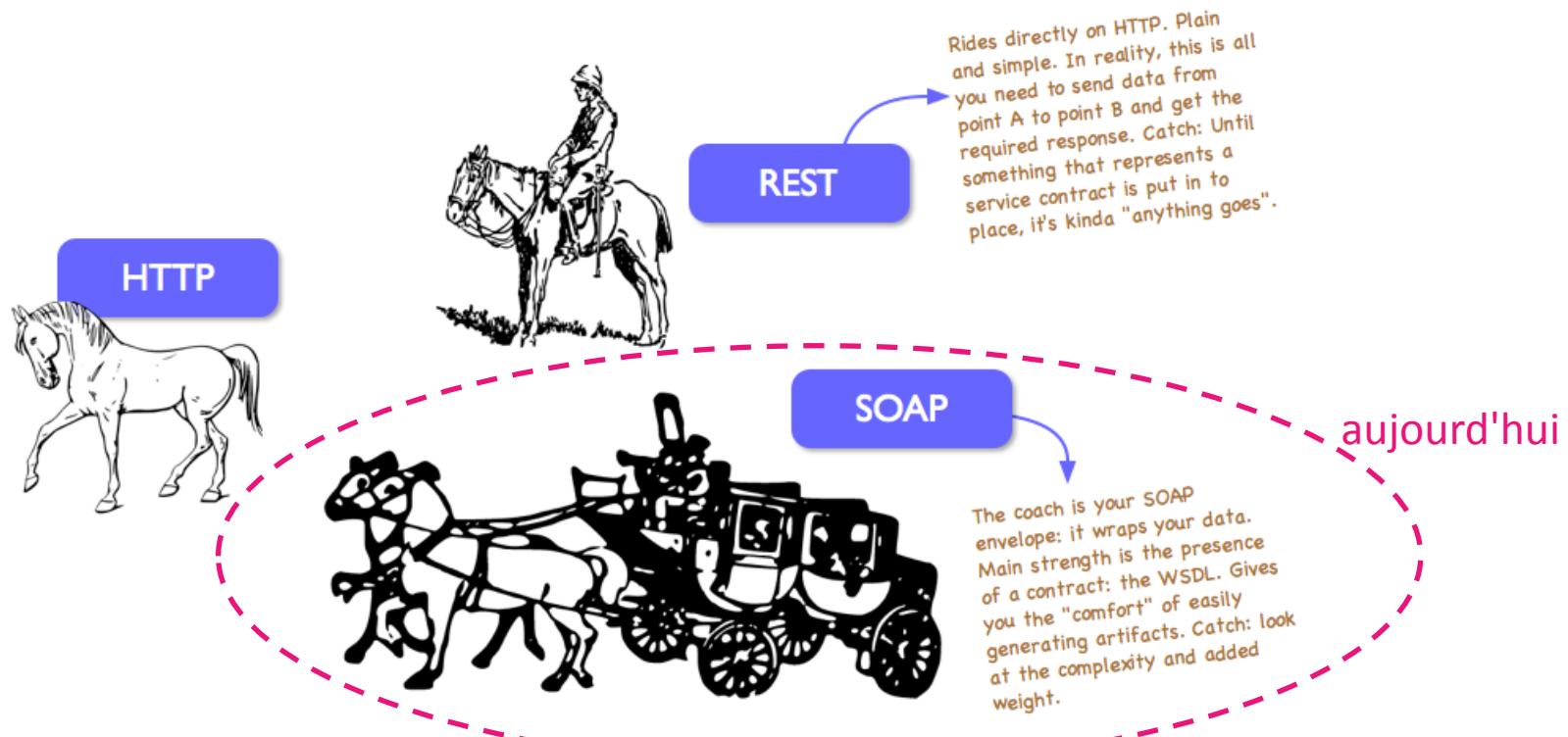
hétérogénéité



vision caricaturale, évidemment, ...

# Les services, plusieurs façons ...

- ▶ **Les services SOAP** qui ressemblent à RPC
- ▶ les services REST (plus récents) qui permettent d'échanger des ressources

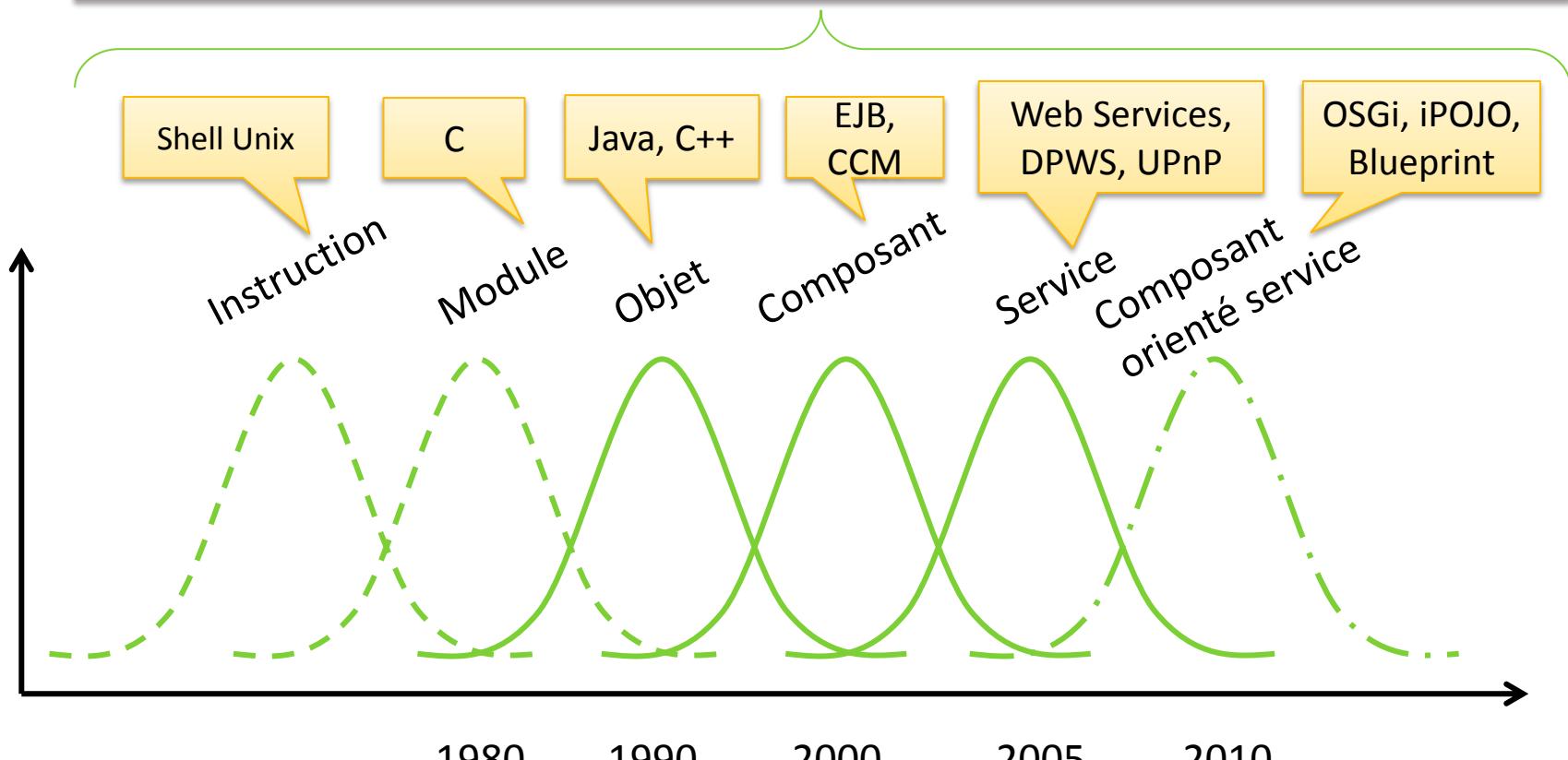


source ?

# Rappel : Evolution des technologies

- ▶ Vision étendue de celle de [Racoon 1997]

**Bonnes pratiques : *Design Patterns*, Architecture Logicielle, Génie Logiciel**



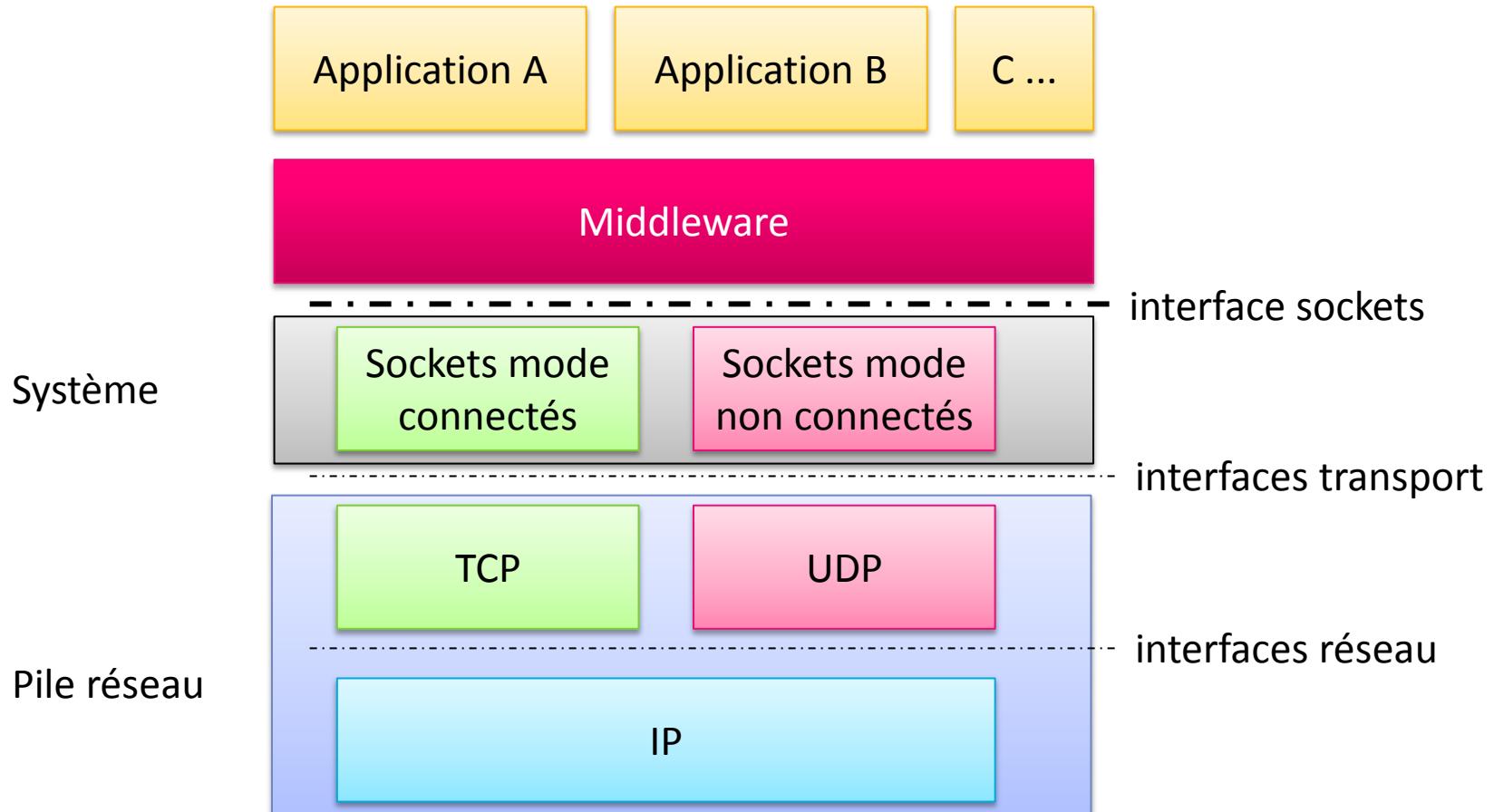


# Rappels

sur les technos de communications

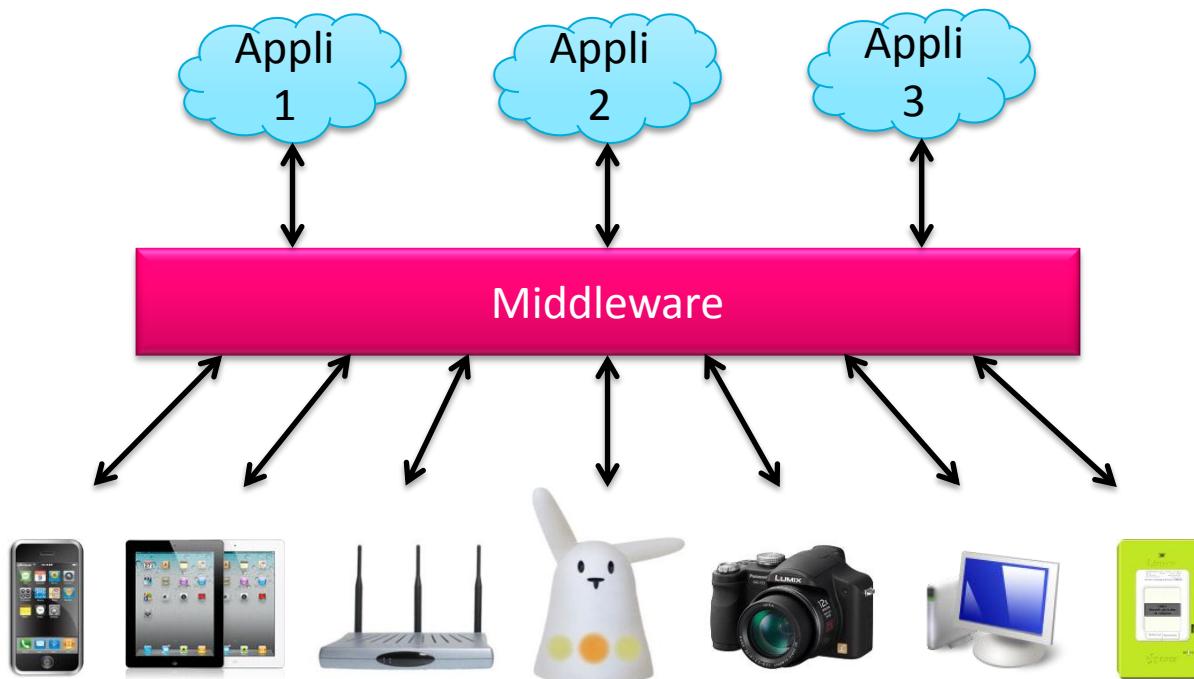
# Notion de Middleware

- ▶ On ajoute une couche entre le système et les applications



# Contexte

Un **intergiciel** (*middleware*) est un programme qui s'insère entre deux applications et qui va permettre de faire communiquer des machines entre elles, indépendamment de la nature du **processeur**, du **système d'exploitation**, voire du **langage**.



# Mode Synchrone

**Mode Synchrone** : lorsqu'une application cliente est **bloquée en attente** d'une réponse à une requête au serveur



- ▶ Un appel synchrone nécessite la présence simultanée du client et du serveur

# Mode Asynchrone

**Mode Asynchrone** : lorsque l'appel n'est pas bloquant, l'application continue son déroulement **sans attendre de réponse**.



- ▶ Un appel asynchrone peut décorreler dans le temps la requête et sa prise en charge

# Intérêts des modes synchrone et asynchrone

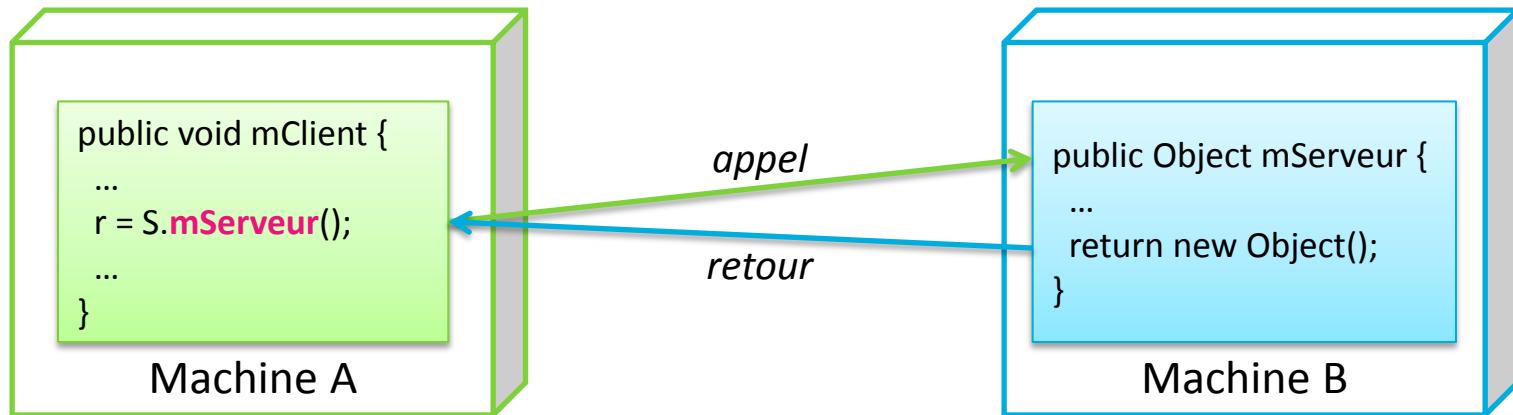
---

- ▶ Mode synchrone :
  - ▶ Plus de fiabilité :
    - ▶ Assure que le message a été envoyé et qu'on a bien reçu une réponse
  
- ▶ Mode asynchrone :
  - ▶ Plus de performance :
    - ▶ Permet de libérer du temps de calcul pour d'autres actions afin de paralléliser différentes tâches

# Communication entre applications

## Type RPC (*Remote Procedure Call*)

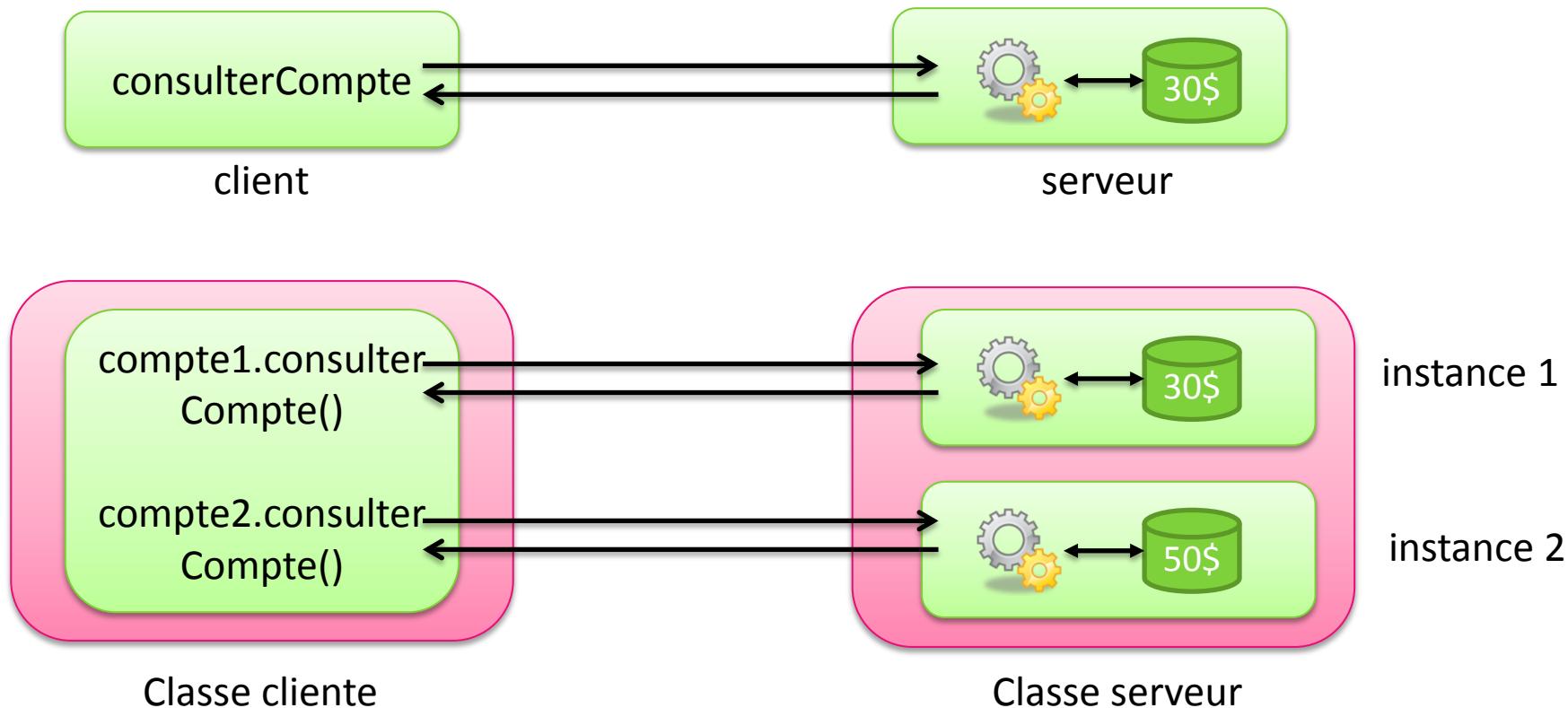
Un **appel de procédure à distance** est l'invocation d'un service (i.e. d'une procédure ou d'une méthode d'un objet) situé sur une machine distante indépendamment de sa localisation ou de son implantation.



- ▶ Le concept est le même que l'appel de sous-procédure, avec un **déroutement du contexte d'appel** et une **attente bloquante** des résultats.

# Les appels de méthodes sur des objets distants

- ▶ Mis en œuvre par CORBA et RMI par exemple
- ▶ Extension du concept aux objets



# Les appels de méthodes sur des objets distants

---

- ▶ Selon la technologie choisie, on manipule :
  - ▶ des **objets du langage** :
    - ▶ instances de classe Java par exemple
    - ▶ cas de **RMI**
    - ▶ Avantage : facilité
    - ▶ Inconvénient : mono-langage
  - ▶ des objets "universels" :
    - ▶ définis par le middleware utilisé, souvent à vocation multi-langages
    - ▶ cas de **CORBA**
    - ▶ Notion de **langage pivot** entre les différents langages
      - Description de l'interface des objets distants (**IDL : interface définition language**)
    - ▶ Avantage : multi-langage
    - ▶ Inconvénient : plus difficile à mettre en œuvre

# Emballage/Déballage (Marshalling/Unmarshalling)

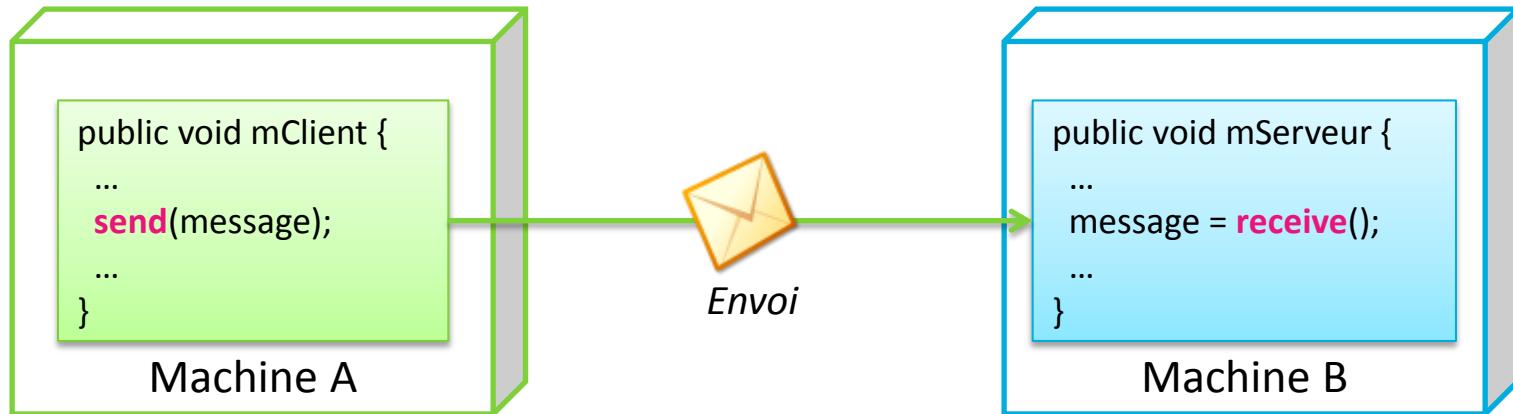
---

- ▶ On parle aussi de sérialisation/désérialisation, linéarisation
- ▶ Encoder l'état mémoire d'un objet pour permettre :
  - ▶ son stockage (persistance)
  - ▶ **sa transmission**
- ▶ Ensemble des règles :
  - ▶ représentation internes des données
  - ▶ structure de l'objet
  - ▶ format binaire/textuel (ex. XML)
- ▶ Java propose des APIs pour la sérialisation/désérialisation
  - ▶ **notamment pour XML (on verra JAXB)**

# Communication entre applications

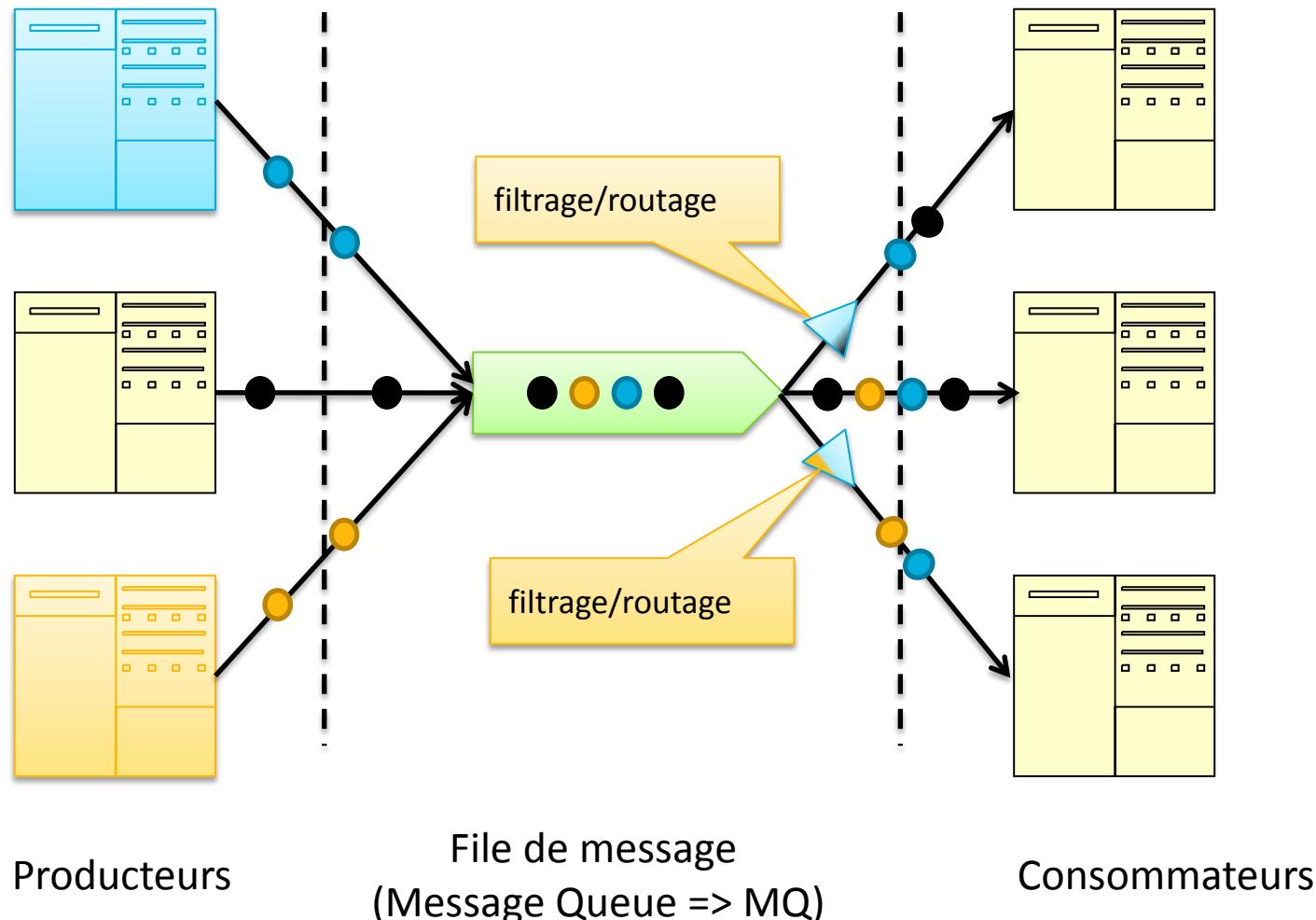
## Type MOM (*Message Oriented Middleware*)

Les applications voulant communiquer entre elles s'échangent  
**des messages** véhiculés par l'infrastructure MOM.



- ▶ Les messages ont une nature complètement générique, ils peuvent représenter tous type de contenu aussi bien du binaire (images, objets serialisés) que du texte (document XML)

# Routage des messages dans les MOMs



# Classification des technologies

---

Type RPC	Type MOM
CORBA ( <i>Common Object Request Broker Architecture</i> )	JMS ( <i>Java Messaging Service</i> )
RMI ( <i>Remote Method Invocation</i> )	AMQP
DCOM ( <i>Microsoft DNA</i> )	
.NET Remoting ( <i>Microsoft .NET</i> )	
SOAP ( <i>Simple Object Access Protocol</i> )	

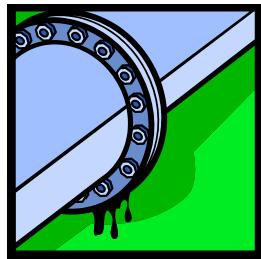


## Idée derrière la notion de service

# Idée générale appliquée au monde réel

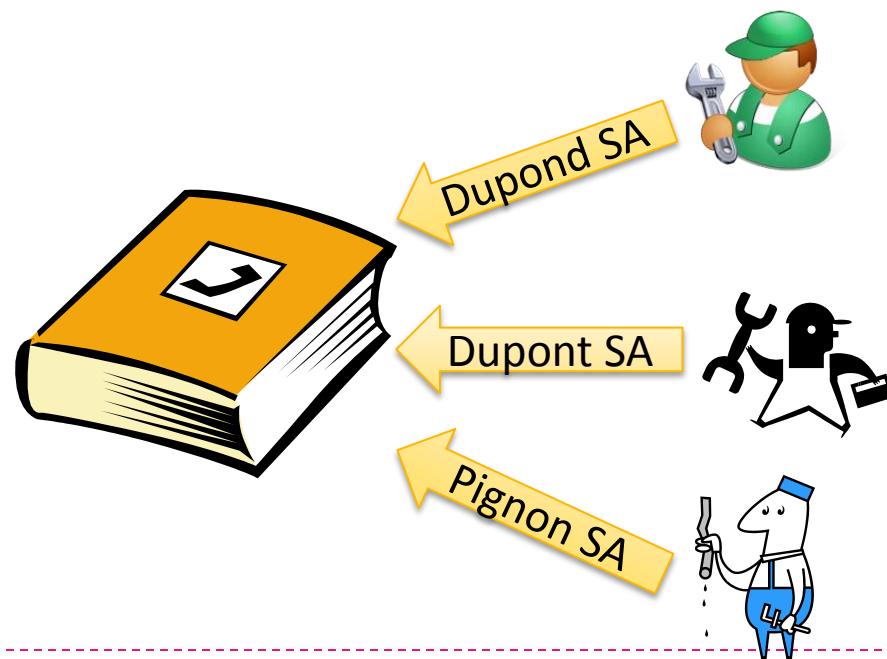
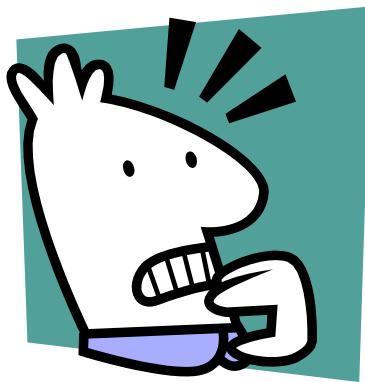
---

- ▶ Bob a une fuite d'eau chez lui



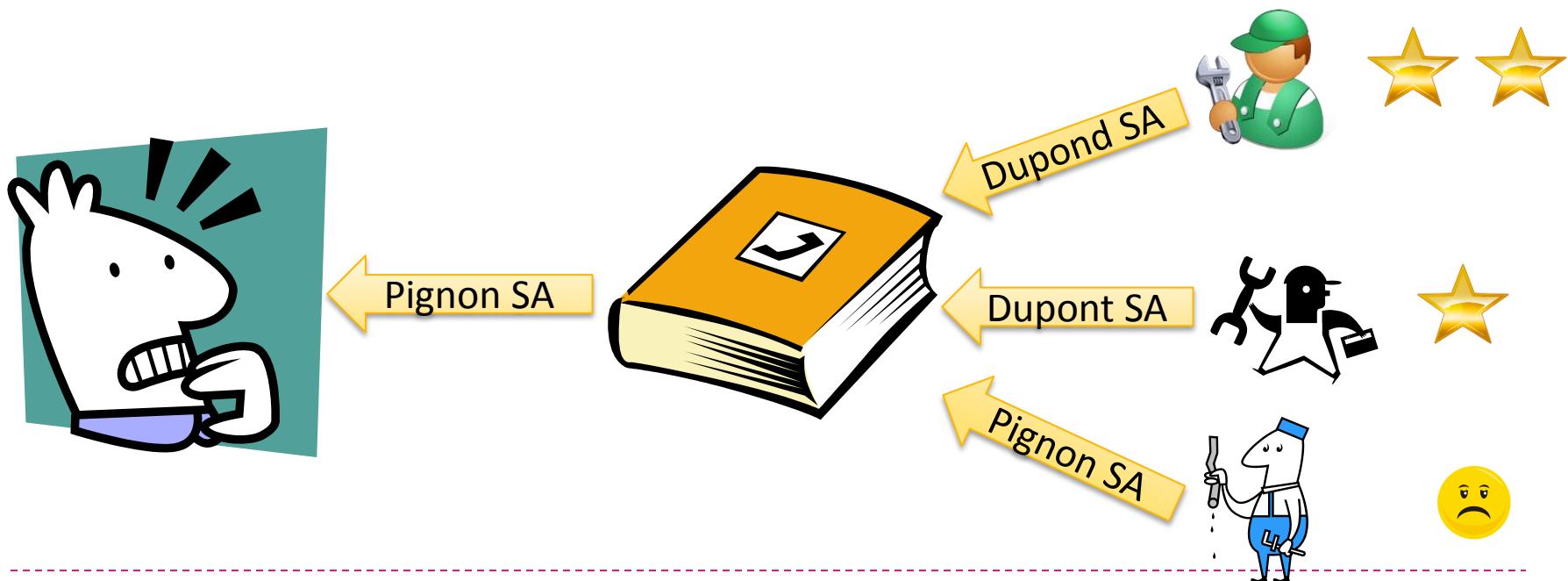
# Idée générale appliquée au monde réel

- ▶ Il cherche dans l'annuaire un ***service de plomberie***
  - ▶ tout le monde est d'accord sur la notion de service de plomberie
    - ▶ d'ailleurs de nombreux fournisseurs de ce service figure déjà dans l'annuaire



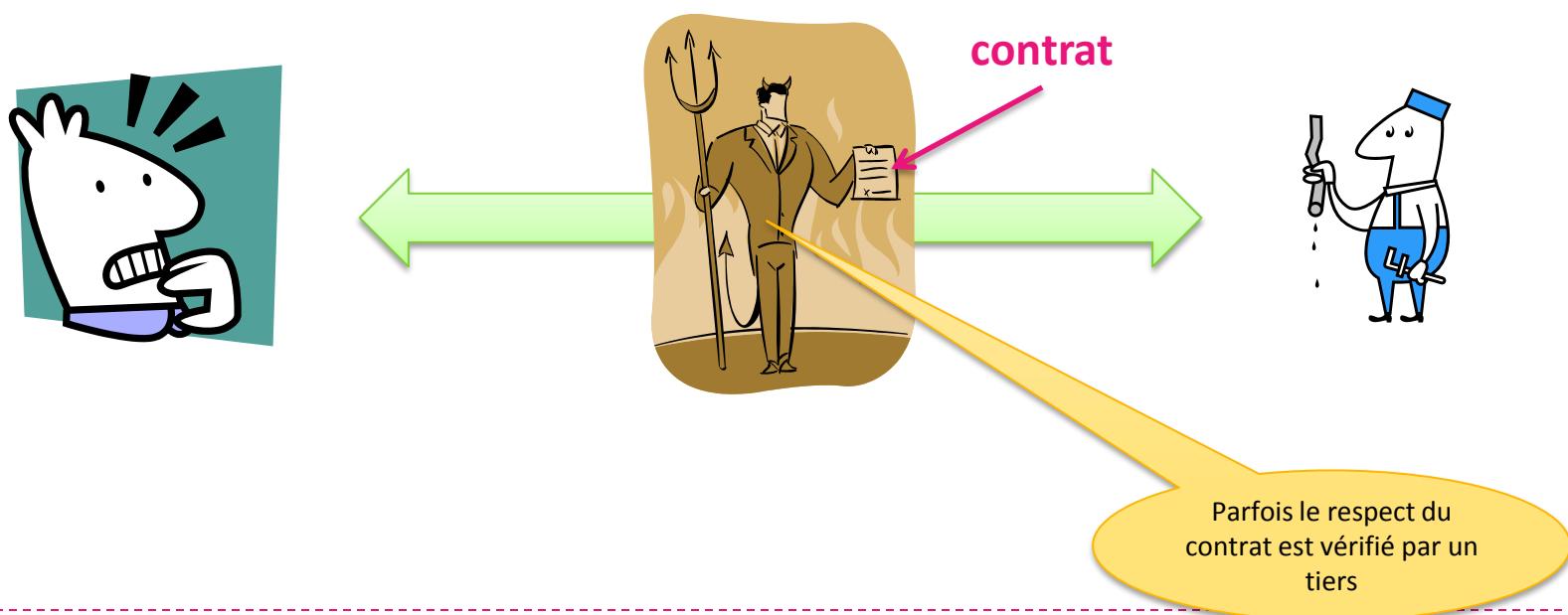
# Idée générale appliquée au monde réel

- ▶ La **qualité et le coût de service** ne sont pas forcément les mêmes
- ▶ Bob choisit parmi les fournisseurs disponibles, celui qui fournit la prestation qui correspond à ses besoins (pas forcément le meilleur ...)



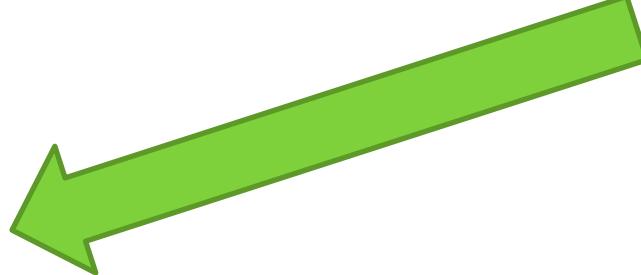
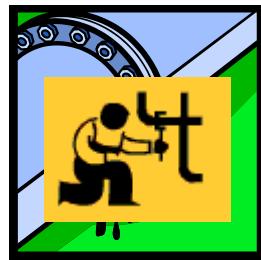
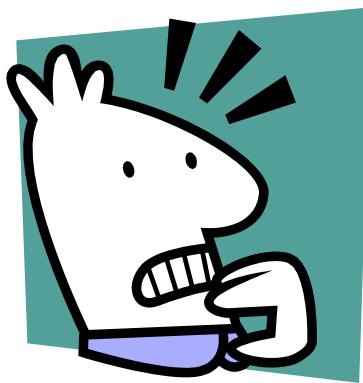
# Idée générale appliquée au monde réel

- ▶ Bob et *Pignon SA* se mettent d'accord sur un **contrat de service** qui décrit la qualité du service et les coûts de la prestation



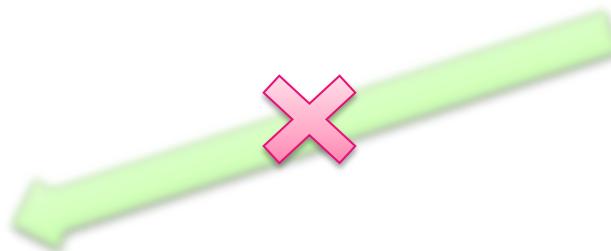
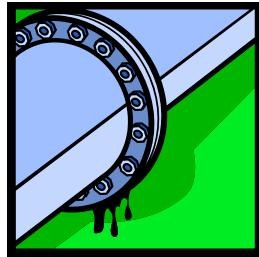
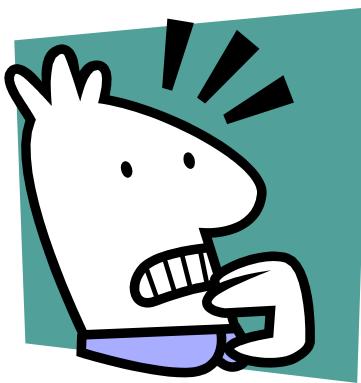
# Idée générale appliquée au monde réel

- ▶ Bob peut enfin utiliser les services de Monsieur Pignon



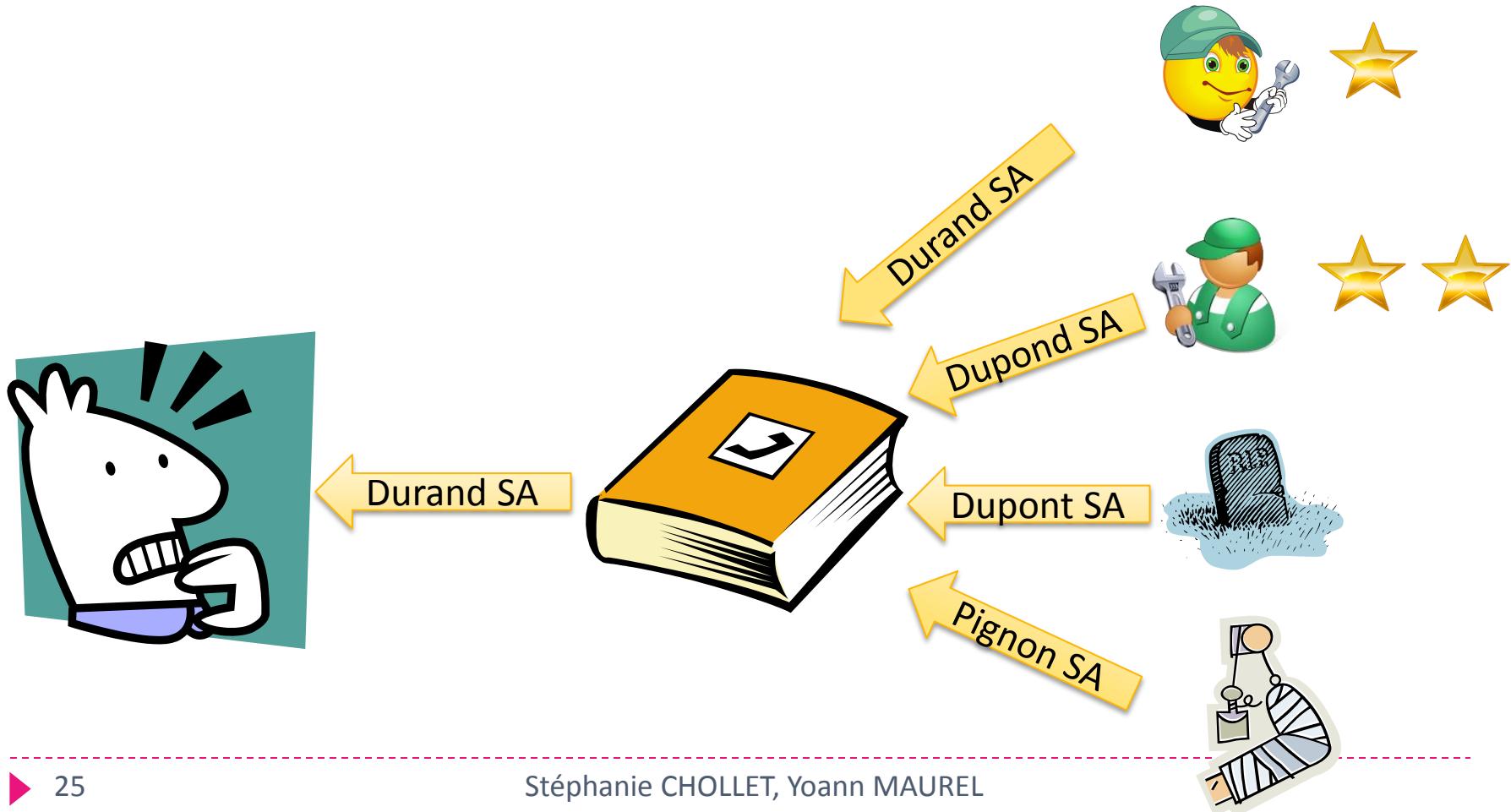
# Idée générale appliquée au monde réel

- ▶ Bob a **toujours** une fuite d'eau chez lui
- ▶ Monsieur Pignon a eu un accident de travail



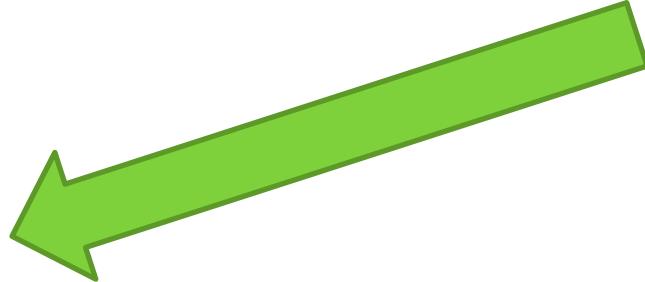
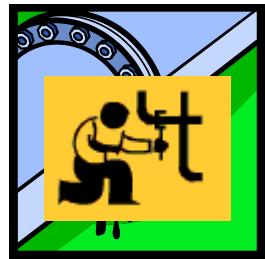
# Idée générale appliquée au monde réel

- ▶ Il recherche un remplaçant
  - ▶ Depuis la dernière recherche, les fournisseurs ont changés



# Idée générale appliquée au monde réel

- ▶ Bob peut enfin faire réparer sa fuite



# Pourquoi est-ce possible ?

Pour les services Web (SOAP)

- ▶ Le **service** est **décrit** de façon standard que tous les clients et fournisseurs connaissent
  - ▶ ici en termes métiers par exemple
- ▶ L'**annuaire** fournit des **points d'accès standard** vers les fournisseurs
  - ▶ le numéro de téléphone
- ▶ Le **contrat de service** est décrit de façon standard
  - ▶ en termes juridiques par exemple
- ▶ Bob et les fournisseurs communiquent de façon standard

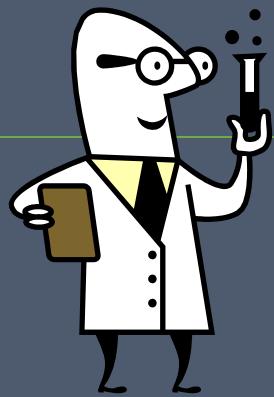
WSDL (*Web Services Description Language*)

Annuaire = UDDI (*Universal Description Discovery and Integration*)

Notion de endpoint

Le SLA  
(*Service Level Agreement*)

SOAP  
(*Simple Object Access Protocol*)

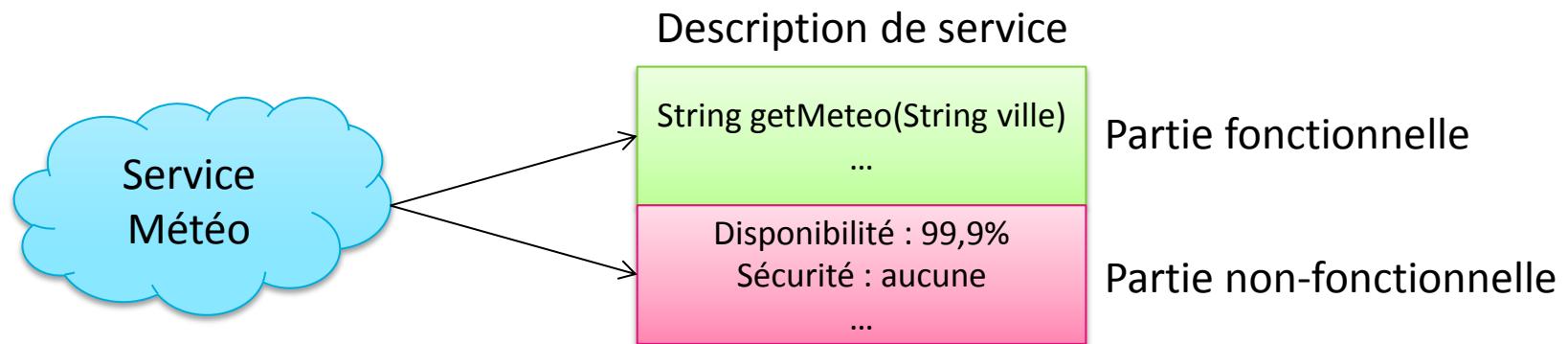


La théorie ...

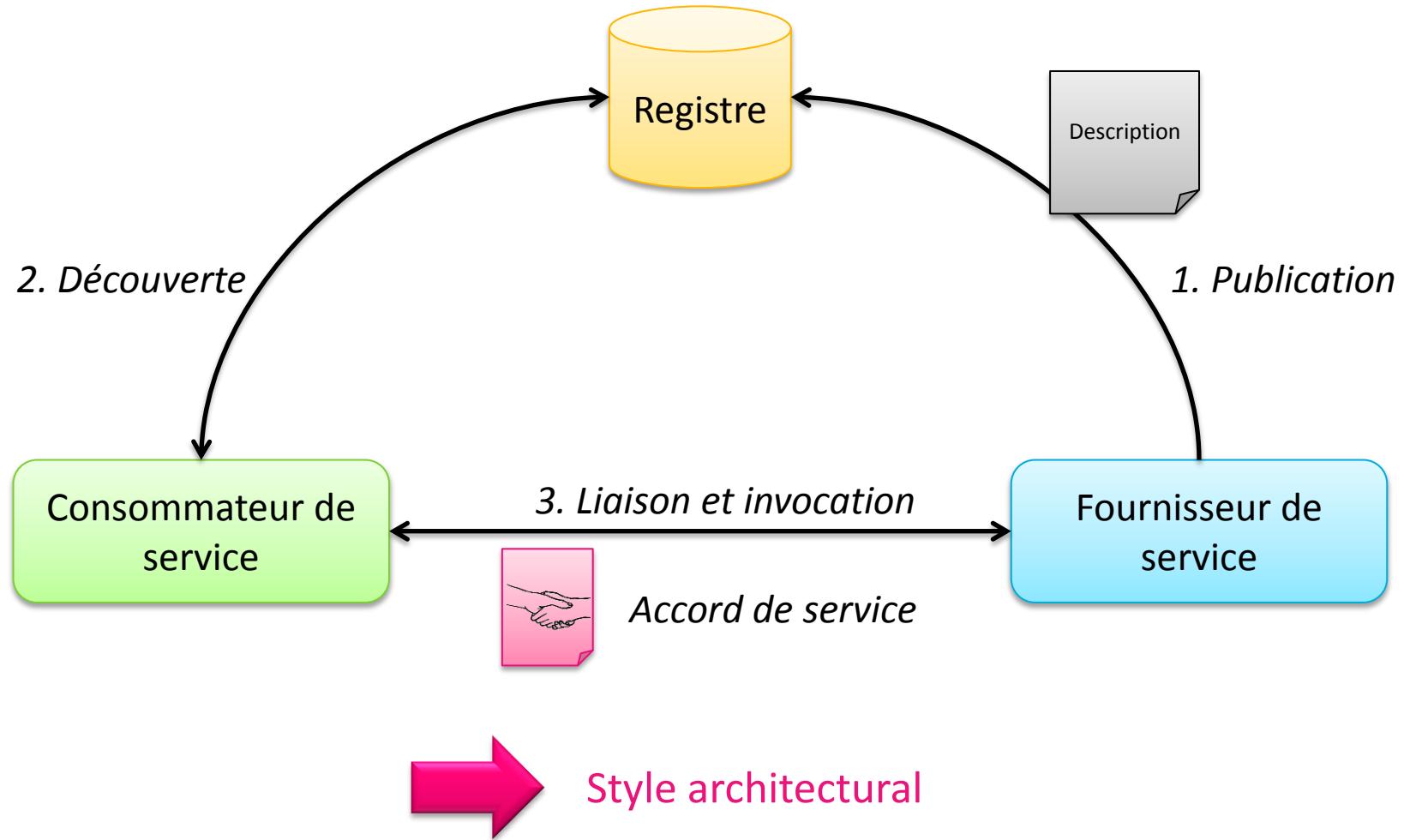
# Service

Un service est une **entité logicielle** qui fournit un ensemble de fonctionnalités définies dans une **description de service**.

- ▶ La description de service contient des informations :
  - ▶ Sur la partie fonctionnelle du service
  - ▶ Sur les aspects non-fonctionnels du service



# Architecture de l'approche à services



# Les acteurs et les interactions

---

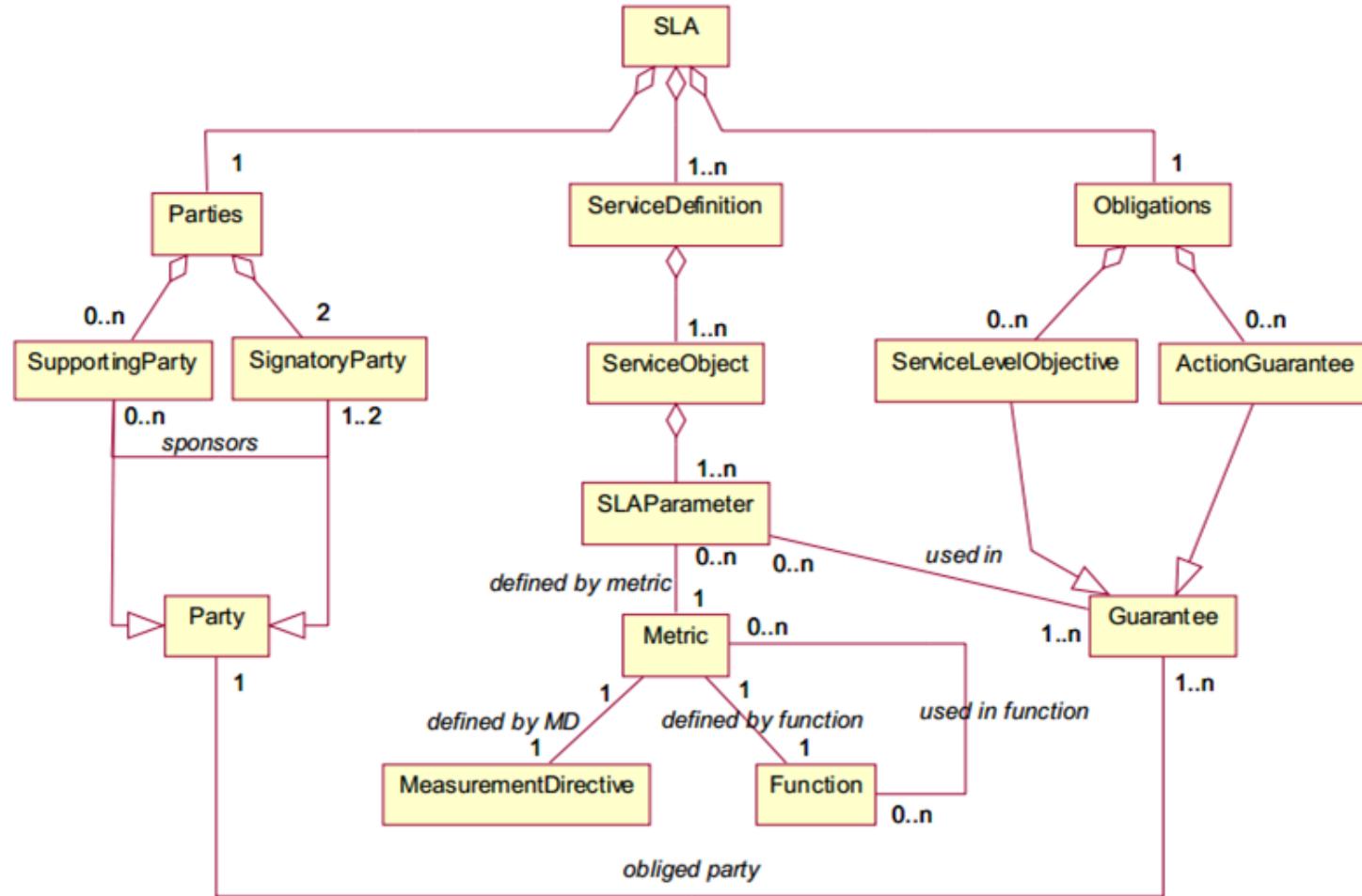
- ▶ **Le fournisseur de service :**
  - ▶ Propose un service décrit dans une spécification
- ▶ **Le registre (ou annuaire) de services :**
  - ▶ Stocke l'ensemble des définitions de services déclarées par les fournisseurs de service
  - ▶ Permet aux consommateurs de rechercher et de sélectionner le service qui leur sera utile
- ▶ **Le consommateur (ou client) de service :**
  - ▶ Utilise des services des fournisseurs

# *Service Level Agreement (SLA)*

Contrat souscrit entre le fournisseur d'un service et un usager de ce service définissant les engagements de ces deux parties. Ces engagements, contenant le **niveau de service** fourni ainsi que **les pénalités encourues** en cas de manquement de part et d'autre, sont définis par des **critères objectifs** de qualité de service pouvant être évalués par les deux parties. [TOUSEAU 2010]

- ▶ Peu de standard ou de formalisme générique
  - ▶ Souvent papier, langue naturelle, solutions ad hoc
- ▶ Représentations existantes
  - ▶ Web Service Level Agreement (WSLA)
  - ▶ Rule-Based Service Level Agreements (RBSLA)
  - ▶ ContractLog [Paschke2003]
  - ▶ WS-Agreement

# Exemple : WSLA



<http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf>

# Intérêt de l'Architecture Orientée Services

---

## ▶ Propriétés :

- ▶ **Faible couplage**
  - ▶ Seule la description est partagée entre les différents acteurs
- ▶ **Masquage de la localisation et de l'hétérogénéité**
  - ▶ De l'implantation et des plates-formes
- ▶ **Substituabilité**
  - ▶ Remplacement de service de façon transparente
- ▶ **Liaison retardée**
  - ▶ Liaison faite uniquement lorsque le consommateur a besoin d'un service

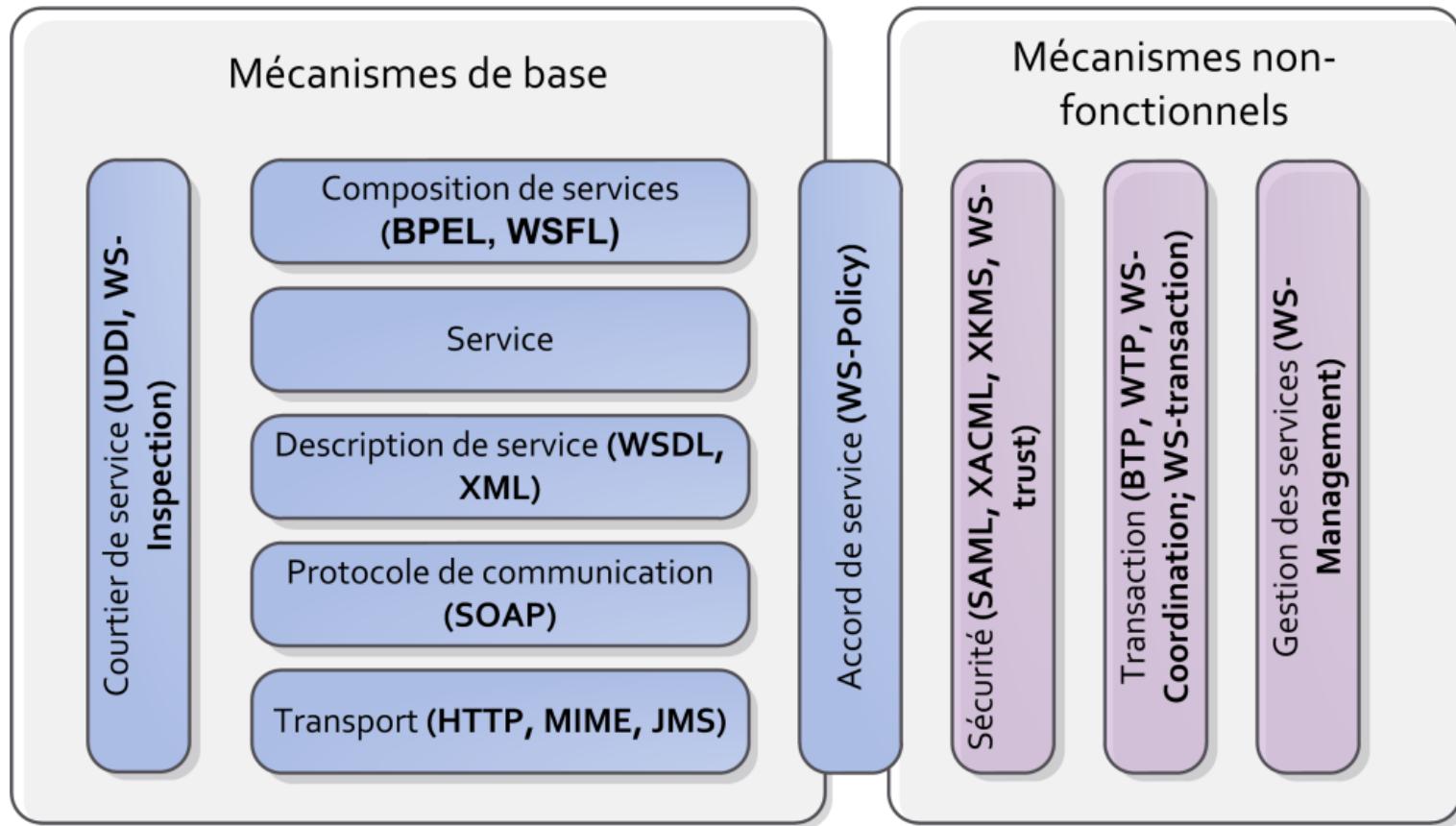
# L'architecture à services

## ► En anglais : *Service-Oriented Architecture*

Ensemble des mécanismes permettant la création d'applications suivant les principes de l'approche orientée services.

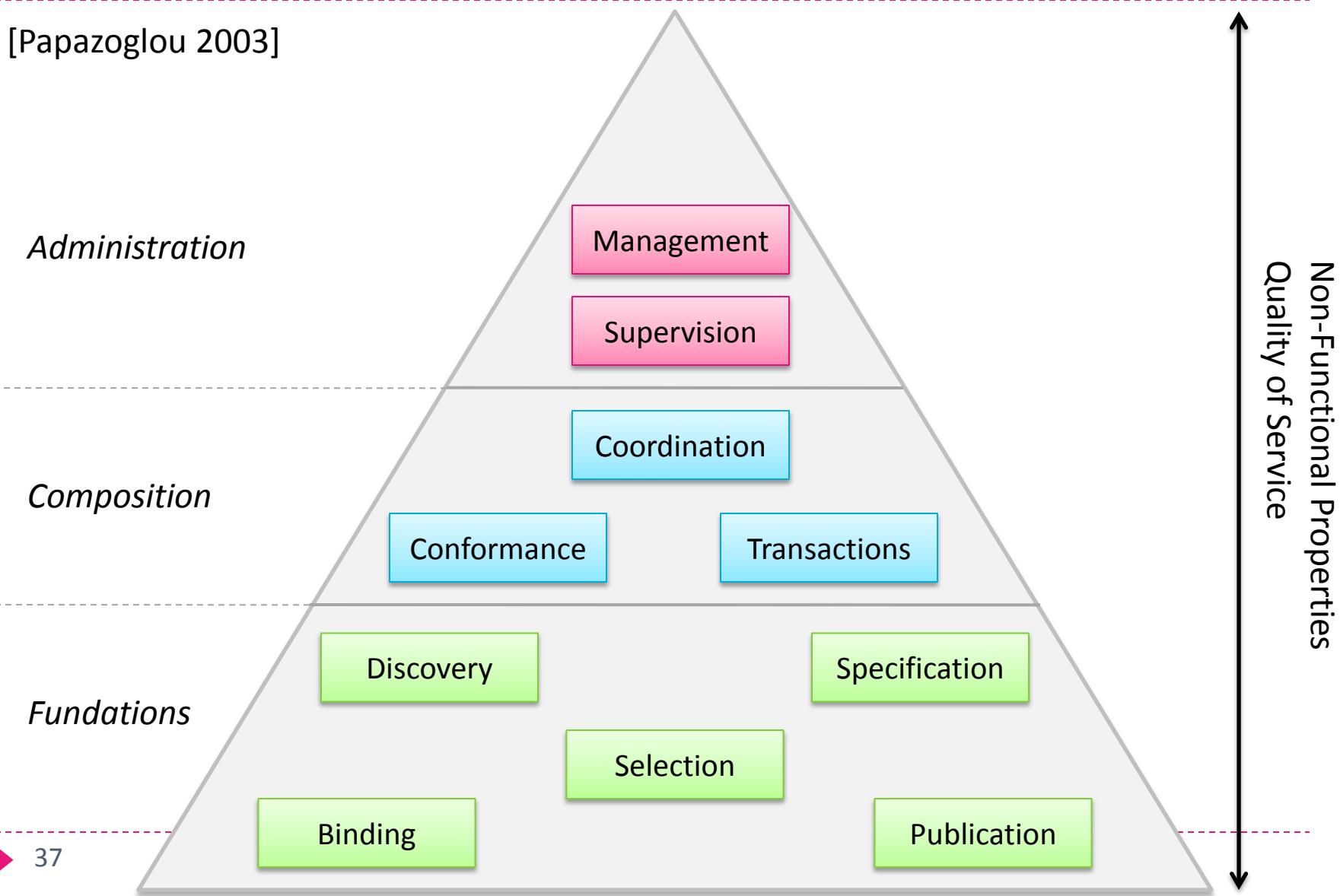
- Mécanismes de base assurant :
  - ▶ La publication, la découverte, la composition, la négociation, la contractualisation et l'invocation des services
- Mécanismes additionnels :
  - ▶ Pour les besoins non-fonctionnels

# L'architecture à services

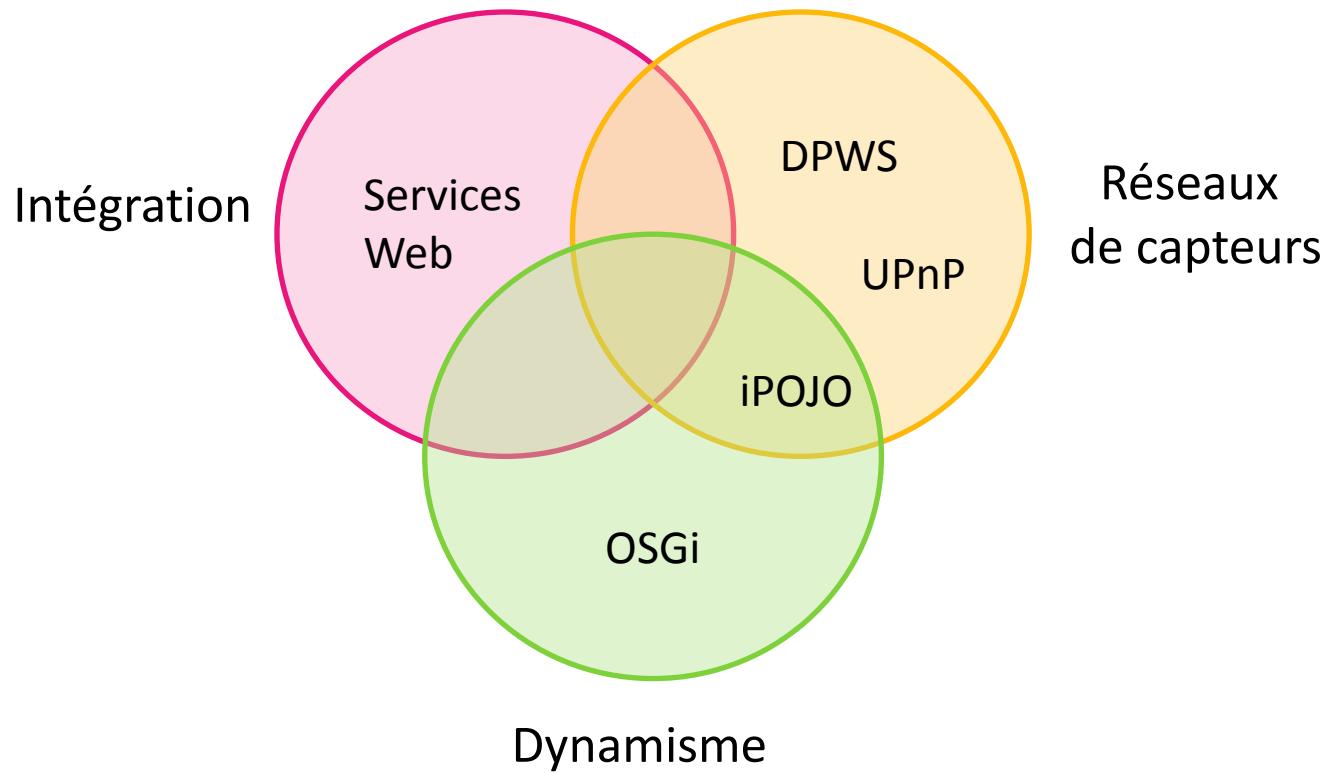


# Besoins d'une application à services

[Papazoglou 2003]



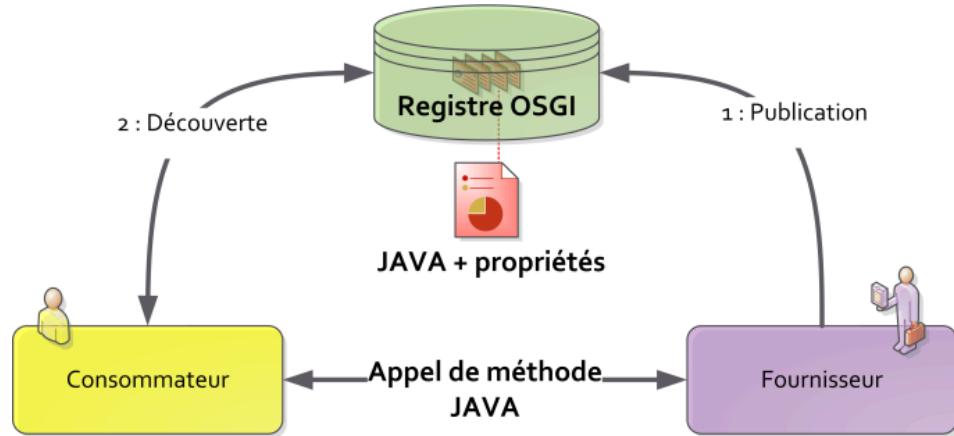
# Les technologies à services ....



# Donc ....

Un service Web est une forme de service mais **tous les services ne sont pas des services Web**

- ▶ Autre exemple (OSGi) :
  - ▶ Description du service :
    - ▶ interface JAVA + propriétés (clef, valeur)
  - ▶ Annuaire
    - ▶ fourni par la plateforme OSGi (plateforme en Java)
  - ▶ Communication
    - ▶ appel direct en JAVA



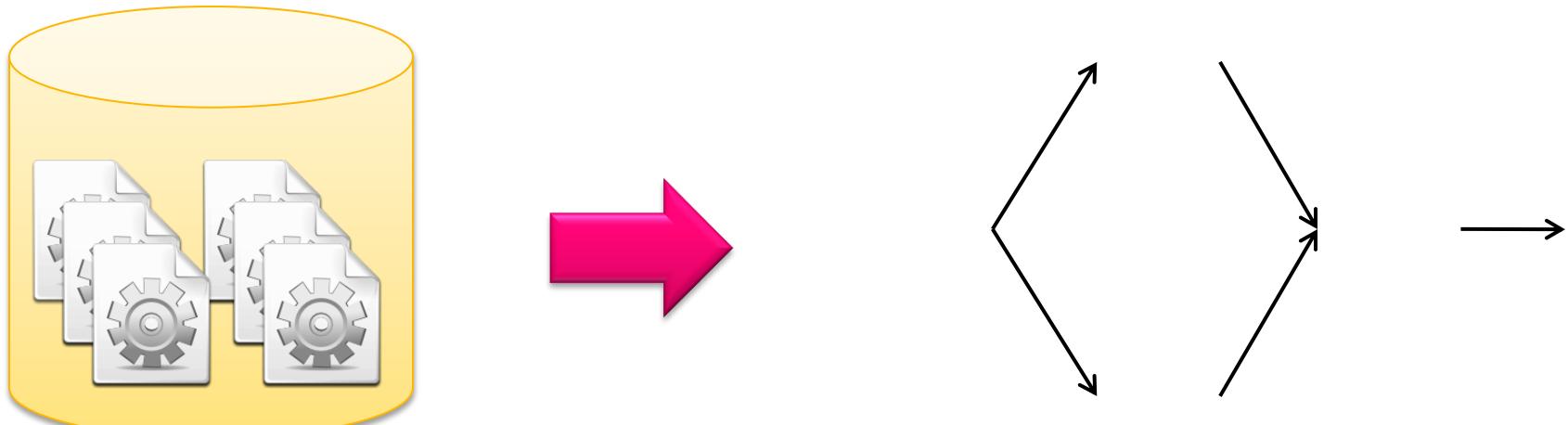


## La danse des services ....

Ou comment les applications à base de services sont composées ...

# Application à base de services

- ▶ Obtenue par composition de services



Registre de services

Consommateur de services

# Processus de composition de services

---

1. Définition de l'architecture fonctionnelle
2. Identification des services
3. Sélection et implantation des services
4. Médiation entre services
5. Déploiement et invocation des services

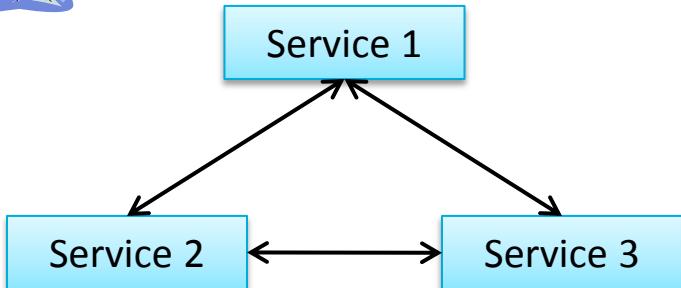


Processus complexe

# Composition à base de procédés

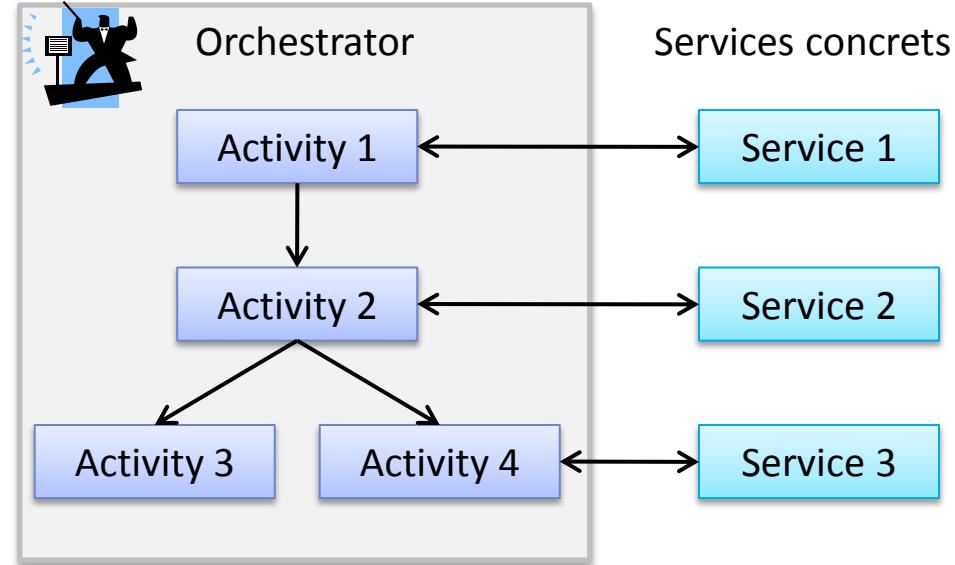


- ▶ Chorégraphie de services
  - ▶ Contrôle **distribué**
  - ▶ Interaction directe entre services



WS-CDL [W3C 2004]

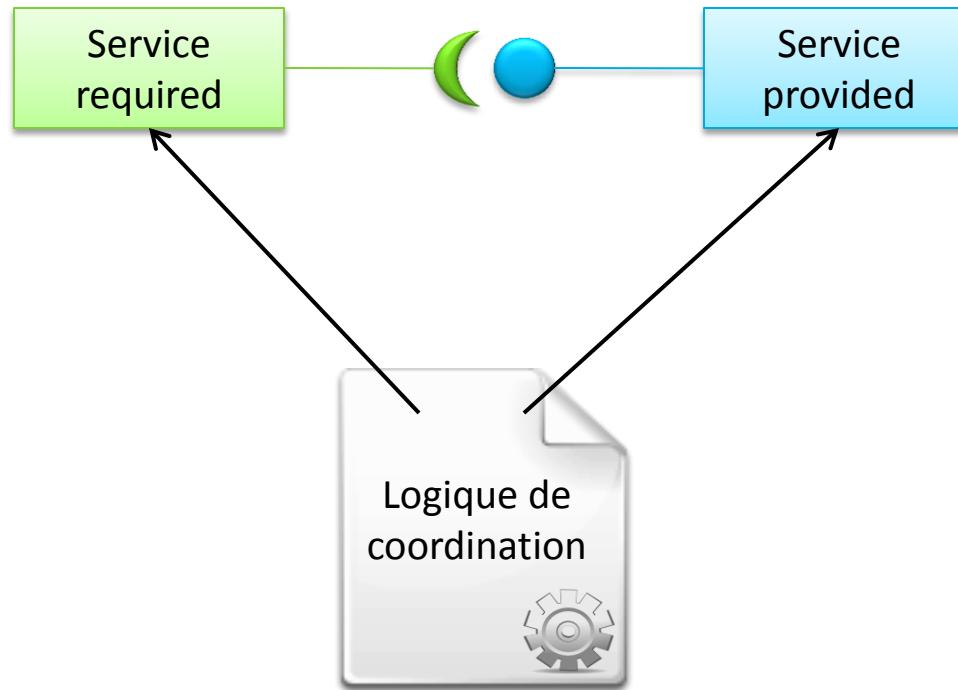
- ▶ Orchestration de services
  - ▶ Contrôle **centralisé**
  - ▶ Interaction indirecte entre services



WS-BPEL [OASIS 2007]

# Composition structurelle

- ▶ Services assemblés
  - ▶ Contrôle intrinsèque aux services
  - ▶ Dépendances à résoudre (syntaxique et sémantique)





## <(rappel ?) sur XML/>

XML est au cœur des échanges entre Services Web (SOAP)

# XML

---

- ▶ Standard du W3C
  - ▶ apparaît en 1991 sur la base de SGML
- ▶ Permet de **structurer des données** sous forme textuelle
- ▶ A base de **balises <nom attribut="valeur">**
- ▶ Les documents doivent être **bien formés** :
  - ▶ toujours balises ouvrantes <element> et fermantes </element>
    - ▶ ou balise ouvrante/fermante <element/>
  - ▶ la valeur d'un attribut est toujours entre double "

```
<?xml version="1.0" encoding ="UTF-8"?>
<salutation langue="français">Hello <espace/> world!</salutation>
```

# Notion de namespaces

- ▶ on utilise des **espaces de noms (namespaces)** pour lever l'ambiguïté entre balises de même nom mais de sémantique différente (`xmlns:prefix="URI"`)

```
<root>

<h:table xmlns:h="http://www.w3.org/TR/html4/">
  <h:tr>
    <h:td>Apples</h:td>
    <h:td>Bananas</h:td>
  </h:tr>
</h:table>

<f:table xmlns:f="http://www.w3schools.com/furniture">
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>

</root>
```

f est un raccourci donné au namespace

ici table est un élément du namespace HTML 4

ici table est un élément du namespace furniture

# Namespace par défaut

- ▶ On peut déclarer un namespace par défaut (`xmlns="namespaceURI"`)

```
<table xmlns="http://www.w3schools.com/furniture">  
    <name>African Coffee Table</name>  
    <width>80</width>  
    <length>120</length>  
</table>
```

Pas de préfixe, donc utilisé par défaut

Plus besoin de préfixer les balises

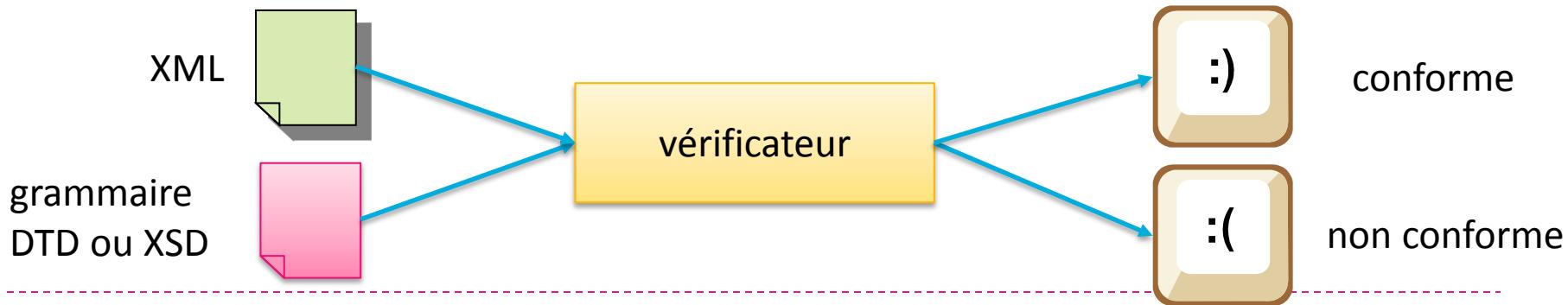
# Exemple de namespaces

- ▶ utilisés par <http://www.w3.org/TR/wsdl>

prefix	namespace URI	definition
wsdl	<a href="http://schemas.xmlsoap.org/wSDL/">http://schemas.xmlsoap.org/wSDL/</a>	WSDL namespace for WSDL framework.
soap	<a href="http://schemas.xmlsoap.org/wSDL/soap/">http://schemas.xmlsoap.org/wSDL/soap/</a>	WSDL namespace for WSDL SOAP binding.
http	<a href="http://schemas.xmlsoap.org/wSDL/http/">http://schemas.xmlsoap.org/wSDL/http/</a>	WSDL namespace for WSDL HTTP GET & POST binding.
mime	<a href="http://schemas.xmlsoap.org/wSDL/mime/">http://schemas.xmlsoap.org/wSDL/mime/</a>	WSDL namespace for WSDL MIME binding.
soapenc	<a href="http://schemas.xmlsoap.org/soap/encoding/">http://schemas.xmlsoap.org/soap/encoding/</a>	Encoding namespace as defined by SOAP 1.1 [8].
soapenv	<a href="http://schemas.xmlsoap.org/soap/envelope/">http://schemas.xmlsoap.org/soap/envelope/</a>	Envelope namespace as defined by SOAP 1.1 [8].
xsi	<a href="http://www.w3.org/2000/10/XMLSchema-instance">http://www.w3.org/2000/10/XMLSchema-instance</a>	Instance namespace as defined by XSD [10].
xsd	<a href="http://www.w3.org/2000/10/XMLSchema">http://www.w3.org/2000/10/XMLSchema</a>	Schema namespace as defined by XSD [10].
tns	(various)	The “this namespace” (tns) prefix is used as a convention to refer to the current document.
(other)	(various)	All other namespace prefixes are samples only. In particular, URIs starting with “ <a href="http://example.com">http://example.com</a> ” represent some application-dependent or context-dependent URI [4].

# Grammaires

- ▶ A priori on peut utiliser n'importe quel nom pour les balises
- ▶ La syntaxe du document ***peut*** être contrainte via des grammaires
  - ▶ La grammaire constraint l'ensemble des balises et leurs usages
  - ▶ deux types : **DTD** ou des **XML Schema**
- ▶ on peut alors vérifier la **conformité** du document à la grammaire



# Exemple de Shema XML

## ► Les XSD s'écrivent en .... XML

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xsschema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xselement name="people" type="personne"/>

<xsccomplexType name="personne">
  <xsssequence>
    <xselement name="age" type="xs:int"/>
    <xselement name="nom" type="xs:string" minOccurs="0"/>
    <xselement name="prenom" type="xs:string" minOccurs="1"/>
  </xsssequence>
</xsccomplexType>
</xsschema>
```

Contrainte sur les types

Contrainte sur la cardinalité  
XML

# Java et XML

---

- ▶ **Le pénible** : analyse/parsing du XML
  - ▶ **DOM (Document Object Model)** : on charge tout le document et on observe
  - ▶ **(Simple API for XML)** : des évènements sont déclenchés à la lecture du fichier
  
- ▶ **La magie** : Bijection entre JAVA et XML :
  - ▶ **JAXB**
  - ▶ JIXB
  - ▶ XMLBeans
  - ▶ Castor
  - ▶ ADB
  - ▶ ...

# JAXB

---

- ▶ En standard dans Java 6 et suivant
- ▶ Réalise l'**emballage/déballage** automatique de XML vers Java et inversement.
- ▶ Permet de générer :
  - ▶ des classes Java à partir d'un Schema XML (commande xjc ...)
  - ▶ un Schema XML à partir de classes java (commande shemagen)
- ▶ JAXB utilise des **annotations** sur les classes JAVA pour préciser la correspondance avec le XML

# Exemple de classe annotée

```
package jaxb.test;  
  
import javax.xml.bind.annotation.XmlRootElement;  
  
@XmlRootElement(name = "people")  
public class Personne {  
  
    protected int age;  
    protected String nom;  
    protected String prenom;  
  
    public int get...  
  
}
```

Le nom de la racine (par défaut le nom de la classe)

L'annotation la plus utile,  
souvent suffisante pour des  
petits fichiers

# Génération de la xsd associée

- ▶ avec la commande **shemagen** sur la classe java

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="people" type="personne"/>

  <xs:complexType name="personne">
    <xs:sequence>
      <xs:element name="age" type="xs:int"/>
      <xs:element name="nom" type="xs:string" minOccurs="0"/>
      <xs:element name="prenom" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

- ▶ C'est comme ça que les outils automatiques génèrent les types en WDSL

# Exemple d'emballage (Java -> XML)

- ▶ On utilise un marshaller (emballage de Java vers XML)

```
public static void main(String[] args) throws JAXBException {  
  
    JAXBContext jc = JAXBContext.newInstance(Personne.class);  
    Marshaller marshaller = jc.createMarshaller();  
  
    Personne p = new Personne();  
    p.setAge(13);  
    p.setNom("Doe");  
    p.setPrenom("John");  
  
    marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);  
    marshaller.marshal(p, System.out);  
  
}
```

# Exemple d'emballage (Java -> XML)

## ► Résultat

```
Personne p = new Personne();
p.setAge(13);
p.setNom("Doe");
p.setPrenom("John");
```



emballage

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<people>
    <age>13</age>
    <nom>Doe</nom>
    <prenom>John</prenom>
</people>
```

# Exemple de déballage (XML -> Java)

```
public static void main(String[] args) throws JAXBException {  
  
    JAXBContext jc = JAXBContext.newInstance(Personne.class);  
    Unmarshaller unmarshaller = jc.createUnmarshaller();  
  
    Personne p = (Personne) unmarshaller.unmarshal(new File("people.xml"));  
  
    System.out.println(p.getPrenom() + " " + p.getNom() + " " + p.getAge());  
  
}
```

# Exemple de déballage (XML -> Java)

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<people>
    <age>94</age>
    <nom>Smith</nom>
    <prenom>Janet</prenom>
</people>
```



déballage

Janet Smith 94

# Quelques annotations ...

---

- ▶ `@XmlRootElement(namespace = "namespace")` permet de définir un namespace
  - ▶ `@XmlType(propOrder = { "field2", "field1",.. })` permet de définir l'ordre des éléments
  - ▶ `@XmlElement(name = "nom")` permet de définir le nom d'un élément particulier
- 
- ▶ A vous de voir ...
    - ▶ <https://jaxb.java.net/>



## Les services Web (à la SOAP)

La pratique ....



# Services Web

Les services Web sont des applications utilisant Internet pour interagir dynamiquement avec d'autres programmes.

## ▶ Plusieurs standard :

- ▶ XML pour le formatage des données
- ▶ HTTP (en général) pour le transport des données
- ▶ WSDL pour la description des services
- ▶ SOAP pour la communication
- ▶ UDDI pour la découverte



# Services Web

Les services Web sont des applications utilisant Internet pour interagir dynamiquement avec d'autres programmes.

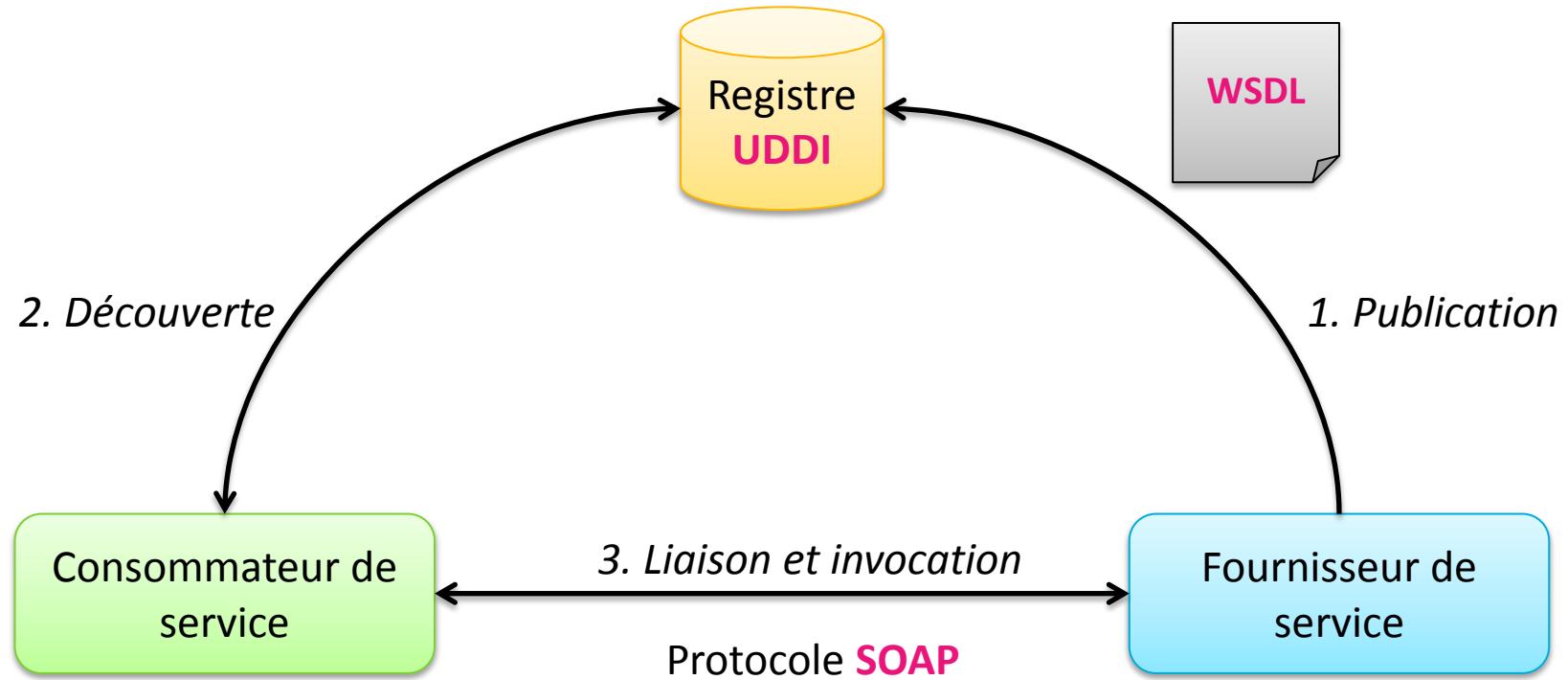
## ▶ Plusieurs standard :

- ▶ XML pour le formatage des données
- ▶ HTTP (en général) pour le transport des données
- ▶ WSDL pour la description des services
- ▶ SOAP pour la communication
- ▶ UDDI pour la découverte

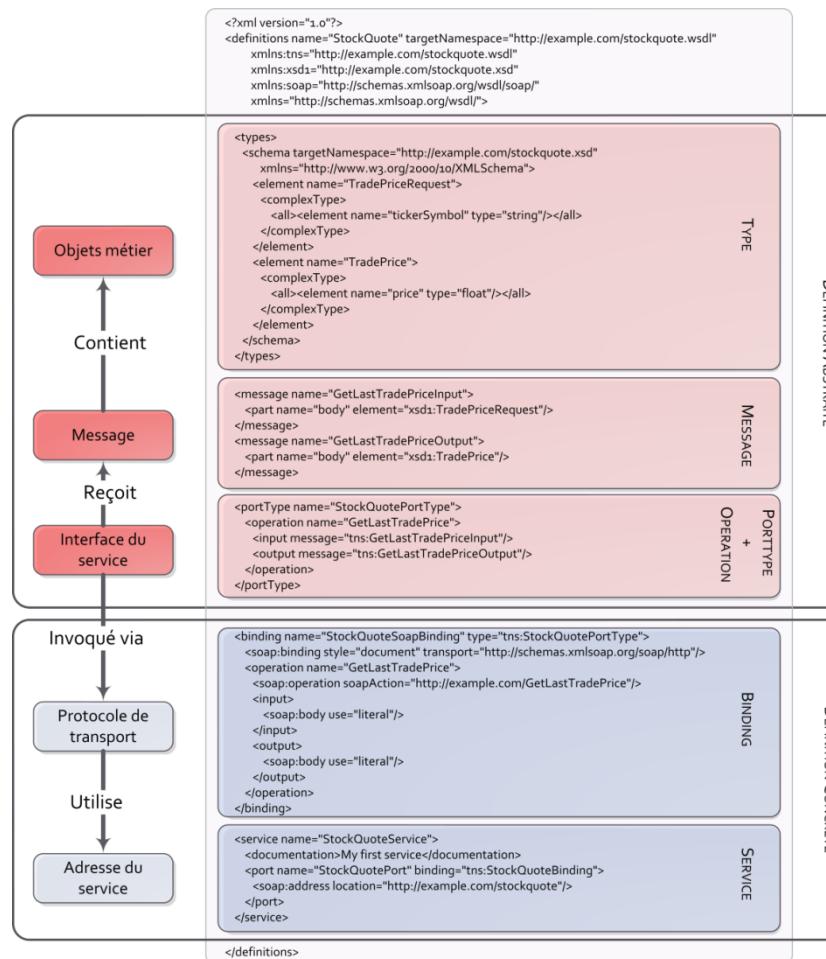
Très peu utilisé



# Architecture



# A quoi ressemble une description en WDSL ?



ceci est un petit service .... :)

# Ca à l'air **vraiment** pénible à écrire...

---

- ▶ C'est vrai ... mais ...



- ▶ Vous n'aurez sans doute pas à en écrire (générateur)
- ▶ Il faut prendre les choses par morceaux ....

# Imaginons un service simple

---

- ▶ Equivalent Java de ce que dois faire le service :

```
package fr.istic.mri.hello;

public class Hello {

    public String getHello(String name) {
        return "Hello " + name + "!";
    }

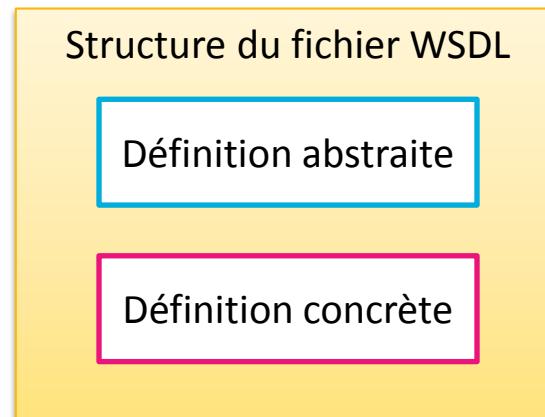
}
```

- ▶ et décrivons ça en WSDL ...

# Structure d'un fichier WSDL

---

- ▶ Définition abstraite de l'interface :
  - ▶ Opérations supportées ainsi que les paramètres et les types
- ▶ Définition concrète de l'accès au service :
  - ▶ Localisation par une adresse réseau du fournisseur
  - ▶ Protocoles spécifiques d'accès



# Détail de la définition abstraite – 1/3

## ▶ Type :

- ▶ Description des types de données
  - ▶ Tirés de la spécification XML Schema (xsd:string)
  - ▶ Définis par l'utilisateur

```
<?xml version="1.0" encoding="utf-8"?><xs:schema xmlns:tns="http://soap.dwp.mri/" xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="unqualified" targetNamespace="http://soap.dwp.mri/" version="1.0">
<xs:element name="getVersion" type="tns:getVersion"/>
<xs:element name="getVersionResponse" type="tns:getVersionResponse"/>
<xs:element name="hello" type="tns:hello"/>
<xs:element name="helloResponse" type="tns:helloResponse"/>
<xs:complexType name="getVersion">
<xs:sequence/>
</xs:complexType>
<xs:complexType name="getVersionResponse">
<xs:sequence>
<xs:element minOccurs="0" name="return" type="xs:string"/>
</xs:sequence>
</xs:complexType>
<xs:complexType name="hello">
<xs:sequence>
<xs:element minOccurs="0" name="arg0" type="xs:string"/>
</xs:sequence>
</xs:complexType>
<xs:complexType name="helloResponse">
<xs:sequence>
<xs:element minOccurs="0" name="return" type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:schema>
```

Définition abstraite

Type

Message

PortType

Operation

# Détail de la définition abstraite – 2/3

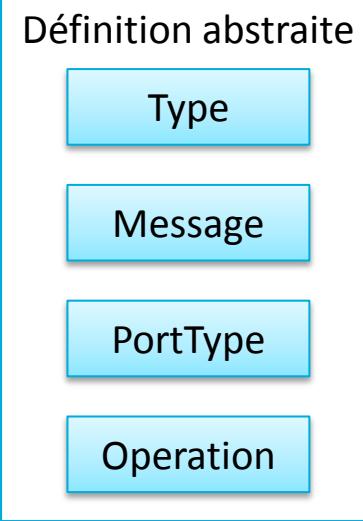
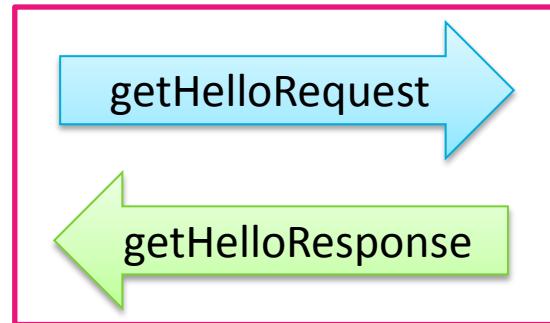
## ▶ Message :

- ▶ Messages unidirectionnels :
  - ▶ entrant : in, sortant : out
- ▶ Un message comporte plusieurs arguments <part>
  - ▶ Chaque argument a un type

## ▶ Opération :

- ▶ Comporte en général 2 messages : requête et réponse

Opération getHello



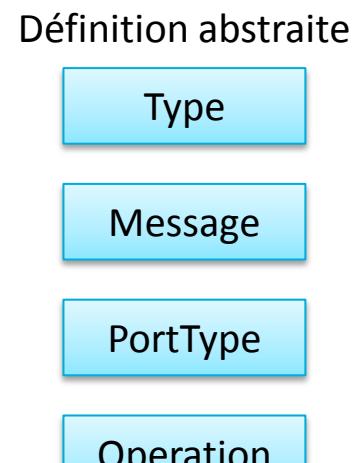
# Détail de la définition abstraite – 3/3

- ▶ PortType (WSDL 2 : interface) :
  - ▶ Ensemble d'opérations

```
<wsdl:message name="getHelloRequest">
  <wsdl:part name="in0" type="soapenc:string"/>
</wsdl:message>

<wsdl:message name="getHelloResponse">
  <wsdl:part name="getHelloReturn" type="soapenc:string"/>
</wsdl:message>

<wsdl:portType name="Helloworld">
  <wsdl:operation name="getHello" parameterOrder="in0">
    <wsdl:input message="impl:getHelloRequest" name="getHelloRequest"/>
    <wsdl:output message="impl:getHelloResponse" name="getHelloResponse"/>
  </wsdl:operation>
</wsdl:portType>
```



# Détail de la définition concrète – 1/2

Définition concrète

Binding

Service

## ▶ Binding :

- ▶ Définit un lien avec le PortType abstrait d'un service réel associé à un protocole et un format

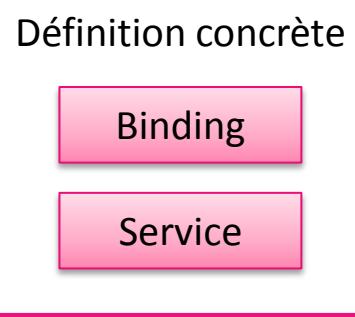
```
<wsdl:binding name="helloSoapBinding" type="impl:Helloworld">
    <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="getHello">
        <wsdlsoap:operation soapAction="" />
        <wsdl:input name="getHelloRequest">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="http://hello" use="encoded" />
        </wsdl:input>
        <wsdl:output name="getHelloResponse">
            <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="http://localhost:8080/axis/services/hello" use="encoded" />
        </wsdl:output>
    </wsdl:operation>
</wsdl:binding>
```

# Détail de la définition concrète – 2/2

## ► Service :

- ▶ Le port définit la localisation concrète (endpoint) du service

```
<wsdl:service name="HelloWorldService">
    <wsdl:port binding="impl:helloSoapBinding" name="hello">
        <wsdlsoap:address location="http://localhost:8080/axis/services/hello"/>
    </wsdl:port>
</wsdl:service>
```



= URL de la servlet

## ► Remarques :

- ▶ Chaque portType n'a qu'un seul binding et chaque binding n'a qu'un seul portType
- ▶ Chaque service peut avoir plusieurs portType, chacun avec une adresse et un protocole différent

# L'annuaire UDDI

UDDI est une spécification d'annuaire qui propose d'enregistrer et de rechercher des fichiers de description de services Web correspondant aux attentes d'un client.

- ▶ Objectif d'UDDI :
  - ▶ Connaître les entreprises qui fournissent des services Web
  - ▶ Découvrir les services Web disponibles qui répondent aux attentes du client
- ▶ Trois types d'annuaires :
  - ▶ Les **pages blanches** permettent de connaître les informations concernant les entreprises
  - ▶ Les **pages jaunes** présentent les services selon leurs fonctionnalités en utilisant une taxonomie industrielle standard
  - ▶ Les **pages vertes** informent sur les services fournis par les entreprises

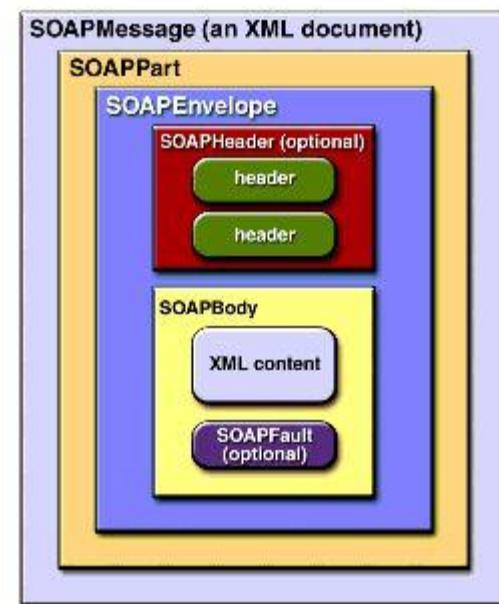
# Protocole SOAP

Le protocole SOAP permet des échanges standardisés de données dans des environnements distribués. SOAP est basé sur XML.

- ▶ S'appuie sur des standards de communication :
  - ▶ HTTP
  - ▶ SMTP
  - ▶ ...

# Structure d'un message SOAP

- ▶ En-tête optionnel :
  - ▶ Permet d'ajouter des extensions : sécurité, transactions...
- ▶ Corps obligatoire :
  - ▶ Contient la méthode à invoquer ainsi que les paramètres



# Exemple de requête

```
POST /StockQuote HTTP/1.1
```

```
Host: www.stockquoteserver.com
```

```
Content-Type: text/xml;
```

```
charset="utf-8"
```

```
Content-Length: nnnn
```

```
SOAPAction: "Some-URI"
```

```
<SOAP-ENV:Envelope  
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"  
    SOAP-  
    ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">>  
    <SOAP-ENV:Body>  
        <m:GetLastTradePrice  
            xmlns:m="Some-URI">  
            <symbol>MOT</symbol>  
        </m:GetLastTradePrice>  
    </SOAP-ENV:Body>  
</SOAP-ENV:Envelope>
```

HTTP

Enveloppe SOAP

# Réponse

HTTP

HTTP/1.1 200 OK Content-Type: text/xml; charset="utf-8"

Content-Length: nnnn

```
<SOAP-ENV:Envelope  
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"  
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">  
    <SOAP-ENV:Body>  
        <m:GetLastTradePriceResponse  
            xmlns:m="Some-URI">  
            <Price>14.5</Price>  
        </m:GetLastTradePriceResponse>  
    </SOAP-ENV:Body>  
</SOAP-ENV:Envelope>
```

# Quelques détails

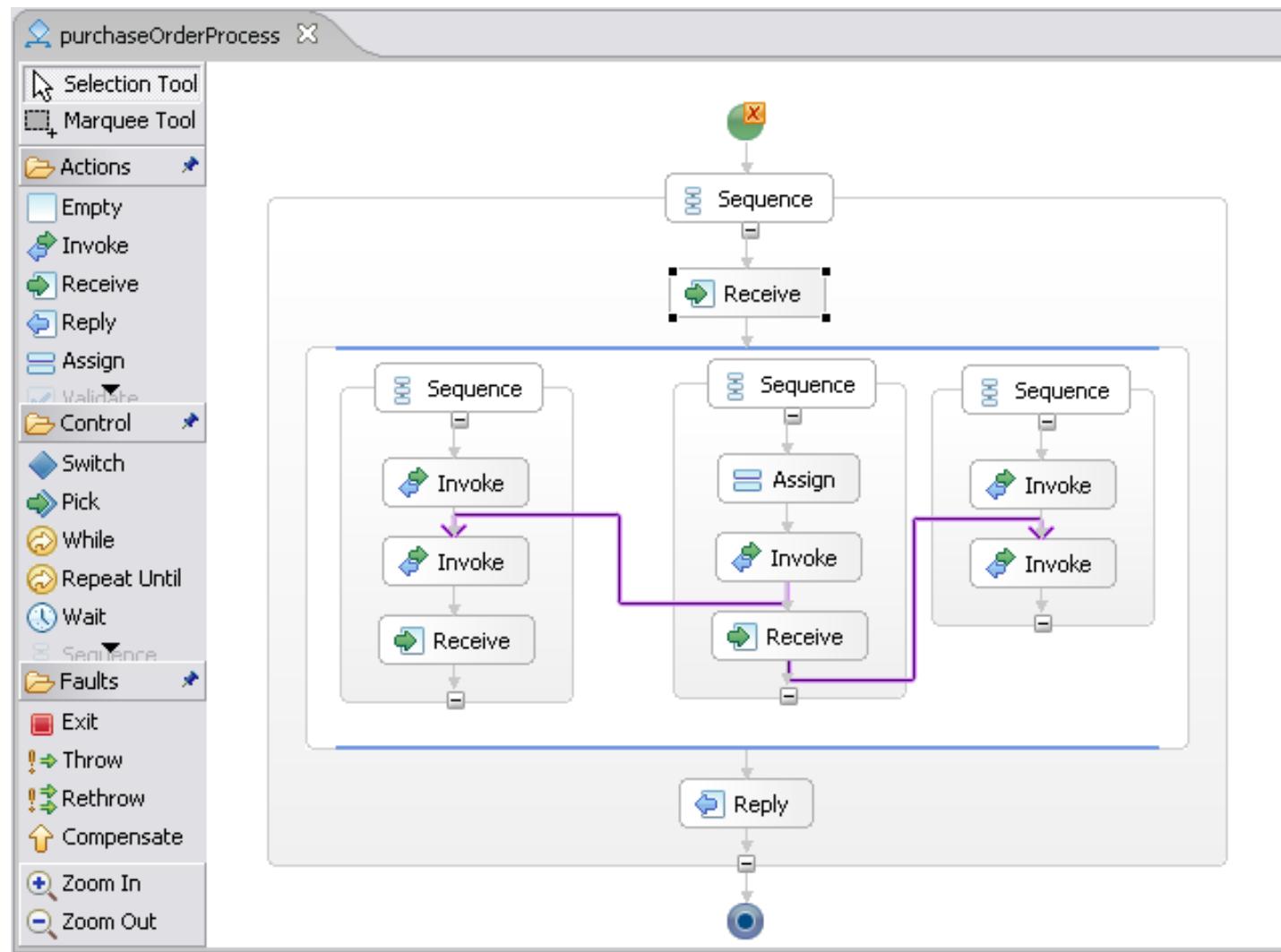
---

- ▶ header : permet de rajouter des informations sur la requête

```
<SOAP-ENV:Header>
<Transaction SOAP-ENV:mustUnderstand="1">
5
</Transaction>
</SOAP-ENV:Header>
```

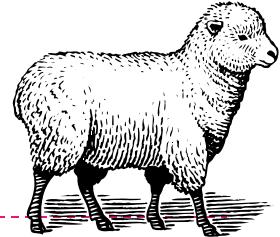
- ▶ fault : permet de signaler une erreur dans l'exécution
- ▶ body : contient le corps du message
  - ▶ suivant la méthode de communication WSDL utilisée, la forme de l'enveloppe est différente
    - ▶ typage des paramètres notamment en RPC/encoded

# Composition de Services Web

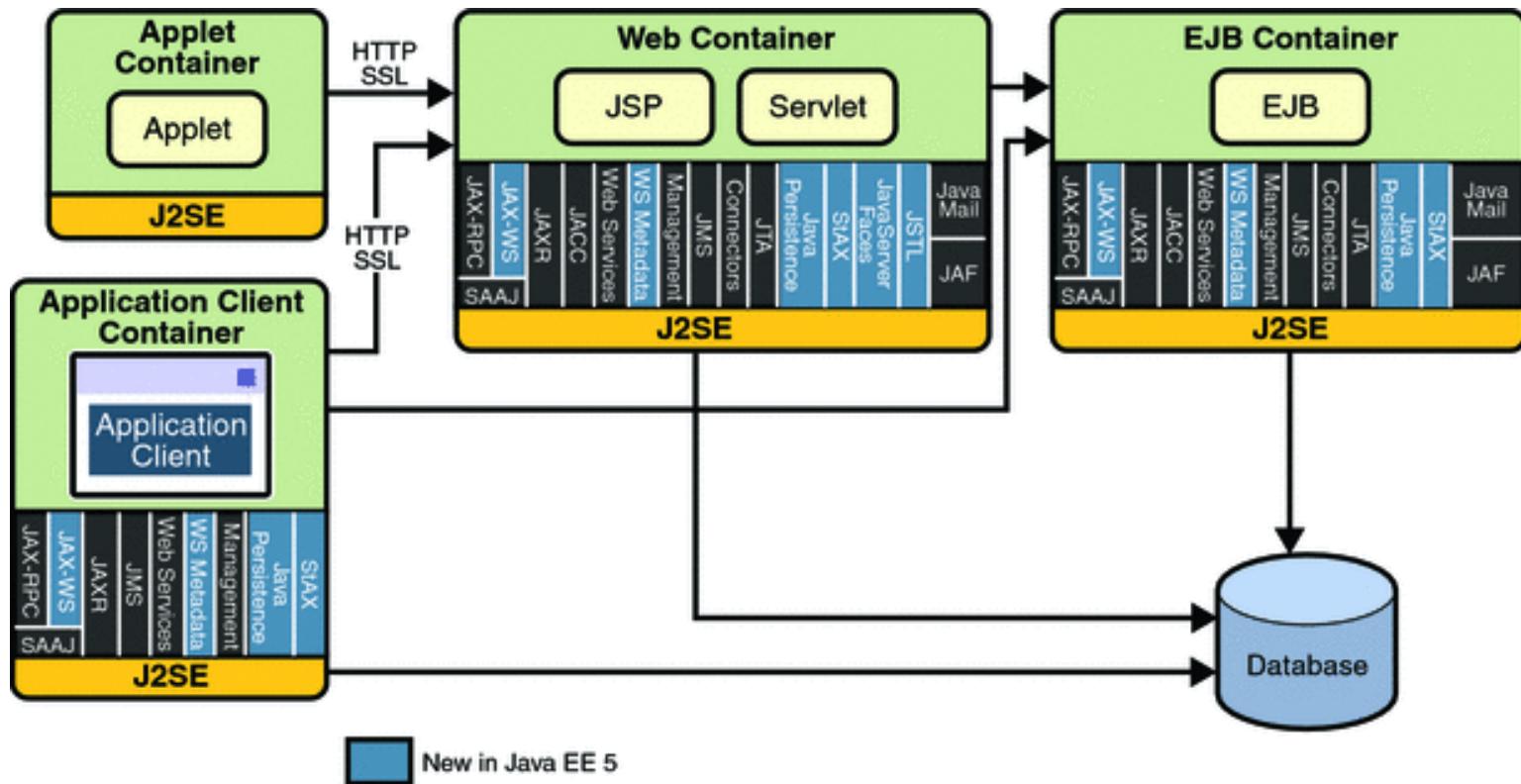




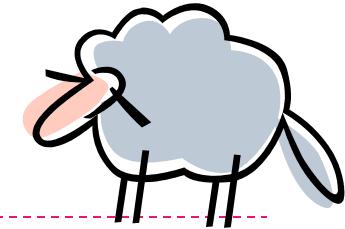
# Développement avec JAX-WS, CXF et Eclipse



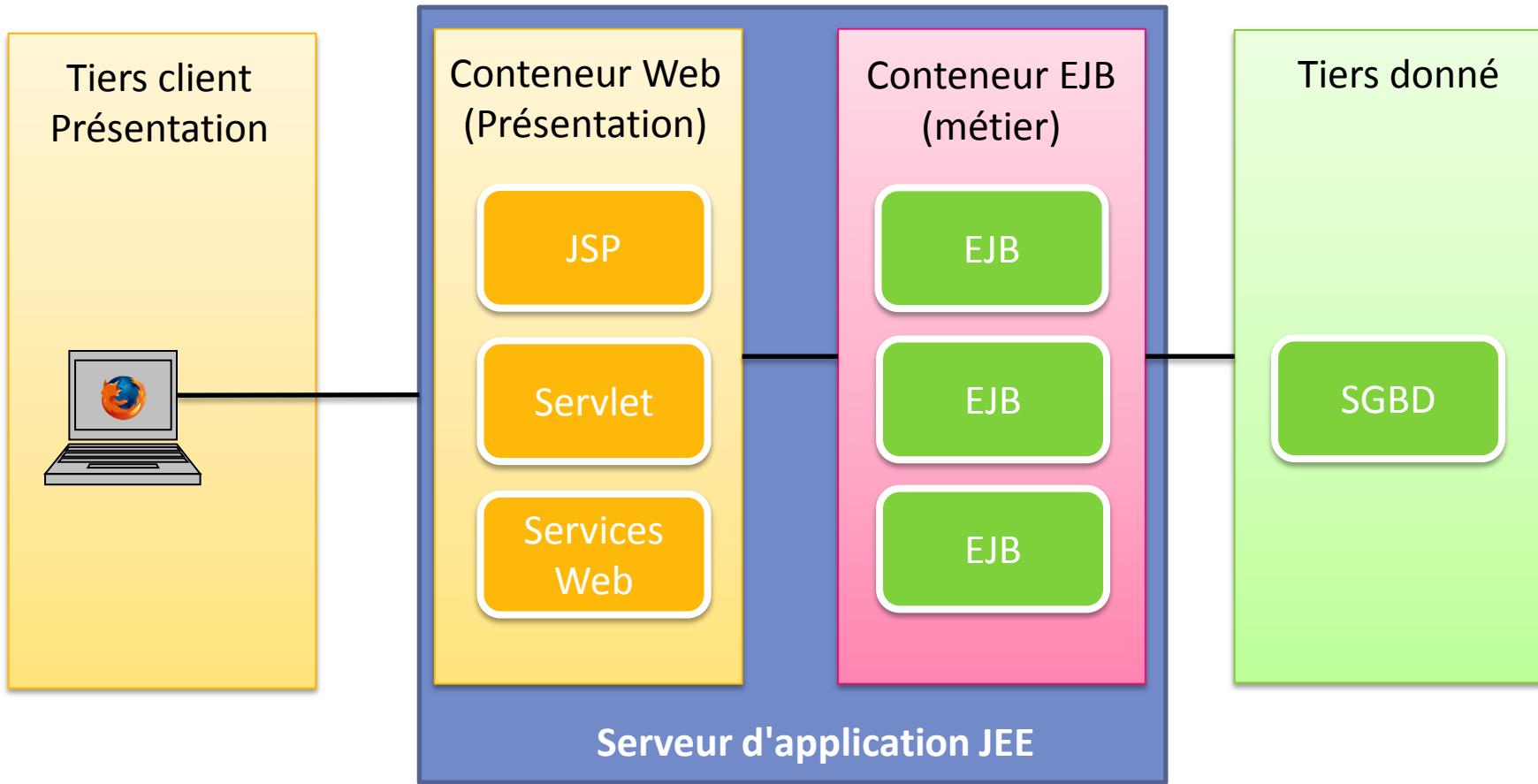
# Dessine moi un Java EE :



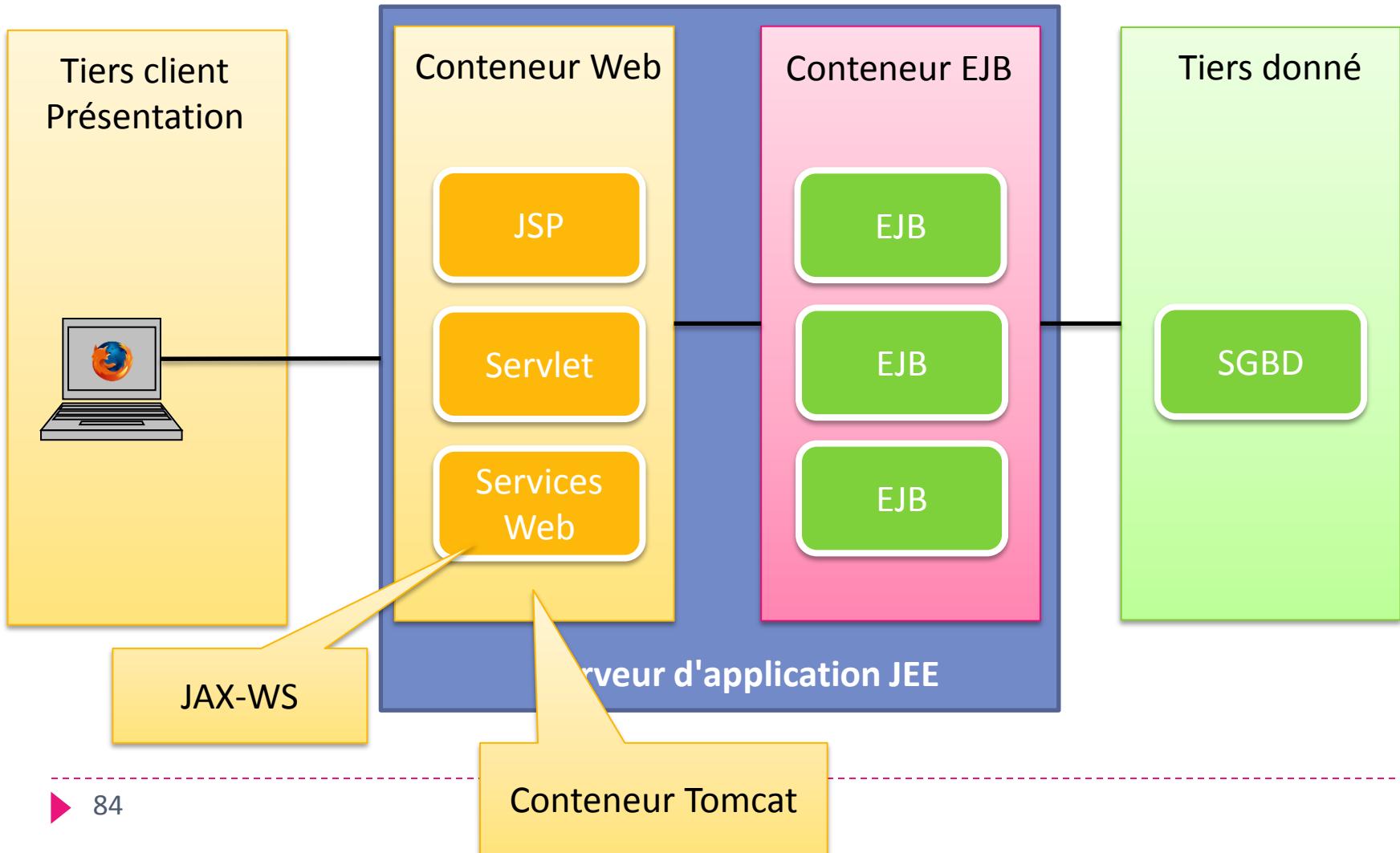
<http://docs.oracle.com/javaee/5/tutorial/doc/bnacj.html>



# Avec moins de gros mots ...



# Aujourd'hui



# Piles d'implantation possibles

---

- ▶ Spécifications pour les Web Services en Java :
  - ▶ JAX-WS
- ▶ Technologies :
  - ▶ Apache Axis 1.4
  - ▶ Apache Axis 2
  - ▶ Codehaus XFire
  - ▶ Apache CXF
  - ▶ Metro



Nécessite des configurations particulières !!!

# Configuration

## ▶ Outils :

- ▶ Java : jdk 1.6 ou supérieur
- ▶ Eclipse
- ▶ ~~Ant ou Maven~~
- ▶ Apache Tomcat

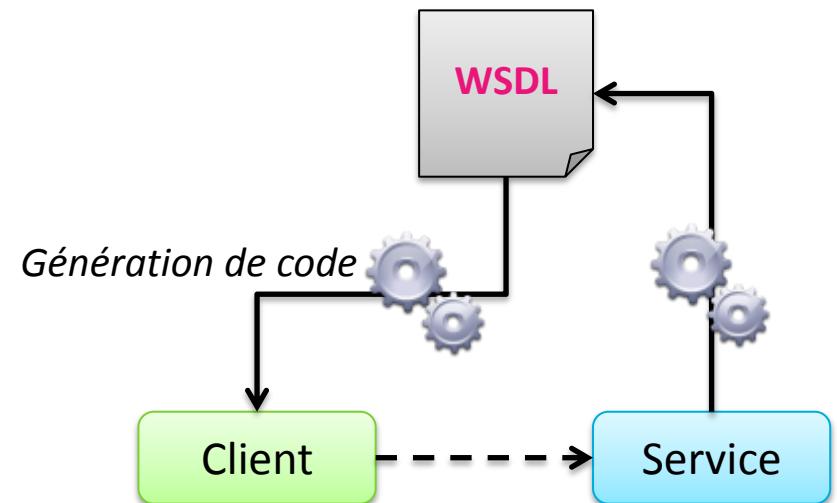
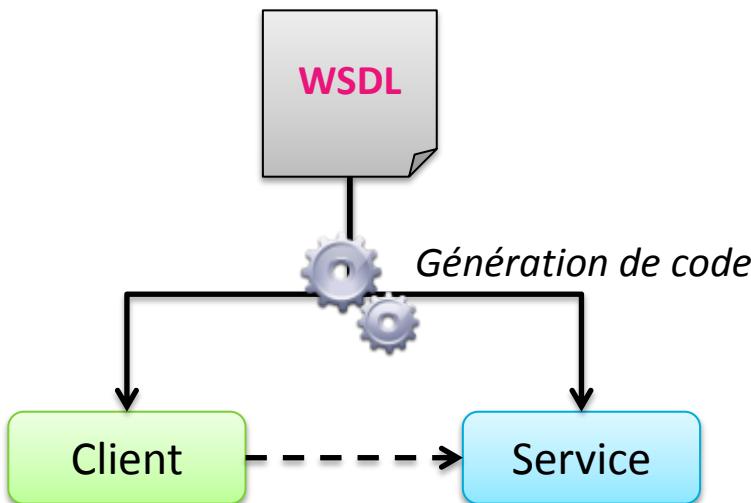
On va s'en passer mais je vous encourage à regarder

## ▶ Bibliothèque :

- ▶ Apache CXF : <http://cxf.apache.org/download.html>

# Comparaison de déploiement

- ▶ Génération de code à partir de l'interface
- ▶ Génération de l'interface à partir du code



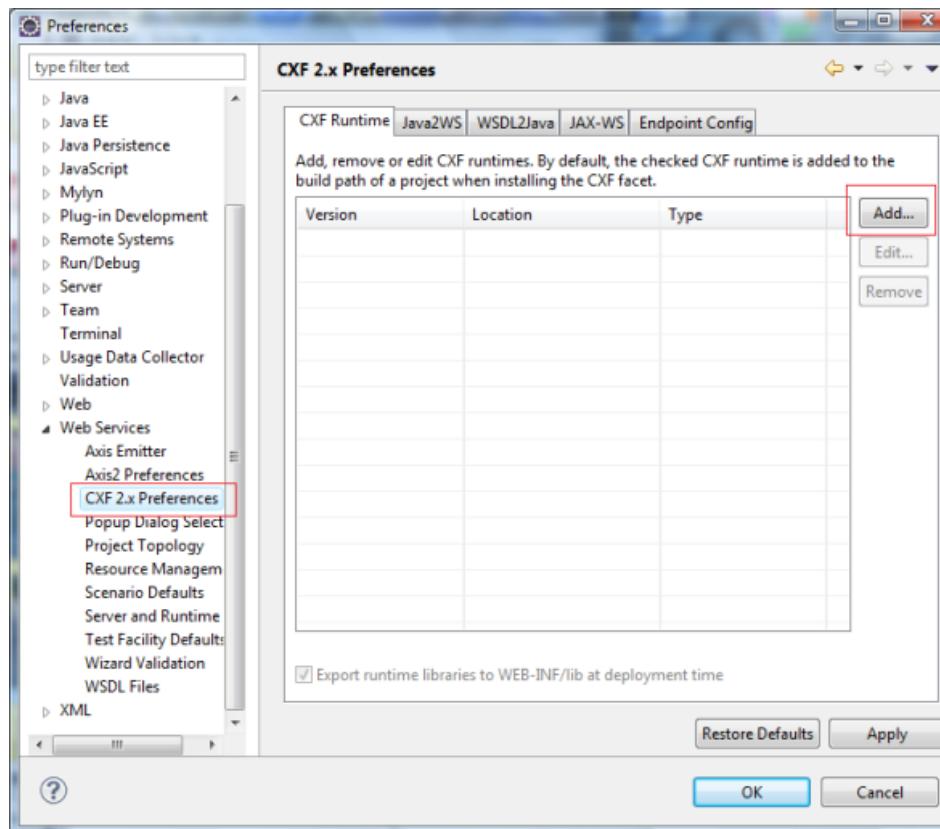
# Etapes

---

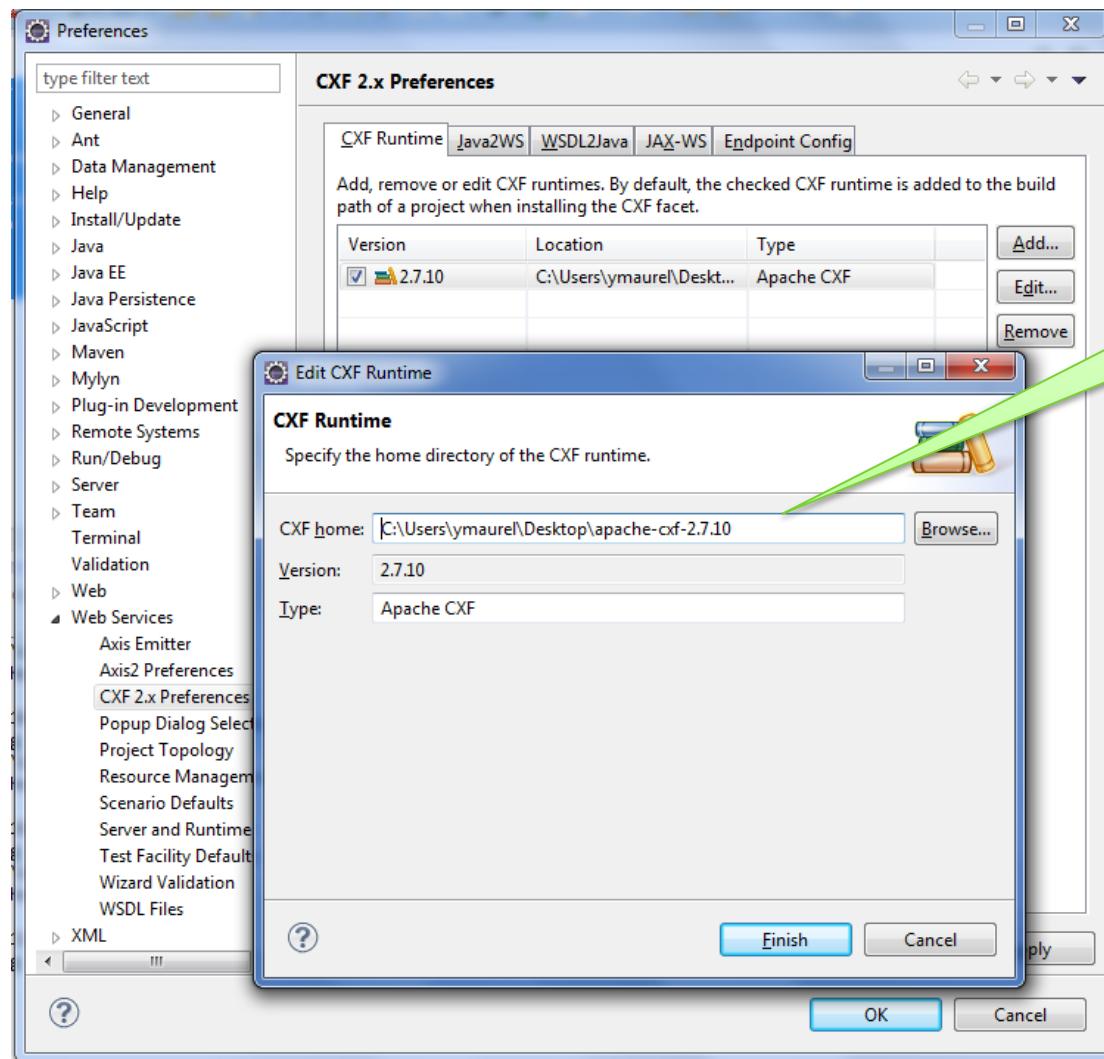
- ▶ Implanter et déployer un service Web
  - 1. Définir l'interface du service Web
  - 2. Implanter le service Web
  - 3. Configurer le service Web
  - 4. Packager le service Web
  - 5. Déployer le service Web
- ▶ Implanter un client
  - 1. Générer les classes nécessaires à partir du WSDL
  - 2. Implanter l'appel au service Web

# Configuration de CXF dans Eclipse JEE

- ▶ Il faut d'abord télécharger la dernière version de CXF (on utilisera la 2.x.x)
- ▶ Et configurer le plugin Eclipse de CXF dans Préférences

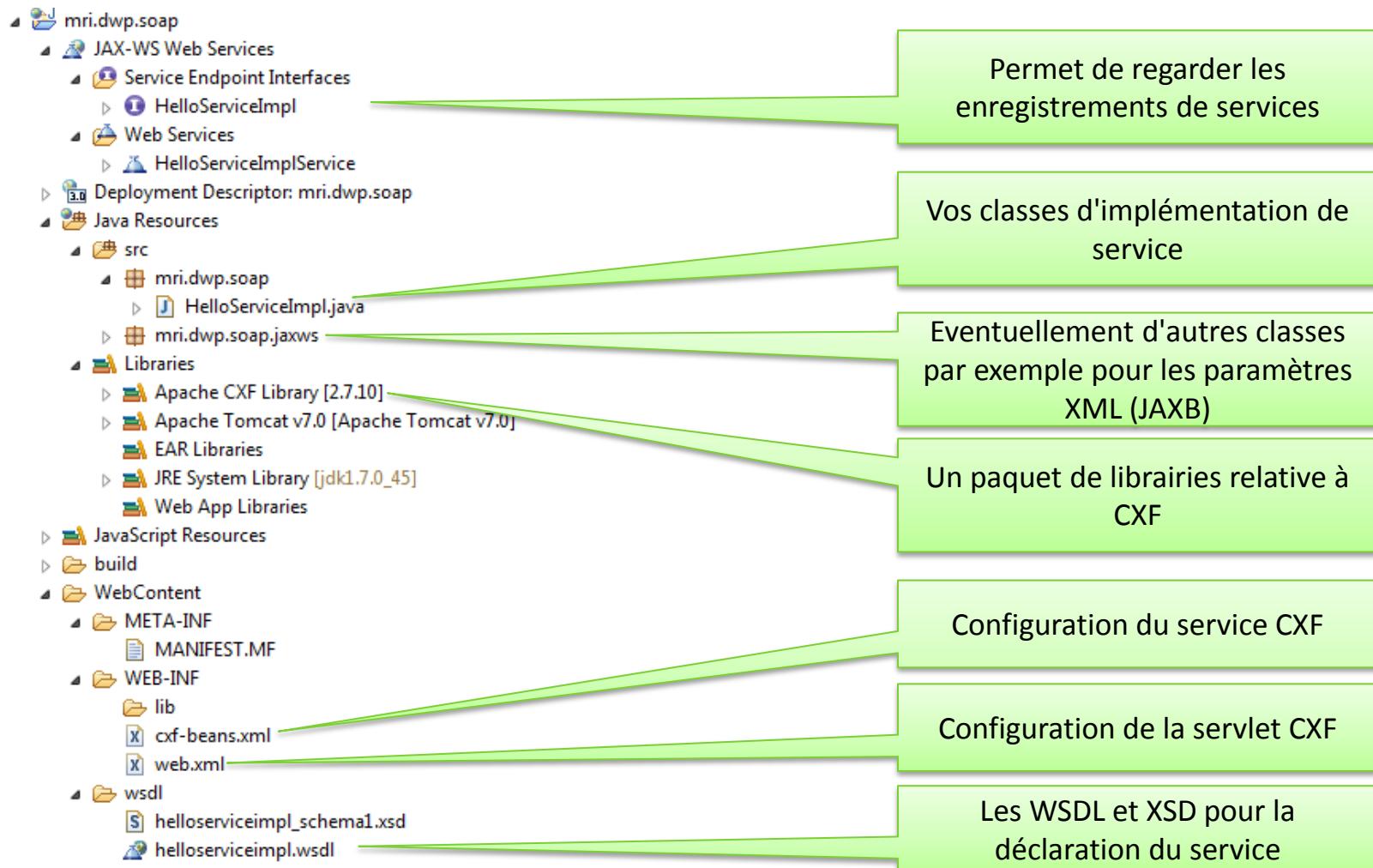


# Configuration de CXF dans Eclipse JEE



Pointe sur  
l'installation de  
CXF

# Projet : Dynamic Web Project



# Implémentation du service (un POJO)

- ▶ La description du service est une interface simple

```
package fr.esisar.service.hello;  
  
import javax.jws.WebService;  
  
public interface HelloInterface {  
  
    public String sayHello(String name);  
}
```

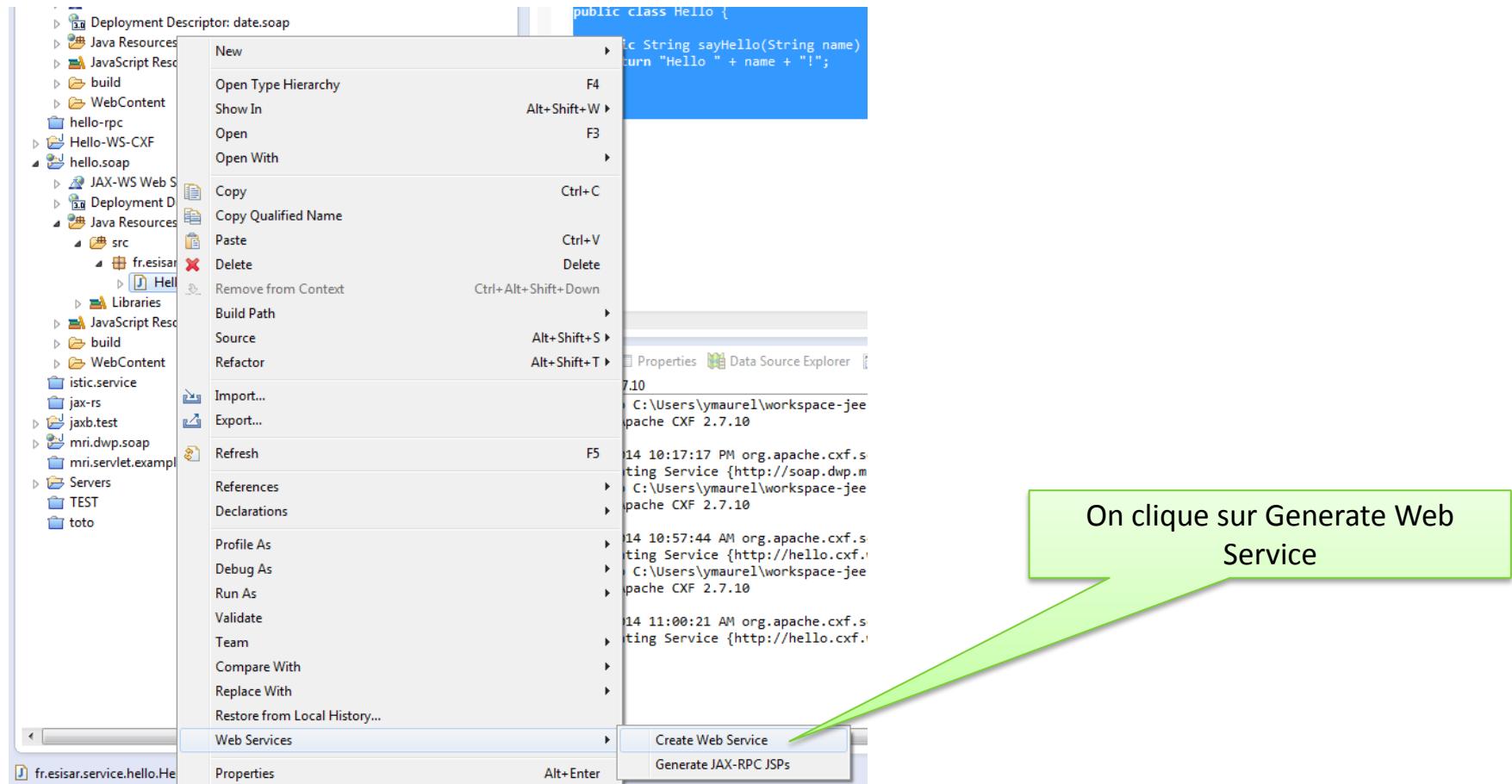
- ▶ Il va falloir l'annoter avec du JAX-WS

# On peut annoter avant la génération (conseillé)

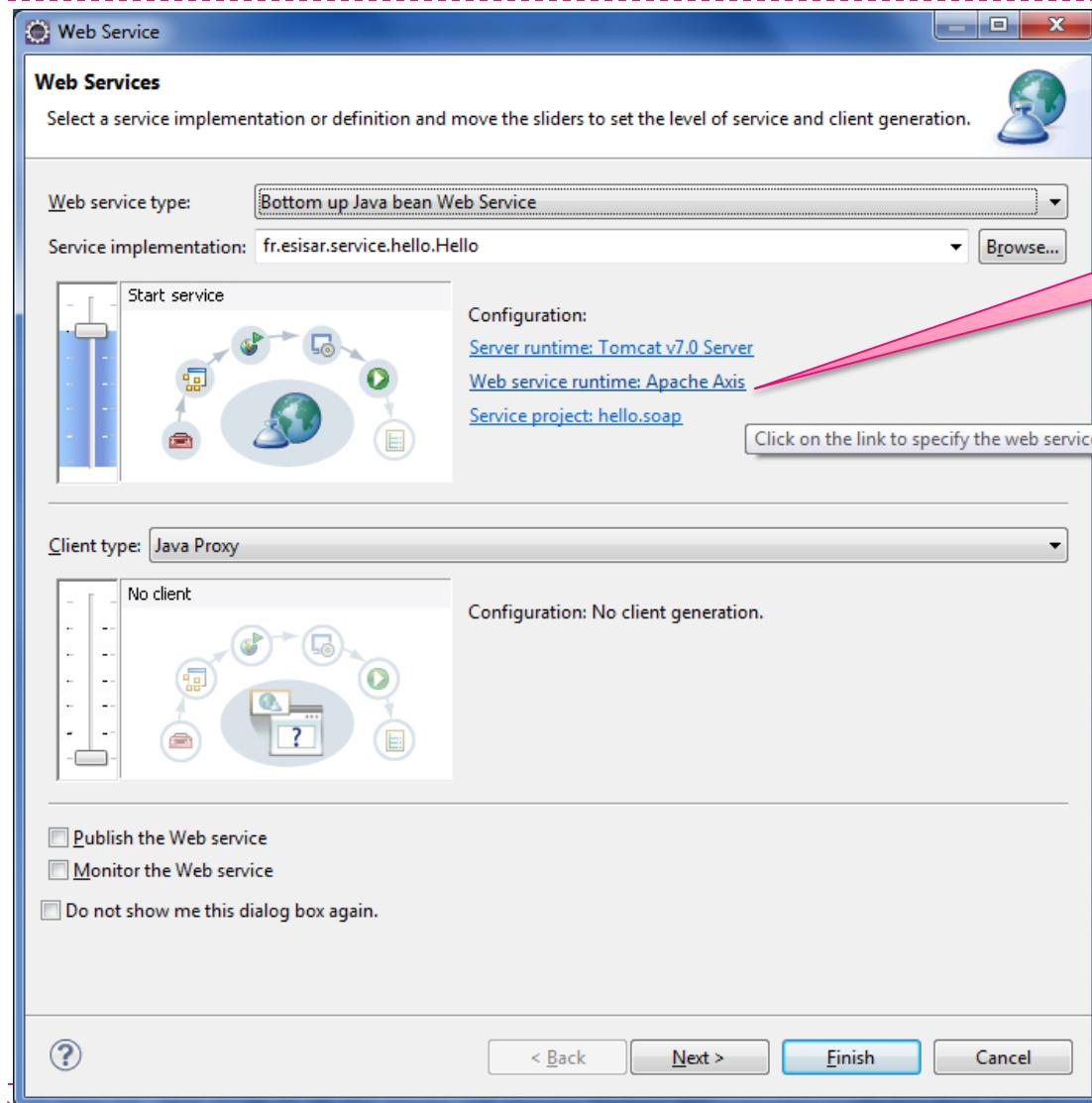
- ▶ C'est la solution conseillée si vous voulez maîtriser le nom de vos paramètres et de vos méthodes dans le WSDL

```
@WebService  
public class Hello {  
  
    @WebMethod(operationName = "sayHello", action = "urn:SayHello")  
    @WebResult(name = "helloGreeting")  
    public String sayHello(@WebParam(name = "name") String name) {  
        return "Hello " + name + "!";  
    }  
}
```

# On utilise l'outil de génération de Web Service

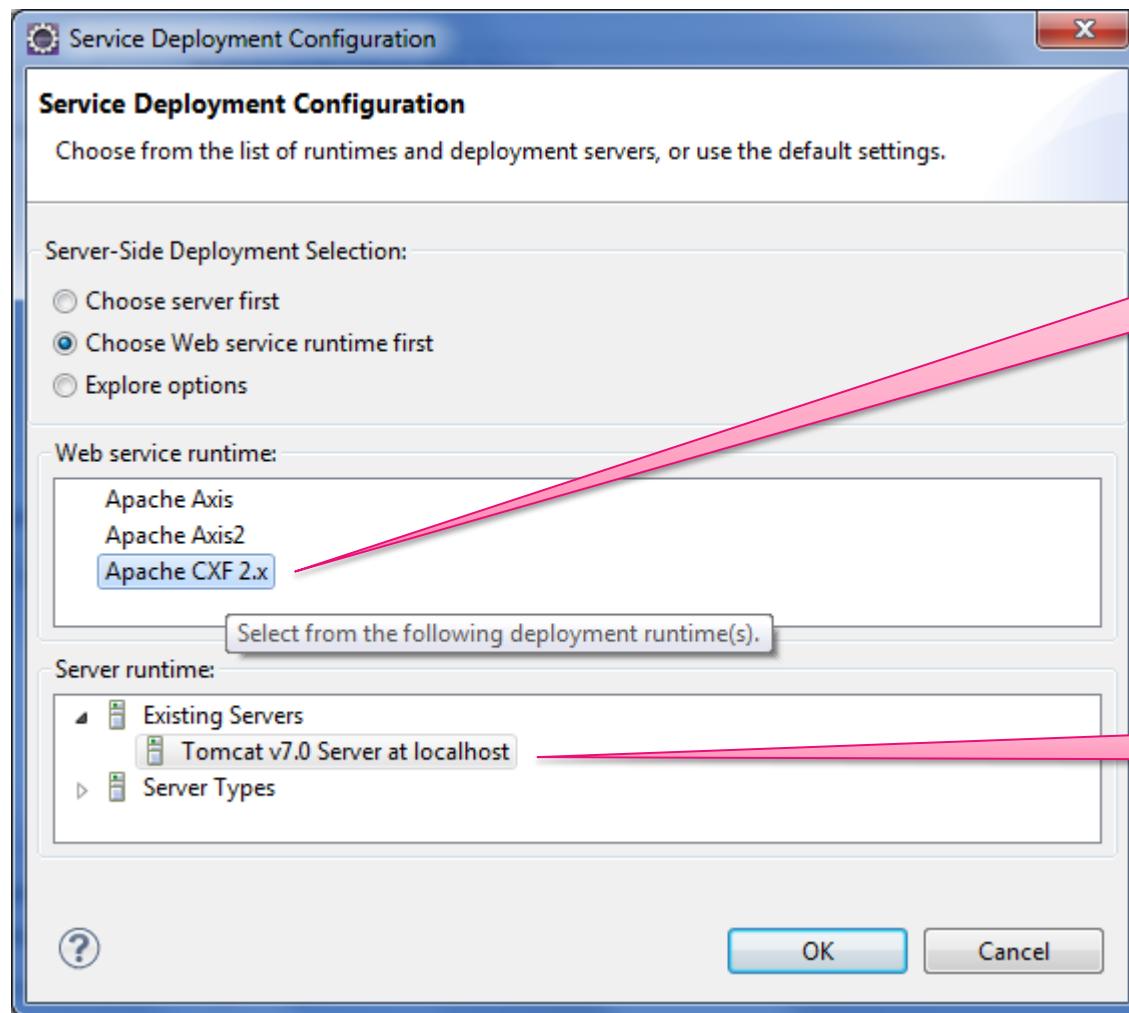


# On utilise l'outil de génération de Web Service



On doit sélectionner CXF !!

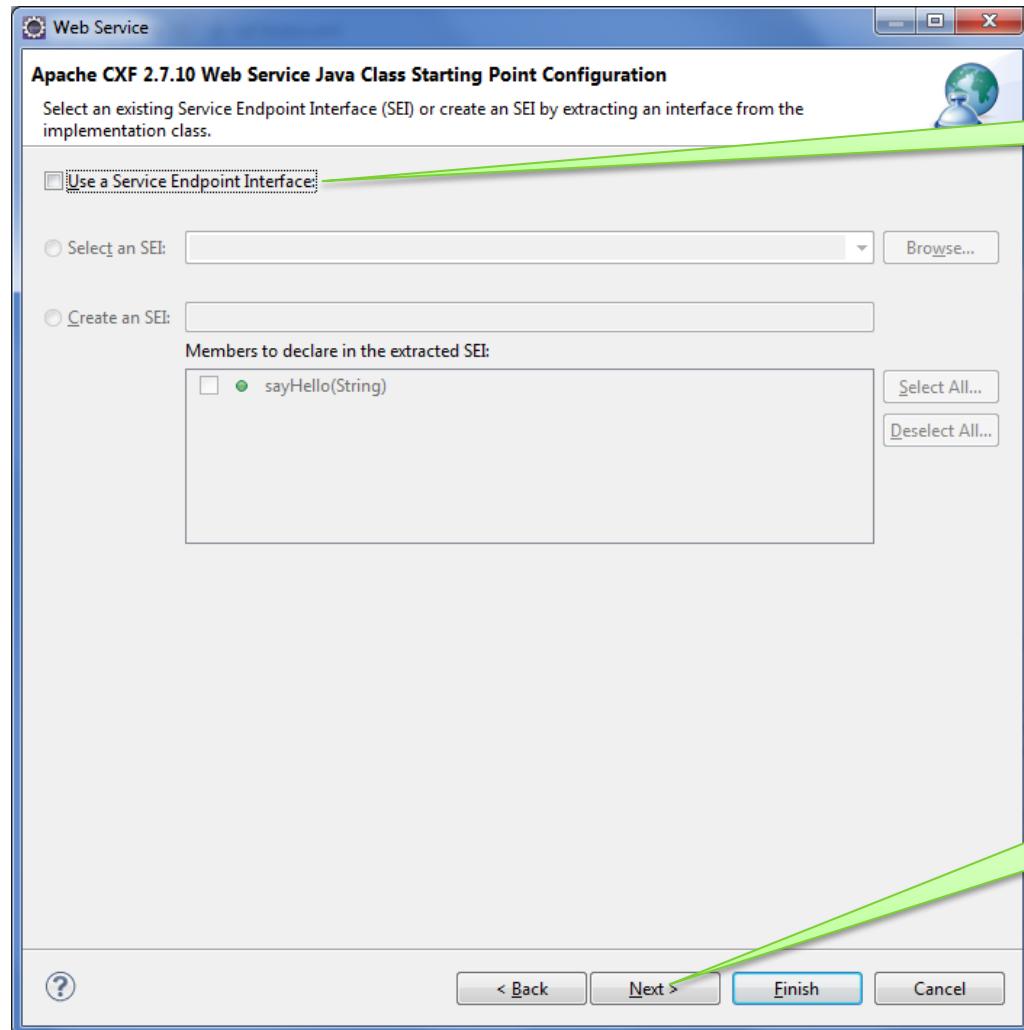
# On utilise l'outil de génération de Web Service



On doit sélectionner CXF !!

On utilise notre serveur Tomcat

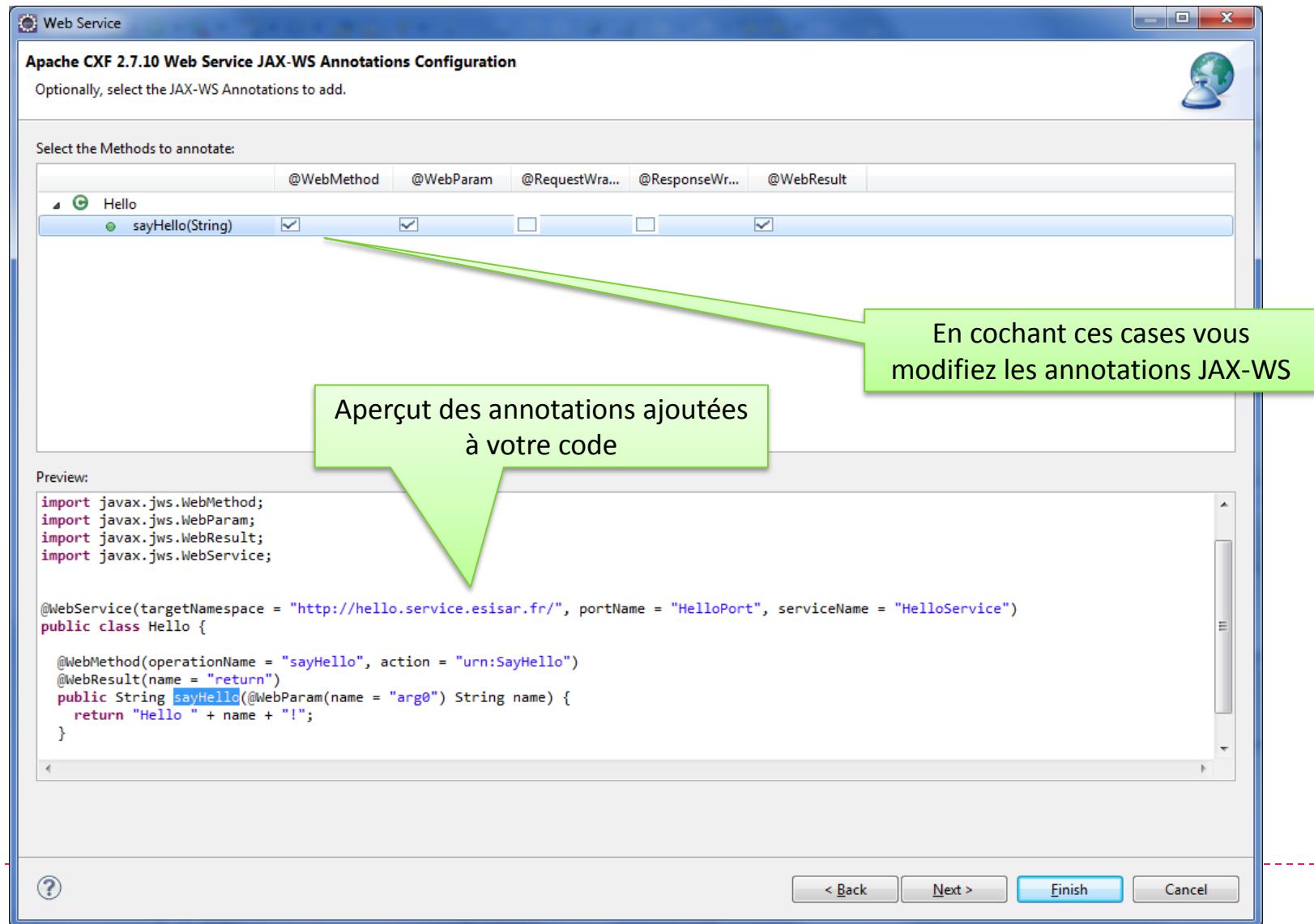
# Configuration des Endpoint



Vous pouvez changer la localisation de votre service

La config par défaut nous conviendra

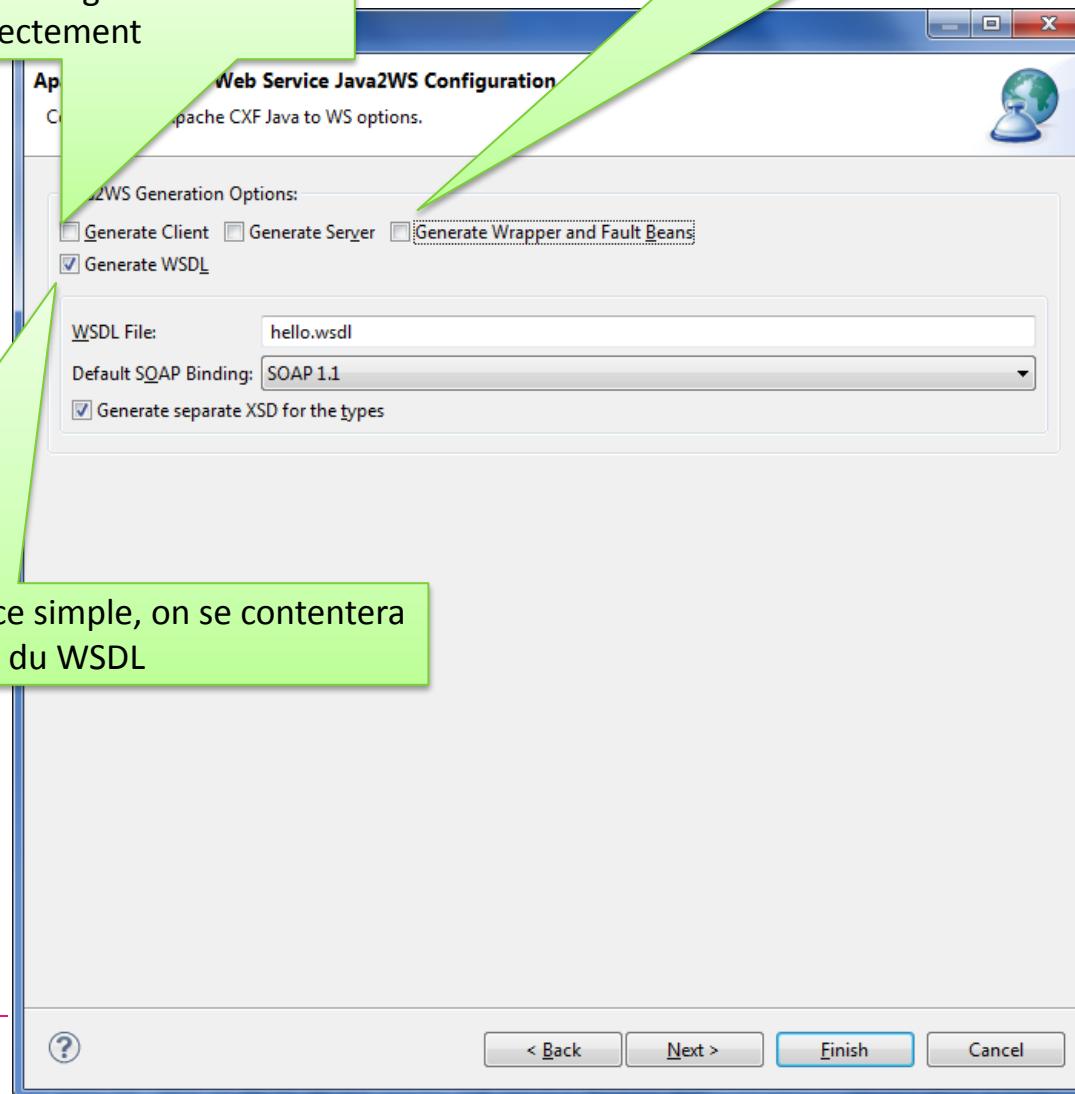
# Génération des annotations



# Génération du WSDL

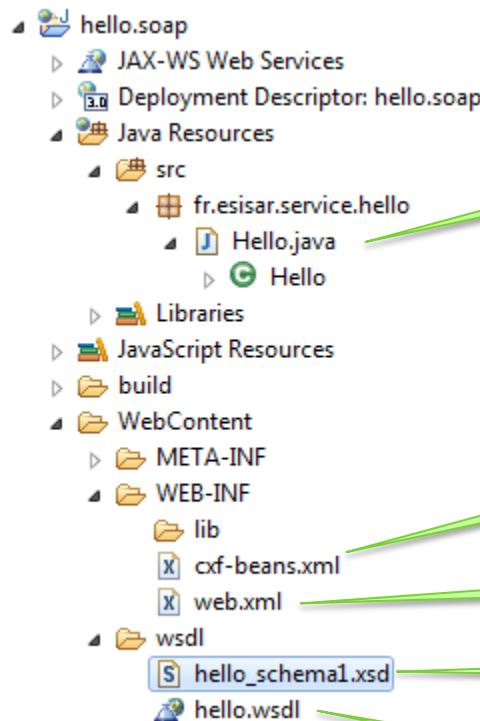
Vous pouvez demander la génération des classes JAXB pour les réponses/requêtes

Vous pouvez demander la génération d'un client directement



Pour notre service simple, on se contentera du WSDL

# Fichiers générés



Votre classe a été modifiée par les annotations si nécessaire

La déclaration de votre service pour CXF

web.xml est modifié pour déclarer une servlet CXF

Déclaration des types utilisés par le WSDL

Votre fichier WSDL

# Interface du service Web annoté

---

```
package fr.esisar.service.hello;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;

@WebService(targetNamespace = "http://hello.service.esisar.fr/", portName = "HelloPort",
serviceName = "HelloService")
public class Hello {

    @WebMethod(operationName = "sayHello")
    @WebResult(name = "helloGreeting")
    public String sayHello(@WebParam(name = "name") String name) {
        return "Hello " + name + "!";
    }

}
```

# Les annotations essentielles

---

- ▶ **@WebService**
  - ▶ permet de spécifier que votre classe est un service
    - ▶ la version courte @WebService est souvent suffisante
- ▶ **@WebMethod**
  - ▶ permet de spécifier les méthodes si vous voulez changer le nom associé dans le WSDL
- ▶ **@WebParam**
  - ▶ permet de spécifier les paramètres
- ▶ **@WebResult**
  - ▶ permet de spécifier le nom du retour

# 3. Configurer le service Web

## ▶ WEB-INF/web.xml

```
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <display-name>cxsf</display-name>
    <servlet>
        <description>Apache CXF Endpoint</description>
        <display-name>cxsf</display-name>
        <servlet-name>HelloWS</servlet-name>
        <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>HelloWS</servlet-name>
        <url-pattern>/services/*</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>60</session-timeout>
    </session-config>
</web-app>
```

### 3. Configurer le service Web

#### ▶ WEB-INF/cxf-servlet.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"
       xmlns:soap="http://cxf.apache.org/bindings/soap"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/bindings/soap
http://cxf.apache.org/schemas/configuration/soap.xsd
http://cxf.apache.org/jaxws
http://cxf.apache.org/schemas/jaxws.xsd">

    <jaxws:server id="jaxwsService"
serviceClass="fr.esisar.service.hello.HelloImpl" address="/HelloWS">
        <jaxws:serviceBean>
            <bean class="fr.esisar.service.hello.HelloImpl" />
        </jaxws:serviceBean>
    </jaxws:server>
</beans>
```

## 5. Déployer dans Tomcat

---

- ▶ Soit par l'interface Web de Tomcat :

The screenshot shows a web-based deployment interface for Tomcat. At the top, a yellow header bar contains the text "Fichier WAR à déployer". Below this, the main form area has a label "Choisir le fichier WAR à téléverser" followed by a "Parcourir..." button and a message "Aucun fichier sélectionné.". At the bottom of the form is a "Deployer" button.

- ▶ Soit manuellement :
  - ▶ En copiant le fichier .war dans le répertoire webapps de Tomcat

## 4. Packager le service Web

- ▶ Créer un fichier une archive *hello.war* qui contient :
  - ▶ Les classes compilées (.class)
  - ▶ Les fichiers de configuration
  - ▶ Les bibliothèques fournies par CXF :

 aopalliance-1.0.jar  
 asm-3.3.1.jar  
 commons-logging-1.1.1.jar  
 cxf-api-2.7.7.jar  
 cxf-rt-bindings-soap-2.7.7.jar  
 cxf-rt-bindings-xml-2.7.7.jar  
 cxf-rt-core-2.7.7.jar  
 cxf-rt-databinding-jaxb-2.7.7.jar  
 cxf-rt-frontend-jaxws-2.7.7.jar  
 cxf-rt-frontend-simple-2.7.7.jar  
 cxf-rt-transports-http-2.7.7.jar  
 cxf-rt-ws-addr-2.7.7.jar  
 cxf-rt-ws-policy-2.7.7.jar  
 geronimo-javamail\_1.4\_spec-1.7.1.jar  
 jaxb-impl-2.2.6.jar  
 neethi-3.0.2.jar

 neethi-3.0.2.jar  
 slf4j-api-1.7.5.jar  
 slf4j-jdk14-1.7.5.jar  
 spring-aop-3.0.7.RELEASE.jar  
 spring-asm-3.0.7.RELEASE.jar  
 spring-beans-3.0.7.RELEASE.jar  
 spring-context-3.0.7.RELEASE.jar  
 spring-core-3.0.7.RELEASE.jar  
 spring-expression-3.0.7.RELEASE.jar  
 spring-web-3.0.7.RELEASE.jar  
 stax2-api-3.1.1.jar  
 woodstox-core-asl-4.2.0.jar  
 wsdl4j-1.6.3.jar  
 xml-resolver-1.2.jar  
 xmlschema-core-2.0.3.jar

# La liste des services ....

- ▶ A l'adresse <http://localhost:8080/nom-du-war/services/>

The screenshot shows a web-based service list interface titled "CXF - Service list". The URL in the address bar is "localhost:8080/soap/services/". The page displays "Available SOAP services" and "Available RESTful services". Under "Available SOAP services", there is one entry for a "Hello" service, which includes its endpoint address, WSDL location, and target namespace.

Service	Details
Hello	Endpoint address: <a href="http://localhost:8080/soap/services/HelloPort">http://localhost:8080/soap/services/HelloPort</a> WSDL : <a href="http://hello.service.esisar.fr/">http://hello.service.esisar.fr/</a> HelloWorld Target namespace: <a href="http://hello.service.esisar.fr/">http://hello.service.esisar.fr/</a>

Available RESTful services:

# 1. Récupérer le fichier WSDL

▶ <http://localhost:8080/soap/services>HelloPort?wsdl>

```
-<wsdl:definitions name="HelloImplService" targetNamespace="http://hello.service.esisar.fr/">
  -<wsdl:types>
    -<xs:schema elementFormDefault="unqualified" targetNamespace="http://hello.service.esisar.fr/" version="1.0">
      <xs:element name="sayHello" type="tns:sayHello"/>
      <xs:element name="sayHelloResponse" type="tns:sayHelloResponse"/>
    -<xs:complexType name="sayHello">
      -<xs:sequence>
        <xs:element minOccurs="0" name="arg0" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
    -<xs:complexType name="sayHelloResponse">
      -<xs:sequence>
        <xs:element minOccurs="0" name="return" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:schema>
</wsdl:types>
-<wsdl:message name="sayHelloResponse">
  <wsdl:part element="tns:sayHelloResponse" name="parameters"></wsdl:part>
</wsdl:message>
-<wsdl:message name="sayHello">
  <wsdl:part element="tns:sayHello" name="parameters"></wsdl:part>
</wsdl:message>
-<wsdl:portType name="HelloInterface">
  -<wsdl:operation name="sayHello">
    <wsdl:input message="tns:sayHello" name="sayHello"></wsdl:input>
    <wsdl:output message="tns:sayHelloResponse" name="sayHelloResponse"></wsdl:output>
  </wsdl:operation>
</wsdl:portType>
-<wsdl:binding name="HelloImplServiceSoapBinding" type="tns:HelloInterface">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
```



# Interroger le service

# Une interface qui décrit le service

```
package fr.esisar.service.hello;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;

/**
 * This class was generated by Apache CXF 2.7.10 2014-03-13T12:37:42.301+01:00
 * Generated source version: 2.7.10
 *
 */

@WebService
public interface Hello_PortType {

    @WebMethod(operationName = "sayHello")
    @WebResult(name = "helloGreeting")
    public java.lang.String sayHello(
        @WebParam(name = "name") java.lang.String name);
}
```

# Un client du service

```
public static void main(String[] args) throws Exception {  
    QName serviceName = new QName("http://hello.service.esisar.fr/",  
        "HelloWorld");  
    QName portName = new QName("http://hello.service.esisar.fr/", "HelloPort");  
  
    Service service = Service.create(new URL(  
        "http://localhost:8080/hello.soap/services/HelloPort?wsdl"),  
        serviceName);  
  
    fr.esisar.service.hello.Hello_PortType client = service.getPort(portName,  
        fr.esisar.service.hello.Hello_PortType.class);  
  
    System.out.println(client.sayHello("Toto"));  
}
```

```
mars 13, 2014 12:46:08 PM org.apache.cxf.service.factory.ReflectionServiceFactoryBean buildServiceFromClass  
Infos: Creating Service {http://hello.service.esisar.fr/}HelloWorld from class fr.esisar.service.hello.Hello_PortType  
Hello Toto!
```



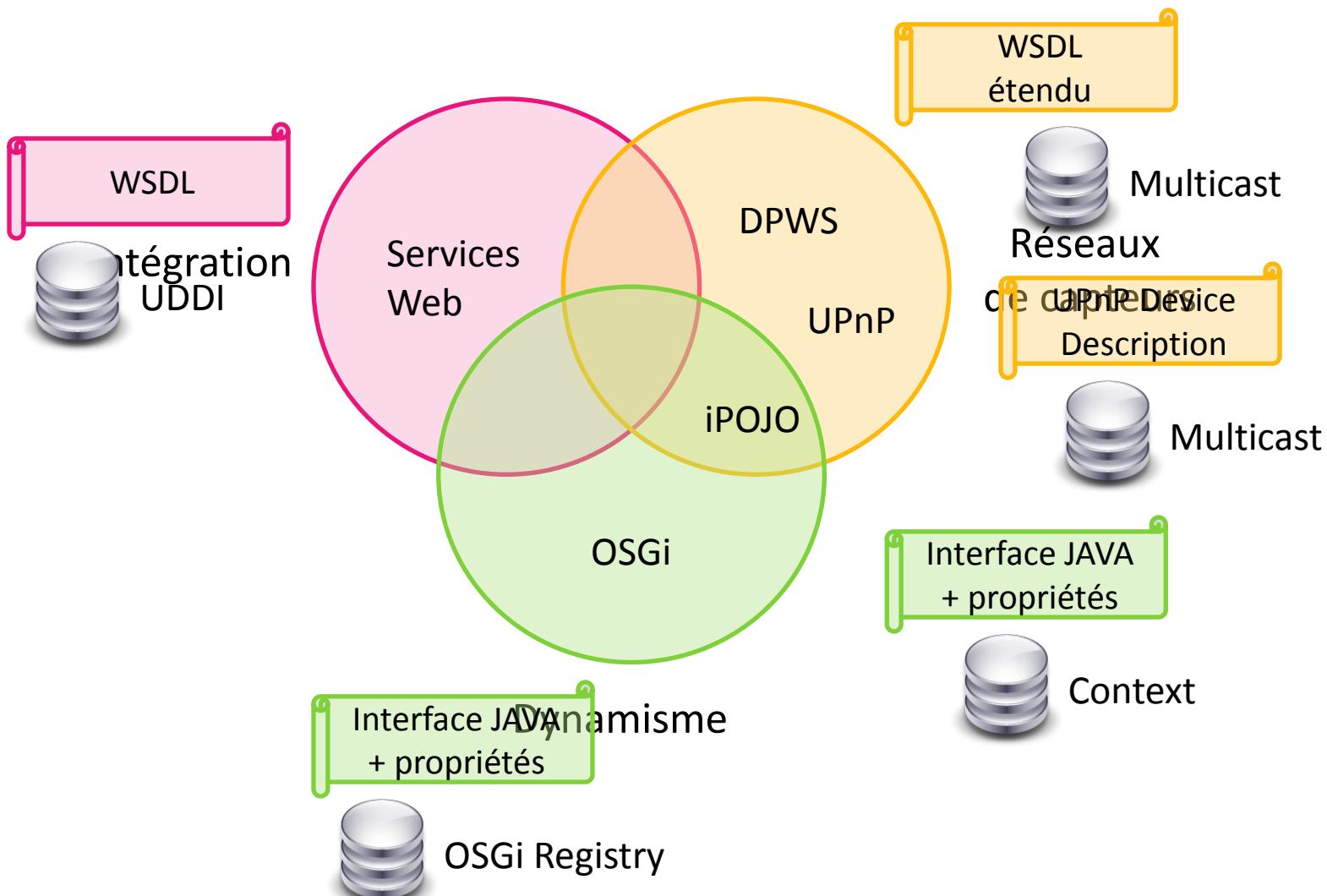
# Synthèse sur SOAP

# Synthèse

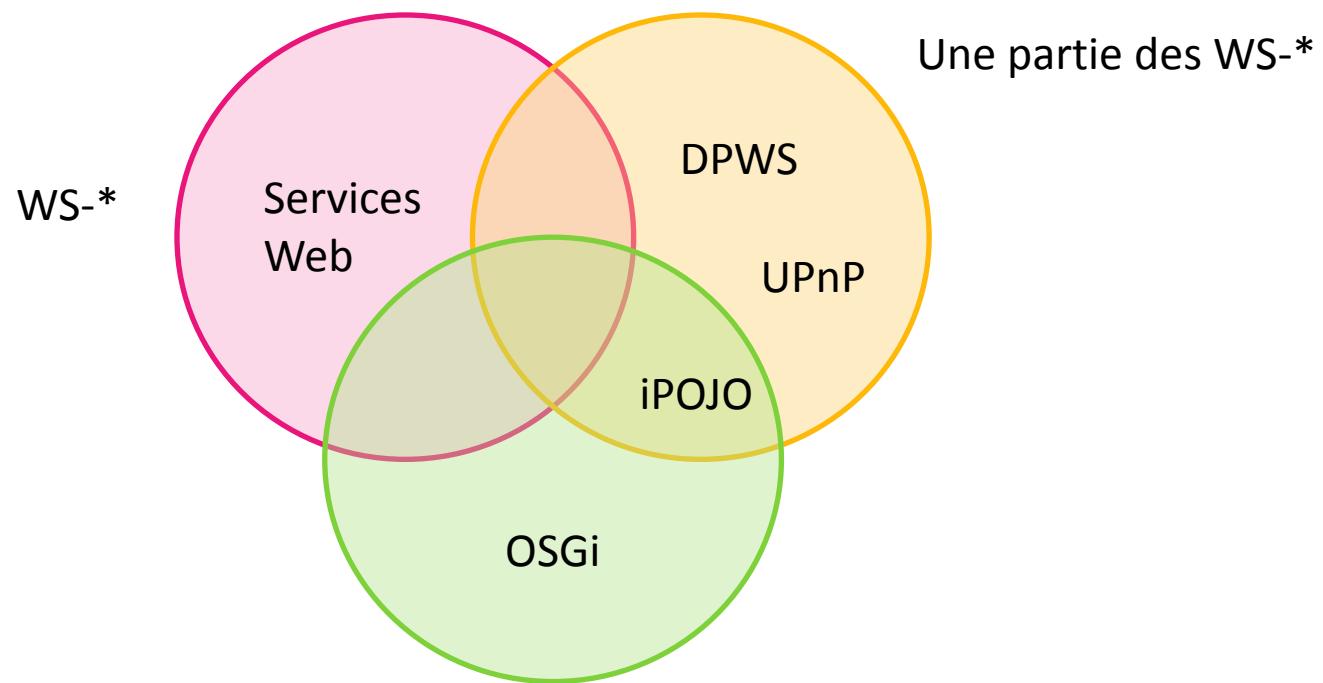
---

- ▶ Service Web (SOAP)
  - ▶ Avantages
    - ▶ Forte popularité
    - ▶ Beaucoup de spécifications définissant les interactions entre fournisseur et consommateur (transactions, sécurité...)
    - ▶ Simplicité d'utilisation
  - ▶ Limites
    - ▶ Beaucoup de spécifications à intégrer
    - ▶ Certaines spécifications et implémentations sont immatures
    - ▶ Pas vraiment faiblement couplé (UDDI non utilisé)
    - ▶ Supporte mal l'évolution dynamique

# Les technologies à services



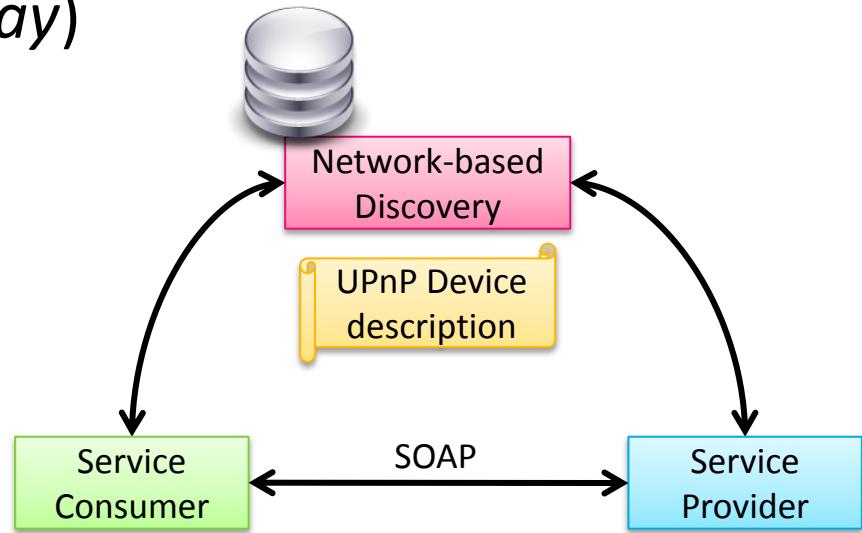
# Les technologies à services – Invocation



# D'autres technologies...

## ▶ UPnP (*Universal Plug and Play*)

- ▶ Service Specification
  - ▶ UPnP Device Description
- ▶ Service Implementation
  - ▶ Many languages
  - ▶ HTTP server required (on the device side)
  - ▶ Generally generation of client proxy
- ▶ Service Discovery
  - ▶ Multicast declaration
- ▶ Service Composition
  - ▶ Client/Server
- ▶ Service Communication
  - ▶ SOAP over HTTP
  - ▶ Synchronous, event-based
- ▶ Service Binding
  - ▶ Direct
  - ▶ Dynamic



# D'autres technologies...

## ▶ DPWS (*Device Profile for Web Services*)

### ▶ Service Specification

- ▶ Extended WSDL

### ▶ Service Implementation

- ▶ Many languages

- ▶ HTTP server required (on the device side)

- ▶ Generally generation of client proxy

### ▶ Service Discovery

- ▶ Multicast declaration

### ▶ Service Composition

- ▶ Client/Server

### ▶ Service Communication

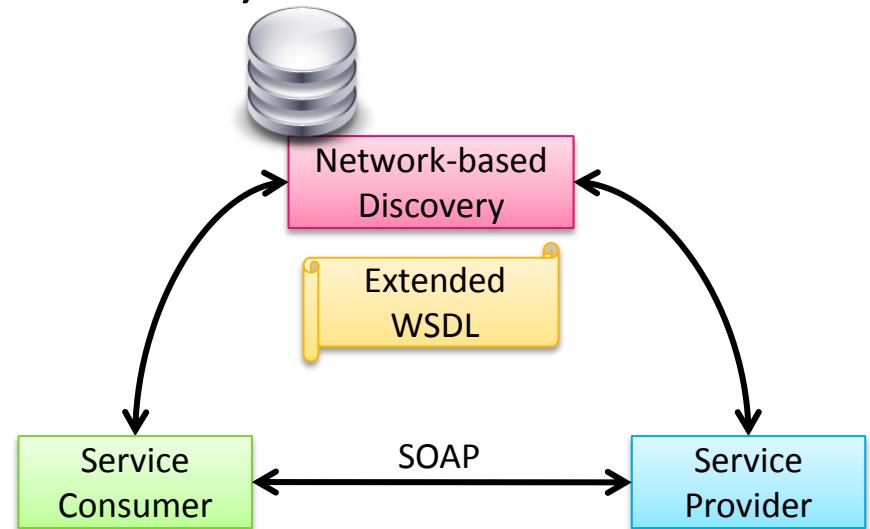
- ▶ SOAP over HTTP

- ▶ Synchronous, event-based

### ▶ Service Binding

- ▶ Direct

- ▶ Dynamic



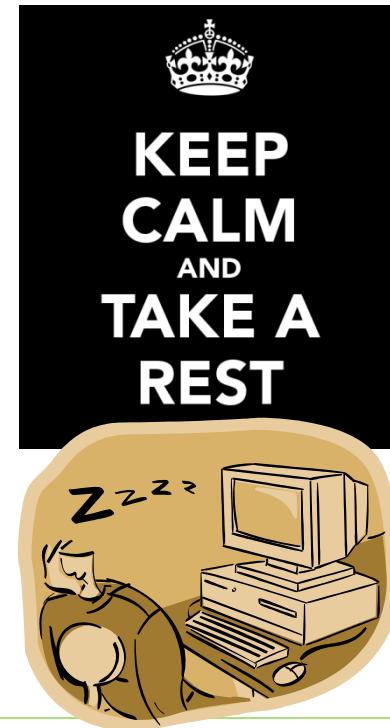
# Bibliographie

---

- ▶ [Papazoglou 2003] Michael P. Papazoglou. Service-Oriented Computing : Concepts, Characteristics and Directions. In *WISE'03 : Proceedings of the Fourth International Conference on Web Information Systems Engineering*, pages 3–12, Washington, DC, USA, 2003. IEEE Computer Society.
- ▶ [Arsanjani 2004] Ali Arsanjani. Service-oriented Modeling and Architecture, November 2004. <http://www.ibm.com/developerworks/webservices/library/ws-soa-design1/>.

# Comparaison

	RMI	RPC	DCOM	CORBA	SOAP
Qui	SUN	SUN/OSF	MicroSoft	OMG	W3C
Plate-formes	Multi	Multi	Win32	Multi	Multi
Langages de Programmation	Java	C, C++, ...	C++, VB, VJ, OPascal, ...	Multi	Multi
Langages de Définition de Service	Java	RPCGEN	ODL	IDL	WSDL
Réseau	TCP, HTTP, IIOP customisable	TCP, UDP	IP/IPX	GIOP, IIOP, Pluggable Transport Layer	HTTP, HTTPR, SMTP, MOM
Firewall	Tunneling HTTP				HTTP
Nommage	RMI, Courtage JINI	IP+Port	IP+Nom	Sv Nom, SvCourtage	IP+Port, URL Courtage UDDI
Transaction	Non	Non	MTS	OTS, XA	Extension applicative dans le header (BTP)
Sécurité	SSL, JAAS	Non	??	SSL	SSL, XKMS, XACML, SAML
Extra	Chargement dynamique des classes			Services Communs Services Sectoriels	

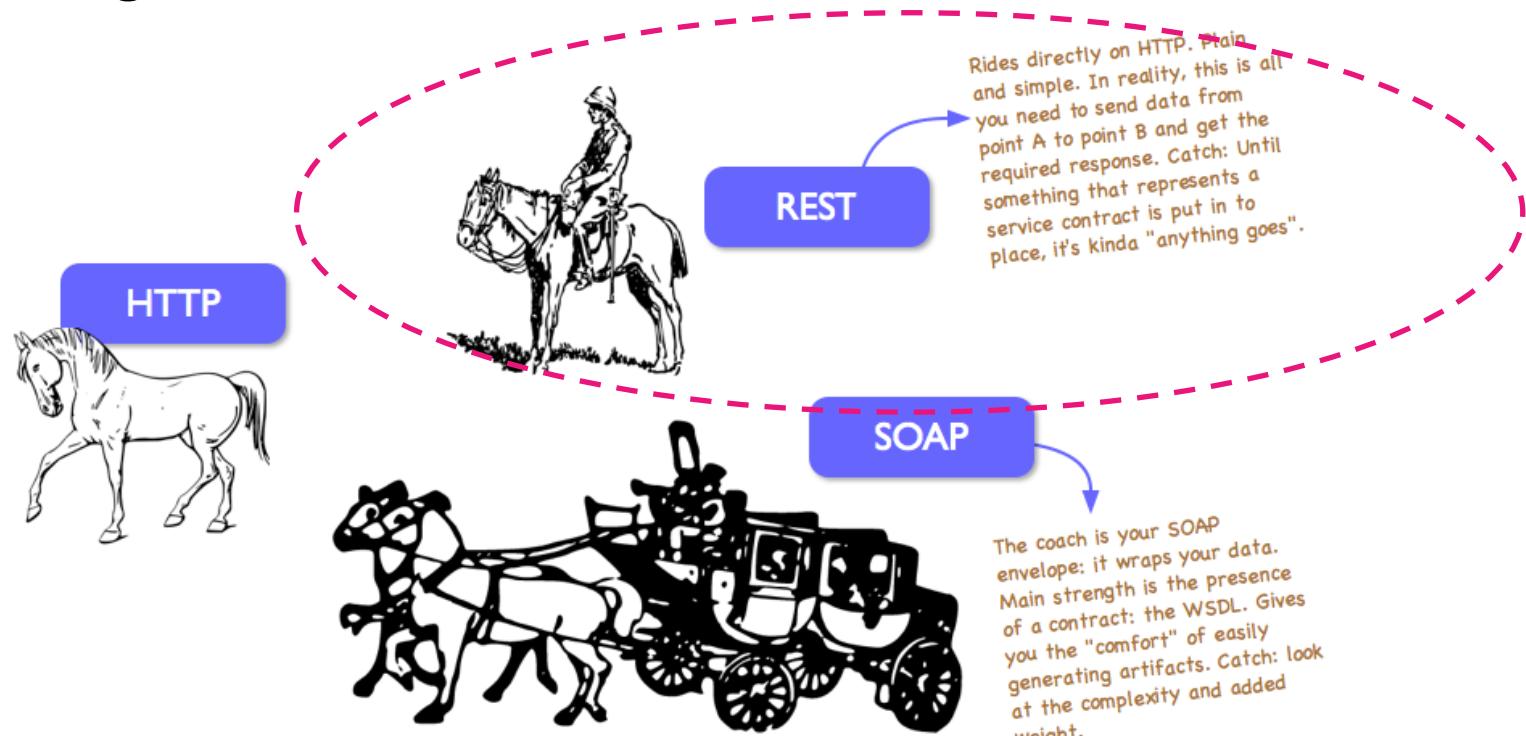


## Services Web - RESTful

Yoann MAUREL

# Les services, plusieurs façons ...

- ▶ Les services SOAP qui ressemblent à RPC
- ▶ **les services REST** (plus récents) qui permettent d'échanger des ressources



source ?

# REST - un **style d'architecture**

---

- ▶ **R**epresentational **S**tate **T**ransfer [Roy Fielding - 2000]
- ▶ Architecture orientée **ressource**/donnée
  - ▶ chaque ressource référencée par un **identifiant global**
- ▶ **Principes:**
  - ▶ **modèle client-serveur**
  - ▶ **sans-état** : les requêtes contiennent toutes les informations suffisantes à leur traitement
  - ▶ **cache possible** : il est possible d'utiliser des serveurs mandataires
  - ▶ **interface uniforme** pour la manipulation des ressources
  - ▶ hiérarchique, code-on-demand

# Qu'est ce qu'une ressource ?

- ▶ Une ressource est une information
    - ▶ une article de journal
    - ▶ une photo
    - ▶ une tâche
    - ▶ la représentation d'une bouteille de vin
  - ▶ 1 ressource = 1 identifiant global (URL)
    - ▶ <http://example.com/livres/isbn-9780345917430>
    - ▶ <http://example.com/livres/isbn-9780007117116>
  - ▶ Non typée et peut être représentée par n'importe quel format d'échange de données (XML, JSON, text, ...)



# JSON ?

---

- ▶ **JavaScript Object Notation**
- ▶ Un objet JSON représente un objet JavaScript
- ▶ Facilite le dialogue avec JavaScript
- ▶ Moins verbeux et lourd que XML
- ▶ Il existe une correspondance JSON <-> XML

# XML <-> JSON

```
<menu id="file" value="File">
  <popup>
    <MenuItem value="New" onclick="CreateNewDoc()" />
    <MenuItem value="Open" onclick="OpenDoc()" />
    <MenuItem value="Close" onclick="CloseDoc()" />
  </popup>
</menu>
```

```
{
  "menu": {
    "id": "file",
    "value": "File",
    "popup": {
      "menuItem": [
        { "value": "New", "onclick": "CreateNewDoc()" },
        { "value": "Open", "onclick": "OpenDoc()" },
        { "value": "Close", "onclick": "CloseDoc()" }
      ]
    }
  }
}
```

[] = tableau

# Web-services dits "RESTful"

---

- ▶ Des services qui respectent l'architecture REST :
  - ▶ ressources **identifiées par des URLs**
  - ▶ communication **sans-état**
    - ▶ le serveur ne garde pas d'informations !
  - ▶ modifient les ressources en utilisant **uniquement les primitives HTTP standard** (GET, PUT, POST, DELETE)
    - ▶ pas d'interface de service à la WSDL
  - ▶ les requêtes et les réponses contiennent la **représentation des ressources** dont le format est précisé par un **type MIME**
    - ▶ application/xml , application/json, ....

# Signification des méthodes HTTP

---

- ▶ **GET** : lire la valeur d'une ressource
- ▶ **PUT** : créer/mettre à jour une ressource
- ▶ **POST** : créer une ressource
- ▶ **DELETE** : effacer une ressource



## Exemple : Service Todo

basé en partie sur

<http://www.vogella.com/tutorials/REST/article.html>

# Exemple de service

- ▶ Gestion d'une liste de tâches (todolist)



- ▶ Ressource = tâche
  - ▶ identifiant de tâche
  - ▶ description
  - ▶ résumé de la tâche
- ▶ Opérations possibles :
  - ▶ avoir la liste des tâches
  - ▶ avoir une tâche particulière
  - ▶ effacer une tâche
  - ▶ ajouter une tâche

# Exemple de service

---

- ▶ Gestion d'une liste de tâches (todolist)
- ▶ Ressource = tâche
  - ▶ identifiant de tâche
  - ▶ description
  - ▶ résumé de la tâche
- ▶ Opérations possibles :

URL	Type de requête	Description
/todolist	GET	avoir liste tâches
/todolist/{id}	GET	avoir tâche ID
/todolist/{id}	DEL	effacer tâche ID
/todolist	PUT	ajoute/maj tâche

# Exemple de représentations

- ▶ Pour une requête GET à <http://example.com/todolist/23>
- ▶ Exemple de représentation en XML :

```
<?xml version="1.0" encoding="UTF-8"?>
<todo>
<id>23</id>
<summary>Courses</summary>
<description>Prendre Lait, oeufs, whisky</description>
</todo>
```

- ▶ Exemple en TEXTE:

```
23;"Courses";"Prendre Lait, oeufs, whisky"
```

- ▶ Exemple en JSON

```
{"id":23,"summary": "Courses"; "description": "Prendre ...."}
```

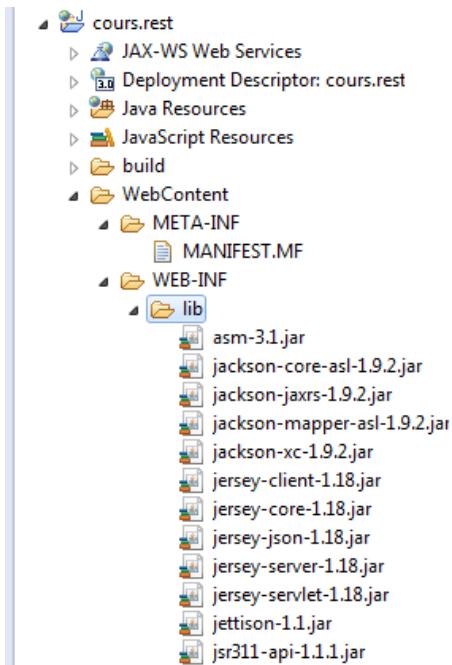
# JAX-RS

---

- ▶ API standard de JAVA
- ▶ Pendant de JAX-WS pour les services RESTful
- ▶ Basée sur des annotations :
  - ▶ ressources = POJOs + annotations
- ▶ Plusieurs implémentations :
  - ▶ **Jersey => on utilisera la version 1**
  - ▶ CXF
  - ▶ RESTEasy
  - ▶ RESTLet

# Création du projet

- ▶ Créer un Dynamic Web Projet
- ▶ Mettre l'ensemble des librairies de Jersey 1.x dans le dossier WebContent/WEB-INF/lib



# Configuration de la servlet Jersey

- ▶ Il faut modifier le fichier web.xml

```
<servlet>
  <servlet-name>Jersey REST Service</servlet-name>
  <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
  <init-param>
    <param-name>com.sun.jersey.config.property.packages</param-name>
    <param-value>mri.todo.resources</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Jersey REST Service</servlet-name>
  <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

Liste des packages dans lesquels se trouvent des services RESTful

Chemin d'accès aux ressources. Les ressources seront accessibles via <http://.../projet/rest/ressources>

# Application TODO

---

- ▶ On imagine qu'on a déjà une application avec deux classes :
  - ▶ une classe Todo pour représenter une tâche
  - ▶ une classe singleton TodoList instanciée une fois qui permet de sauvegarder la liste des tâches.
    - ▶ idéalement utilise une base de données ou fait de la persistance
- ▶ Nous souhaitons rendre accessibles les tâches via un service RESTful

# La classe Todo avec JAXB

```
@XmlElement  
public class Todo {  
    private String id;  
    private String summary;  
    private String description;  
  
    public Todo(){  
    }  
    public Todo (String id, String summary){  
        this.id = id;  
        this.summary = summary;  
    }  
    public String getId() {  
        return id;  
    }  
    public void setId(String id) {  
        this.id = id;  
    }  
    //....  
}
```

On utilise JAXB pour permettre la conversion de Java vers JSON/XML et réciproquement

# La classe TodoList

```
public class TodoList {  
  
    private static final TodoList INSTANCE = new TodoList();  
  
    private Map<String, Todo> todos = new HashMap<String, Todo>();  
  
    public static final TodoList getInstance() {  
        return INSTANCE;  
    }  
  
    public synchronized Set<Todo> getTodos() {  
        return new HashSet(todos.values());  
    }  
  
    public synchronized void newTodo(Todo todo) {  
        todos.put(todo.getId(), todo);  
    }  
  
    public synchronized Todo getTodo(String id) {  
        return todos.get(id);  
    }  
  
    public synchronized void delTodo(String id) {  
        todos.remove(id);  
    }  
}
```

On accède à l'instance singleton via  
TodoList.getInstance()

Servira aussi à la mise à jour

# Application TODO

---

- ▶ Nous souhaitons rendre accessibles les tâches via un service RESTful
- ▶ On va coder deux ressources :
  - ▶ TodosRessource pour avoir accès à la liste des tâches
    - ▶ accessible via <http://..../todolist>
  - ▶ TodoRessource pour avoir accès à un Todo particulier
    - ▶ accessible via <http://..../todolist/{id}>
    - ▶ TodoListRessource va déléguer vers cette classe

# Les ressources sont identifiées par des URL

---

- ▶ JAX-RS associe chaque ressource à une classe Java
  - ▶ un POJO (Plain Old Java Object) = classe toute simple
- ▶ L'identifiant de la ressource est fournie par l'annotation **@PATH(url)**
  - ▶ l'url est donnée de **façon relative (/toto)**
  - ▶ il peut y'avoir une **part variable** (/toto/{id}) indique que {id} est variable
    - ▶ partie variable récupérée en paramètre lors du traitement via annotation **@PathParam**
    - ▶ on peut utiliser des patterns : **/toto/{id}-{name}.do**
- ▶ **@PATH** se met au niveau des méthodes et des classes

# Annotation @Path sur la classe

```
@Path("/todolist")
```

```
public class TodosResource {
```

```
//...
```

```
    @GET
```

```
    @Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
```

```
    public List<Todo> getTodosBrowser() {
```

```
        List<Todo> todos = new ArrayList<Todo>();
```

```
        todos.addAll(TodoList.getInstance().getTodos());
```

```
        return todos;
```

```
}
```

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

La todolist est accessible via /todolist

Les méthodes qui ne sont pas préfixée par @Path sont appelée quand on accède à /todolist

# @Path sur la méthode et @PathParam

**@Path("/todolist")**

```
public class TodosResource {
```

La todolist est accessible via /todolist

```
//...
```

Permet d'accéder à une ressource particulière via /todolist/{todo} où {todo} est une variable

@PathParam permet de préciser que le paramètre id est associé à {todo}

**@Path("{todo}")**

```
public TodoResource getTodo(@PathParam("todo") String id) {  
    return new TodoResource(uriInfo, request, id);  
}
```

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

# Primitive GET, POST, PUT, DEL

- ▶ **@POST, @GET, @PUT, @DELETE**
- ▶ Permettent de préciser la primitive HTTP utilisée

## @GET

```
@Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
public List<Todo> getTodosBrowser() {
    List<Todo> todos = new ArrayList<Todo>();
    todos.addAll(TodoList.getInstance().getTodos());
    return todos;
}
```

# Format de représentation

- ▶ Le format (MIME) consommé et produit est défini par les annotations **@Consumes** et **@Produces**
- ▶ MediaType.APPLICATION\_XML => "application/xml"

```
@GET  
@Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })  
public List<Todo> getTodosBrowser() {  
    List<Todo> todos = new ArrayList<Todo>();  
    todos.addAll(TodoList.getInstance().getTodos());  
    return todos;  
}
```

```
@PUT  
@Consumes({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })  
public Response putTodo(JAXBElement<Todo> todoJaxb) {  
    Todo todo = todoJaxb.getValue();  
    TodoList.getInstance().newTodo(todo);  
    return Response.created(uriInfo.getAbsolutePath()).build();  
}
```

# Délégation vers une autre classe Ressource

- ▶ On parle de sous-ressource
- ▶ La méthode possède un Path avec un paramètre
- ▶ Le type de retour est une classe Ressource (annotée)
- ▶ Le type de requête (GET, POST, ...) n'est pas précisé

```
@Path("/todolist")
public class TodosResource {

    //Pas de @GET, @PUT, @POST, ....
    @Path("{todo}")
    public TodoResource getTodo(@PathParam("todo") String id) {
        return new TodoResource(uriInfo, request, id);
    }

    //....
}
```

## Annotation **@Context**

---

- ▶ Permet d'injecter des objets liés au contexte de l'application
- ▶ Les objets supportés sont de la classe :
  - ▶ UriInfo : pour obtenir des informations sur l'URL
  - ▶ Request : pour obtenir des informations sur la requête
  - ▶ HttpHeaders : pour obtenir des informations sur l'en-tête
  - ▶ SecurityContext : pour obtenir des informations sur la sécurité

# Exemple d'utilisation

```
@Path("/todolist")
public class TodosResource {
```

**@Context**

**UriInfo** uriInfo;

infos sur l'URL

**@Context**

**Request** request;

infos sur la requête

```
//...
```

à vous de consulter la javadoc ...

# Classe TodosResource

```
@Path("/todolist")
public class TodosResource {

    // e.g. ServletContext, Request, Response, UriInfo
    @Context
    UriInfo uriInfo;
    @Context
    Request request;

    @GET
    @Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
    public List<Todo> getTodos() {
        List<Todo> todos = new ArrayList<Todo>();
        todos.addAll(TodoList.getInstance().getTodos());
        return todos;
    }

    @Path("{todo}")
    public TodoResource getTodo(@PathParam("todo") String id) {
        return new TodoResource(uriInfo, request, id);
    }
}
```

Permet d'avoir la liste des tâches

On utilise notre objet singleton TodoList qui stocke les tâches

Délègue à TodoResource les opération sur une ressource

On retourne un objet TodoResource

# Essayons

- ▶ Run On Server
- ▶ La ressource liste est accessible à l'adresse :
  - ▶ <http://localhost:8080/cours.rest/rest/todolist>



The screenshot shows a web browser window with the address bar containing the URL `http://localhost:8080/cours.rest/rest/todolist`. The page content displays an XML document representing a list of todos:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
- <todos>
  - <todo>
    <description>du whisky</description>
    <id>1</id>
    <summary>Boire</summary>
  </todo>
  - <todo>
    <description>beaucoup</description>
    <id>2</id>
    <summary>Dormir</summary>
  </todo>
</todos>
```

Regardons maintenant la classe TodoResource ....

## Type personnalisés en valeur de retour

- ▶ On utilise JAXB pour permettre d'utilisation des types non fournis par Java
  - ▶ gère JSON et XML automatiquement en fonction de Consumes/Produces
- ▶ On peut directement retourner un type **Todo** (si annoté via JAXB)

```
// Application integration
@GetMapping
@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
public Todo getTodo() {
    Todo todo = TodoList.getInstance().getTodo(id);
    if (todo == null)
        throw new RuntimeException("Get: L'" + id + " n'a pas été trouvé");
    return todo;
}
```

# Type personnalisés en paramètres

- ▶ Il faut utiliser **JAXBElement<Classe>** pour les paramètres
  - ▶ on récupère l'objet avec **getValue()**

```
@PUT  
@Consumes({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })  
public Response putTodo(JAXBElement<Todo> todoJaxb) {  
    Todo todo = todoJaxb.getValue();  
    TodoList.getInstance().newTodo(todo);  
    //....  
}
```

## Réponse : classe javax.ws.rs.core.Response

---

- ▶ A chaque réponse, un code statut est envoyé aux clients dans le HTTP. Exemple :
  - ▶ 200 par exemple en cas de réussite
  - ▶ 204 indique que le serveur a traité la requête mais qu'il n'y a pas de contenu dans la réponse
  - ▶ les classiques erreurs : 404, 405, 406, ...
- ▶ Pour gérer les réponse on utilise la classe Response avec le patron Builder. Exemple :
  - ▶ Response.noContent().build() pour renvoyer un 204 si la ressource n'existe pas
  - ▶ Response.created(uriInfo.getAbsolutePath()).build() retourne l'URL de la ressource nouvellement créée

# Exemple pour la méthode PUT

- ▶ On ne renvoie **pas de réponse** si l'objet existe déjà sinon on renvoie l'**URL de l'objet créé**

```
@PUT  
@Consumes({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })  
public Response putTodo(JAXBElement<Todo> todoJaxb) {  
  
    Todo todo = todoJaxb.getValue();  
  
    // vérifie que l'élément n'existe pas déjà  
    if (TodoList.getInstance().getTodo(todo.getId()) != null) {  
        return Response.noContent().build();  
    }  
  
    TodoList.getInstance().newTodo(todo);  
    return Response.created(uriInfo.getAbsolutePath()).build();  
}
```

# Code de TodoResource

```
public class TodoResource {  
    @Context  
    UriInfo uriInfo;  
    @Context  
    Request request;  
    String id;  
    public TodoResource(UriInfo uriInfo, Request request, String id) {  
        this.uriInfo = uriInfo;  
        this.request = request;  
        this.id = id;  
    }  
  
    @GET  
    @Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })  
    public Todo getTodo() {  
        Todo todo = TodoList.getInstance().getTodo(id);  
        if (todo == null)  
            throw new RuntimeException("Get: L'" + id + " n'a pas été trouvé");  
        return todo;  
    }  
  
    @DELETE  
    public void deleteTodo() {  
        TodoList.getInstance().delTodo(id);  
    }  
  
    @PUT  
    @Consumes({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })  
    public Response putTodo(JAXBElement<Todo> todoJaxb) {  
        //....  
    }
```

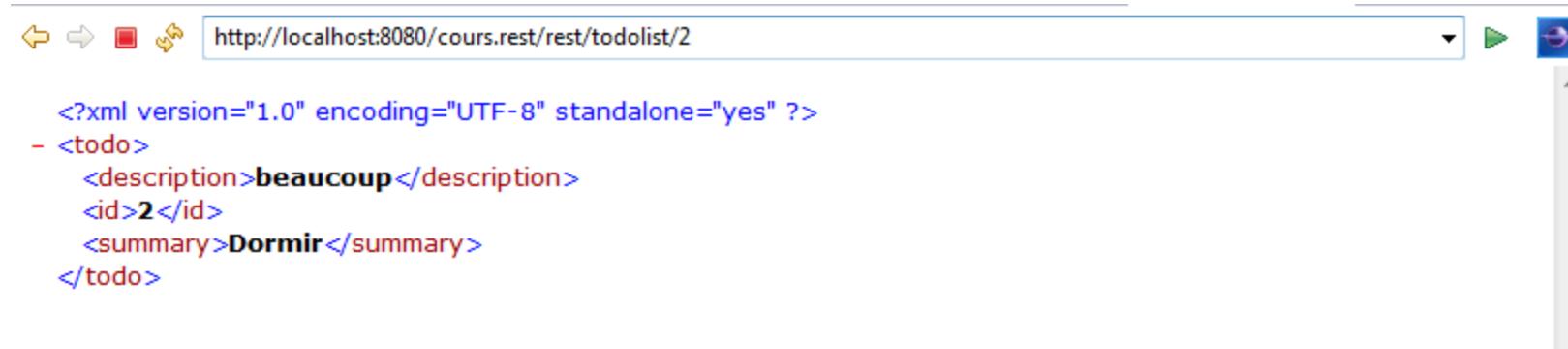
Obtention d'une tâche

Effacement d'une tâche

Ajout d'une tâche (cf code slide précédent)

# Test de GET sur une ressource

- ▶ La tâche d'identifiant 2 est accessible à l'adresse :
  - ▶ <http://localhost:8080/cours.rest/rest/todolist/2>



A screenshot of a web browser window. The address bar contains the URL `http://localhost:8080/cours.rest/rest/todolist/2`. The page content area displays the following XML response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
- <todo>
  <description>beaucoup</description>
  <id>2</id>
  <summary>Dormir</summary>
</todo>
```



## Client du service Todo

# API Client

---

- ▶ JAX-RS ne définit pas d'API standard pour les clients Java
  - ▶ dépendant de l'implémentation



- ▶ Jersey en propose une
  - ▶ peut interroger n'importe quel service REST
    - ▶ pas seulement les services implémentés en JAVA

# Création d'un Client

- ▶ On crée un client avec une config par défaut :

```
ClientConfig config = new DefaultClientConfig();
Client client = Client.create(config);
```

- ▶ On crée une WebResource

- ▶ Permet de manipuler de faire des requête et obtenir des réponses sur une ressource

URL du service utilisé

```
URI serviceURI = UriBuilder.fromUri("http://localhost:8080/cours.rest").build();
WebResource service = client.resource(serviceURI);
```

Permet de manipuler les ressources à cette URL

# Construction d'une requête

---

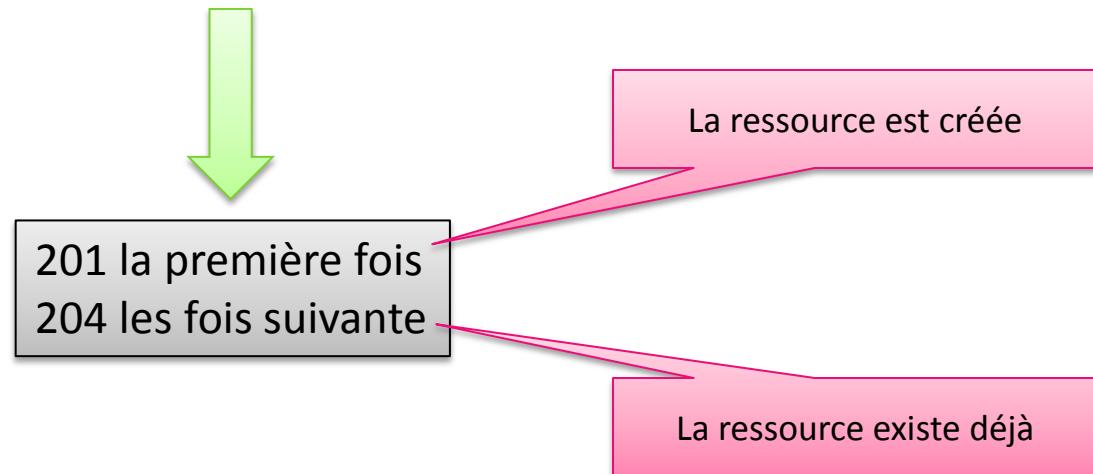
## ▶ Pattern Builder

- ▶ **path permet de construire le chemin**
- ▶ **accept permet de définir le type de retour souhaité**
- ▶ header permet d'ajouter des champs à l'header
- ▶ cookie : gestion des cookies
- ▶ queryParam : paramètres de requête HTTP
- ▶ **get, put, post, delete permet d'envoyer une requête de type GET, PUT, POST, DELETE (terminaison de l'appel)**

```
ClientResponse response = service.path("rest").path("todolist")
    .path(todo.getId()).accept(MediaType.APPLICATION_XML)
    .put(ClientResponse.class, todo);
```

# Exemple d'appel à PUT

```
Todo todo = new Todo("3", "Blabla");
ClientResponse response = service.path("rest").path("todolist")
    .path(todo.getId()).accept(MediaType.APPLICATION_XML)
    .put(ClientResponse.class, todo);
System.out.println(response.getStatus());
```



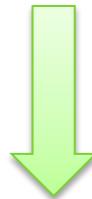
# Exemple d'appel à GET avec réponse JSON

- ▶ On veux l'ensemble des tâches

Le type de retour est une chaîne

```
System.out.println(service.path("rest").path("todolist")
    .accept(MediaType.APPLICATION_JSON).get(String.class));
```

On demande du JSON



```
{"todo":[{"description":"beaucoup","id":"2","summary":"Dormir"}, {"description":"du whisky","id":"1","summary":"Boire"}, {"id":"3","summary":"Blabla"}]}
```

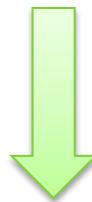
# Exemple d'appel à GET avec réponse XML

- ▶ On veux l'ensemble des tâches

Le type de retour est une chaîne

```
System.out.println(service.path("rest").path("todolist")
    .accept(MediaType.APPLICATION_XML).get(String.class));
```

On demande du XML



```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?><todoes><todo><description>beaucoup</description><id>2</id><summary>Dormir</summary></todo><todo><id>3</id><summary>Blabla</summary></todo><todo><description>du
whisky</description><id>1</id><summary>Boire</summary></todo></todoes>
```

# Obtention d'une ressource particulière

```
Todo tache = service.path("rest").path("todolist/1")
    .accept(MediaType.APPLICATION_XML).get(Todo.class);
```

```
System.out.println(tache.getId() + ";" + tache.getSummary() + ";" +
    + tache.getDescription());
```

On demande la tâche 1

On donne une classe  
annotée en JAXB qui permet  
de représenter l'objet



1;Boire;du whisky

► La classe Todo n'est pas forcément la même que celle définie par le Serveur, il suffit qu'elle soit annotée correctement

# GET : Obtention d'un tableau via JAXB

```
Todo[] todos = (Todo[]) service.path("rest").path("todolist")
    .accept(MediaType.APPLICATION_JSON).get(Todo[].class);
```

```
System.out.println(todos[1].getSummary());
```

L'ordre n'est pas celui des id

On demande l'ensemble des tâches

On indique la classe tableau de Todo



Blabla

On obtient la description de la tâche 3

## DEL sur une ressource

---

- ▶ Comme il n'y a pas de code de retour d'état dans notre ressource, on utilise simplement :

```
Service.path("rest").path("todolist/3").delete();
```



# RPC avec des service RESTful

# RPC avec des services RESTful

- ▶ On peut faire du RPC :
  - ▶ en exposant des ressources avec des PathParam qui correspondent à une méthode
  - ▶ Eventuellement en créant des objets réponses/requêtes pour les requêtes/réponses complexes

```
@XmlRootElement  
public class ReponseOperation {  
  
    private int resultat;  
  
    public int getResultat() {  
        return resultat;  
    }  
  
    public void setResultat(int resultat) {  
        this.resultat = resultat;  
    }  
  
}
```

Exemple d'objet réponse

# Exemple de calculatrice



```
@Path("/calc")
public class ServiceCalculatrice {

    // This can be used to test the integration with the browser
    @POST
    @Path("calculer")
    @Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
    public ReponseOperation calculer(RequeteOperation req) {

        ReponseOperation resultat = new ReponseOperation();

        if (req.getOperation().equals("+")) {
            resultat.setResultat(req.getOp1() + req.getOp2());
        } else if (req.getOperation().equals("-")) {
            resultat.setResultat(req.getOp1() - req.getOp2());
        } else {
            resultat.setResultat(0);
        }
        return resultat;
    }

    // This can be used to test the integration with the browser
    @GET
    @Path("{op1}/{op2}/{operation}")
    @Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
    public ReponseOperation calculer(@PathParam("op1") int op1,
                                    @PathParam("op2") int op2, @PathParam("operation") String operation) {

        ReponseOperation resultat = new ReponseOperation();

        if (operation.equals("+")) {
            resultat.setResultat(op1 + op2);
        } else {
            resultat.setResultat(0);
        }
        return resultat;
    }
}
```



## Appel via JavaScript/JQuery

# En JavaScript et Jquery

- ▶ On utilise `$.ajax(requete)`
- ▶ Exemple d'appel sur un service Calculatrice

```
function demanderLeResultatAuServeur() {  
    $.ajax({  
        type : 'POST', // On indique que l'on envoie des informations  
        contentType : 'application/json', // on envoie une requête au format  
        // JSON  
        url : rootURL + "/calc/calculer", // on indique l'URL du service  
        // calculer  
        dataType : "json", // on veux du JSON en retour  
        data : formToJson(), // on transforme le formulaire en donnée JSON  
        success : function(data, textStatus, jqXHR) {  
            // si réussi  
            // on fait afficher le résultat dans le formulaire  
            afficherLeResultatDuCalcul(data);  
        },  
        error : function(jqXHR, textStatus, errorThrown) { // en cas d'erreur  
            console.log('calculer error: ' + textStatus); // on log un message  
            // dans la console  
        }  
    });  
}
```

Appel d'un callback en cas de succès

Appel d'un callback en cas d'erreur

# On manipule les Objets JSON directement

```
// La méthode afficherLaReponse permet de faire afficher le résultat obtenu
function afficherLeResultatDuCalcul(reponseOperation) {
    // on change la valeur du champ d'id result.
    // comme on reçoit un objet de type ReponseOperation mais au format JSON (un
    // objet javascript), il suffit de prendre la valeur du champ résultat.
    // on le fait de la même façon que pour Java.
    $('#result').val(reponseOperation.resultat);
}
```



On change la valeur de la balise input HTML d'id result

# Réaction à des évènements

- ▶ On peut demander la mise à jour sur des évènements :

```
// on demande à faire le calcul quand on appuie sur le bouton = d'id btnCalc
$('#btnCalc').click(function() {
    demanderLeResultatAuServeur();
    return false;
});

// on demande à faire le calcul quand on change l'opération
$("#operation").change(function() {
    demanderLeResultatAuServeur();
});

// on demande à faire le calcul à chaque entrée sur le clavier
$("#calculer").keypress(function() {
    demanderLeResultatAuServeur();
});
```