Web Services

Définition

- **Les Web Services sont des services offerts via le web.**

- **Par exemple, un client demande le prix d’un article en envoyant un message sur le Web. Ce message contient la référence de l’article. Le Web Service va recevoir la référence, effectuer le traitement du service et renvoyer le prix au client via un autre message.**

Principes

Pourquoi un nouveau middleware ?

Limitations des middleware étudiés (objet, composant)

Passage à large échelle : Web

- Protocoles hétérogènes
  - IIOP, RMI, DCOM
  - Pare-feu (ex en 210A : un objet CORBA sous XP)
- Pas d'ouverture des services
  - Notion de moteur de recherche inexistante
- Trop de contraintes sur le client !
  - Doit posséder les souches
  - Difficulté de construire dynamiquement

Limitations des middleware

Inconvénients Intrinsèques

- Complexité
  - CORBA : IDL, MapEcho, …
  - EJB : Container, JNDI, …
- Pérennité : remise en question
  - CORBA, EJB, .Net, …
- Prix
  - Plates-formes
  - Compétences

Solutions existantes

- Modification du Protocole
  - RMI / IIOP
- Passerelles
  - CORBA vers DCOM
- Portage d'applications existantes difficile
- Solutions non standards

Extension vers les services

- **WSOA : Web Services Oriented Architecture**

- **Deux préoccupations**

  - **ÉCHELLE : Augmenter la productivité des entreprises à travers des partenariats => nécessité d 'ouvrir un parc applicatif à l'extérieur sous forme de services offerts**
    - ***Granularité variable : fonctions, composants, applications, processus métier***

  - **INTEROPÉRABILITÉ des applications (intra- ou inter-entreprises)**
    - ***Comment faire fonctionner un applicatif (ou un composant) fondé sur une technologie CORBA avec un autre fondé sur .Net ?***

Approche Envisagée

- **Un nouveau Protocole : SOAP**
  - **Basé sur XML**
    - ***Portabilité, Hétérogénéité***
  - **Porté sur des protocoles large échelle existants**
    - ***HTTP, SMTP, …***
- **Paradigme orienté service : WSDL**
  - **Définition de services offerts (en XML)**
- **Découverte automatique des services (dynamicité) : UDDI**
  - **Référentiel de Web Service (Pages Jaunes, Blanches)**

Service Web : définitions

- **Les Services Web sont des « applications modulaires basées sur Internet qui exécutent des tâches spécifiques et qui respectent un format spécifique » *Mark Colan (IBM)***

- **Un service Web est un composant logiciel représentant une fonction applicative (ou un service applicatif)**
  - **Accessible depuis une autre application (un client, un serveur ou un autre service Web) à travers le réseau Internet en utilisant les protocoles de transports disponibles**
  - **Implanté comme une application autonome ou comme un ensemble d'applications**

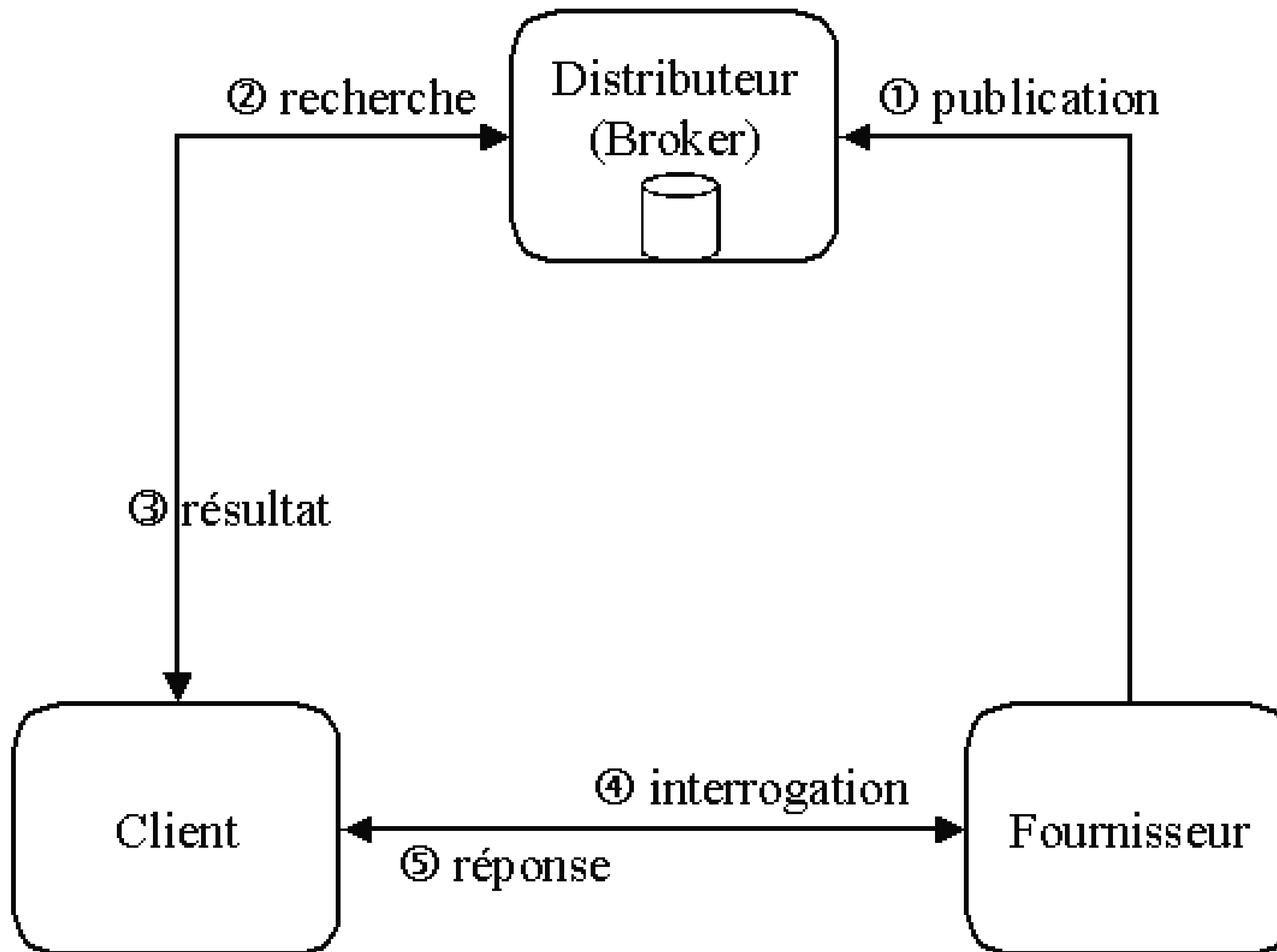
- **Technologie permettant à des applications de dialoguer à distance via Internet, indépendamment des plates-formes et des langages sur lesquelles elles reposent**

Service Web : comment ?

- Les services Web s'appuient sur un ensemble de protocoles standardisant les modes d'invocation mutuels de composants applicatifs
- XML : format utilisé pour décrire et échanger les données
- SOAP : protocole d’invocation d’un service Web
- WSDL : description XML de l'interface publique d’un service Web
- UDDI : centralisation des descriptions de services Web dans un référentiel commun utilisé pour rechercher un service ou pour publier un service

Standards du W3C (World Wide Web Consortium)

Service Web : mise en œuvre



XML

Le langage de définition du format des documents

XML

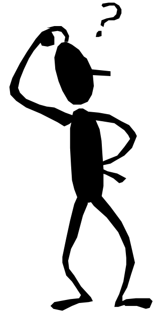
- **eXtended Markup Language**
- **Un document XML est constitué de deux entités distinctes :**
  - **Le fond (*le contenu*)**
  - **La forme (la structure), identifiée par des balises qui encadrent le contenu typé (les données)**

```
<livre>
    <titre> le super livre </titre>
    <chapitre>
        <numero> 1 </numero>
        <titre> titre du chapitre 1 </titre>
        <paragraphe> blabla blabla </paragraphe>
    </chapitre>
    <chapitre>
        …
    </chapitre>

</livre>
```

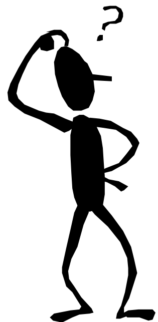
Le contenu est encadré par la balise (ouvrante et fermante).
Il est typé.

Principes

- Écrire un doc XML = utiliser des balises dans un certain ordre et des types de contenu
- D'où viennent les balises ?
  - Ensemble non-prédéfini et non fini de balises (pas comme html)
  - Chaque utilisateur peut définir ses propres balises



- Définir les balises, leur ordre et le typage de leur contenu = définir la grammaire du doc XML
  - Exemple du livre
- Grammaires standards :
  - MathML, SVG, XMI, …



Comment écrit-on une grammaire ?

Langage de grammaire d’un doc XML

- **DTD**
- **Le langage XML Schéma**
  - **Permet de définir le schéma d’un doc XML**
    - ***Les balises, leur ordre et le typage de leur contenu (= les données encadrées par les balises)***
  - **Un schéma est un doc XML !! (`.xsd` = XML Schema Description)**
- **Écrire un schéma (une `.xsd`) = écrire un doc XML**
  => utiliser un ensemble de balises ordonnancées ayant un contenu typé
- **Cet ensemble est défini dans le langage XML Schema**
- **Un document XML est dit** valide **lorsqu’il est**

conforme **à une grammaire**

Exemple de schéma : `livre.xsd`

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="livre">
<xs:complexType><xs:sequence>
    <xs:element name="titre" type="xs:string"/>
    <xs:element name ="chapitre">
    <xs:complexType><xs:sequence>
          <xs:element name="numero" type="xs:int"/>
          <xs:element name="titre" type="xs:string"/>
          <xs:element name="paragraphe" type="xs:string"/>
    </xs:sequence></xs:complexType></xs:element>
</xs:sequence></xs:complexType></xs:element>
</xs:schema>
```

Type de la balise

Balise pré-définie dans le langage XML Schema (mot du langage)

Nom d'une balise d'un doc XML valide à `livre.xsd`

Autre exemple

- Le schéma `personne.xsd`

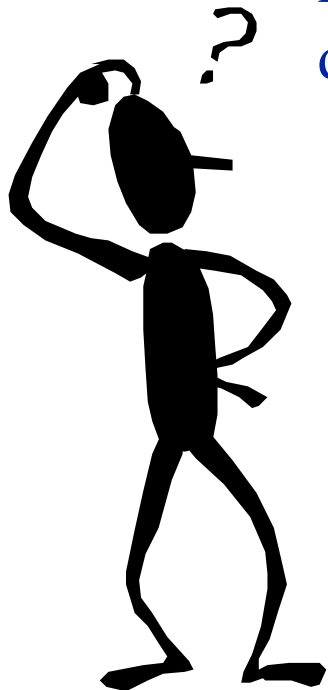
```
<?xml version="1.0" encoding="UTF-
  8"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/
  XMLSchema">
<xs:element name="personne">
  <xs:complexType><xs:sequence>
  <xs:element
       name="nom"
       type="xs:string"/>
  <xs:element
       name="prenom"
       type="xs:string"/>
  <xs:element
       name="date_naissance"
       type="xs:date"/>
</xs:sequence></xs:complexType>
</xs:element>
</xs:schema>
```

- Un doc XML valide: `personne.xml`

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


<personne xmlns:xsi="
  http://www.w3.org/2001/XMLSchema-
  instance"
  xsi:noNamespaceSchemaLocation="per
  sonne.xsd">

  <nom>De Latour</nom>
  <prenom>Jean</prenom>
  <date_naissance>1967-08-
  13</date_naissance>

</personne>
```

Soit un document XML qui inclut des balises issues de différentes grammaires (i.e., +sieurs fichiers xsd)

Ex : un livre de géométrie qui utilise
- le schéma `livre.xsd` et
- le schéma `svg.xsd`



Et si le schéma `livre.xsd` ET le schéma `svg.xsd` définissent chacun la balise `auteur` ??

```
<livre>
     <auteur>…</auteur>…
<figure><auteur>…</auteur></figure>
…
```

Prbl de conflit de nommage des balises !!

Espaces de noms de balises

- <u>Principe</u> : similaire à celui de package en java ou en UML !
  - Espace de noms : unité de partitionnement qui sert d'espace de désignation
- <u>Utilisation</u> : il suffit d'utiliser le nom qualifié de la balise ! c-à-d de préfixer le nom de la balise par le nom logique de l'espace de noms
  - Ex java : un package `P1` contient une classe `C1` et `P2` contient une autre classe qui s'appelle aussi `C1` => `P1.C1`, `P2.C1`
  - Ex UML : un package `P1` contient une classe `C1` et `P2` contient une autre classe qui s'appelle aussi `C1` => `P1::C1`, `P2::C1`
  - Ex doc XML : la balise `<auteur>` de l'espace de noms ayant le nom logique `nLivre` se note `<nLivre:auteur>`
- <u>Définition du nom logique</u> : le nom logique d'un espace de noms utilisé dans un doc XML est défini <u>dans n'importe quelle balise</u> par un triplet : l'attribut `xmlns`, le nom logique et l'URI (identifiant Web du `.xsd`)

```
<nom_balise xmlns:nom_logique="URI">
```

Ex. d’un doc XML avec 2 espaces de noms

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<commande
  xmlns:produit="http://localhost/XML/produit"
  xmlns:client="http://localhost/XML/client">

 <produit:numero>p49393</produit:numero>
 <produit:nom>JXY Rasierer VC100</produit:nom>
 <produit:quantite>1</produit:quantite>
 <produit:prix>69</produit:prix>

 <client:numero>c2029</client:numero>
 <client:nom>Marius, Escartefigues</client:nom>
 <client:adresselivraison>Cours Mirabeau 14, 13100 Aix en
  Provence</client:adresselivraison>

</commande>
```

Récapitulatif sur le langage XML Schema

- **Permet d'écrire des grammaires (= schémas) de doc XML**
  - **Pré-définit un ensemble de « mots »**
    - ***Des balises ordonnancées avec typage de contenu***
  - **Pré-définit un ensemble de types simples**
    - ***string, byte, int, long, float, decimal, double, time, boolean, date,…***
    - ***http://xmlfr.org/w3c/TR/xmlschema-0/#simpleTypesTable***

- **Écrire un schéma en langage XML Schéma, c'est**
  - **Écrire un doc XML**
  - **En utilisant les balises qui sont les mots du langage XML Schema**

```
<xs:schema …… />
<xs:element …… />
<xs:complexType>
…
```

- **Ces balises appartiennent à l'espace de noms**

`http://www.w3.org/2001/XMLSchema`

SOAP

Le protocole de couche application pour invoquer un service Web

SOAP

- Simple Object Access Protocol
- Protocole de transmission de messages pour l'invocation de services Web
  - Définit la structure des messages échangés par les applications via Internet
    - *Format des messages défini en XML*
  - Achemine les messages (en utilisant des protocoles standards sous-jacents)
    - *HTTP pour des appels synchrones*
    - *SMTP ou bus MOM pour des appels asynchrones*

Structure d'un message SOAP (1)

- **Un message SOAP est un document XML de la forme**
  - **Une déclaration XML (optionnelle), suivie de**
  - **Une Enveloppe SOAP (l'élément racine) : balise `Envelope`, qui est composée de :**
    - ***Une En-tête SOAP (optionnelle) : balise `header`***
      - **Pour des info d'authentification ou de gestion de session**
    - ***Un Corps SOAP : balise `body`***

Structure d'un message SOAP (2)

la balise `Envelope`

```
<soap:Envelope
      xmlns:soap=«http://……»
      soap:encodingStyle=«http://……»>
            <soap:Header> ……
            </soap:Header>

            <soap:Body>
                  blabla
            </soap:Body>

</soap:Envelope>
```

Identificateur de l'espace de noms utilisé dans le doc (il peut y en avoir plusieurs)

Schéma définissant les règles de sérialisation des données typées (facultatif)

Ex : appel de méthode

Ex : un service d'addition de 2 entiers

- Soit un service :

```
int add(int a, int b)
```

- L'utilisation de ce service par un client générera l'acheminement sur le réseau de 2 msg SOAP
  - Un pour la requête et un pour la réponse

Structure du message SOAP : la requête

```
<soapEnv:Envelope
   xmlns:serviceAddition="URImyAddition"
   xmlns:soapEnv="http://schemas.xmlsoap.org/soap/envelope/"
  <soapEnv:Body

     <serviceAddition:add>
          <arg0>5</arg0>
          <arg1>12</arg1>
     </serviceAddition:add>
  </soapEnv:Body>
</soapEnv:Envelope>
```

Structure du message SOAP : la réponse

```
<soapEnv:Envelope
   xmlns:serviceAddition="URImyAddition"
   xmlns:soapEnv="http://schemas.xmlsoap.org/soap/envelope/"
   <soapEnv:Body
      <serviceAddition:addResponse>
           <return>17</return>
      </serviceAddition:addResponse>
   </soapEnv:Body>
</soapEnv:Envelope>
```

URImyAddition

```
<xs:schema version="1.0" targetNamespace="URImyAddition">
<xs:element name="add" type="tns:add"/>
<xs:element name="addResponse" type="tns:addResponse"/>

  <xs:complexType name="add">
       <xs:sequence>
              <xs:element name="arg0" type="xs:int"/>
              <xs:element name="arg1" type="xs:int"/>
       </xs:sequence>
  </xs:complexType>

  <xs:complexType name="addResponse">

       <xs:sequence>
              <xs:element name="return" type="xs:int"/>
       </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Encodage

- Un message SOAP contient des données
  - Typées et interprétables par le service

=> Il faut définir un moyen d’encoder ces données qui soit compatible avec XML

- Encodage : représentation XML d’une donnée (valeur)
- Décodage : reconstitution d’une valeur à partir de sa forme XML
- La représentation XML d’une donnée est structurée en fonction du type de la donnée

=> Nécessaire de définir le type d’une donnée

Définition de types

- **Par mécanisme « utilisateur »**
- **Par utilisation de schémas (écrits en langage XML Schéma)**
  - **Pour décrire le type d'une donnée de type simple, c'est immédiat puisque le langage XML Schéma propose un système de types (int, boolean, string, …).**
    - *Voir l'exemple de* `URImyAddition`

```
<xs:element name="arg0" type="xs:int"/>
<arg0>5</arg0>
```

  - **Pour décrire le type d'une donnée de type complexe, utilisation de la balise** `<complexType>`

Acheminement des msg : SOAP avec HTTP

- **SOAP peut être facilement porté sur Http.**
  - **Convient au mode Request/Response de http**
  - **Le message SOAP est encapsulé dans une requête POST avec un content-type text/xml**
  - **Définition d'un header http : SOAPAction**

Msg SOAP encapsulé dans HTTP

```
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>DIS</symbol>
      </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Msg SOAP encapsulé dans HTTP

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"/>
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      < faultactor SOAP-ENV:Server</ faultactor>
       <faultstring>Server Error</faultstring>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP & RPC

- Pour faire un RPC SOAP, il faut:
  - L’URI de l’objet cible
  - Le nom de la méthode
  - Les paramètres de la méthode
- SOAP s’appuie sur le protocole sous-jacent (http) pour l’URI de l’objet.
- Le nom de la méthode et les paramètres sont encodés dans le message SOAP sous forme de structure.

WSDL

Le langage de définition
de l’interface d’un service Web

WSDL

- **Web Services Description Language**
- **Description en XML de l'interface publique d'utilisation des services Web (~ IDL des objets CORBA)**

- **Séparation entre description de la fonctionnalité abstraite du service et les détails concrets (où et comment la fonctionnalité est offerte)**
  - **Description abstraite des messages échangés entre un fournisseur de services et un client**
  - **Mise en relation, liaison (binding) de cette description avec un protocole (et donc le format des messages)**

Présentation

Une description WSDL :

1. Décrit l’interface d’un service web (méthodes, types des paramètres)
   Cette description peut être comparée à la description IDL CORBA, elle peut servir à générer automatiquement des amorces.

2. Décrit les aspects techniques d’implantation d’un service web (quel est le protocole utilisé, quelle est l’adresse du service)
   Cette description sert à se connecter concrètement à un service web.

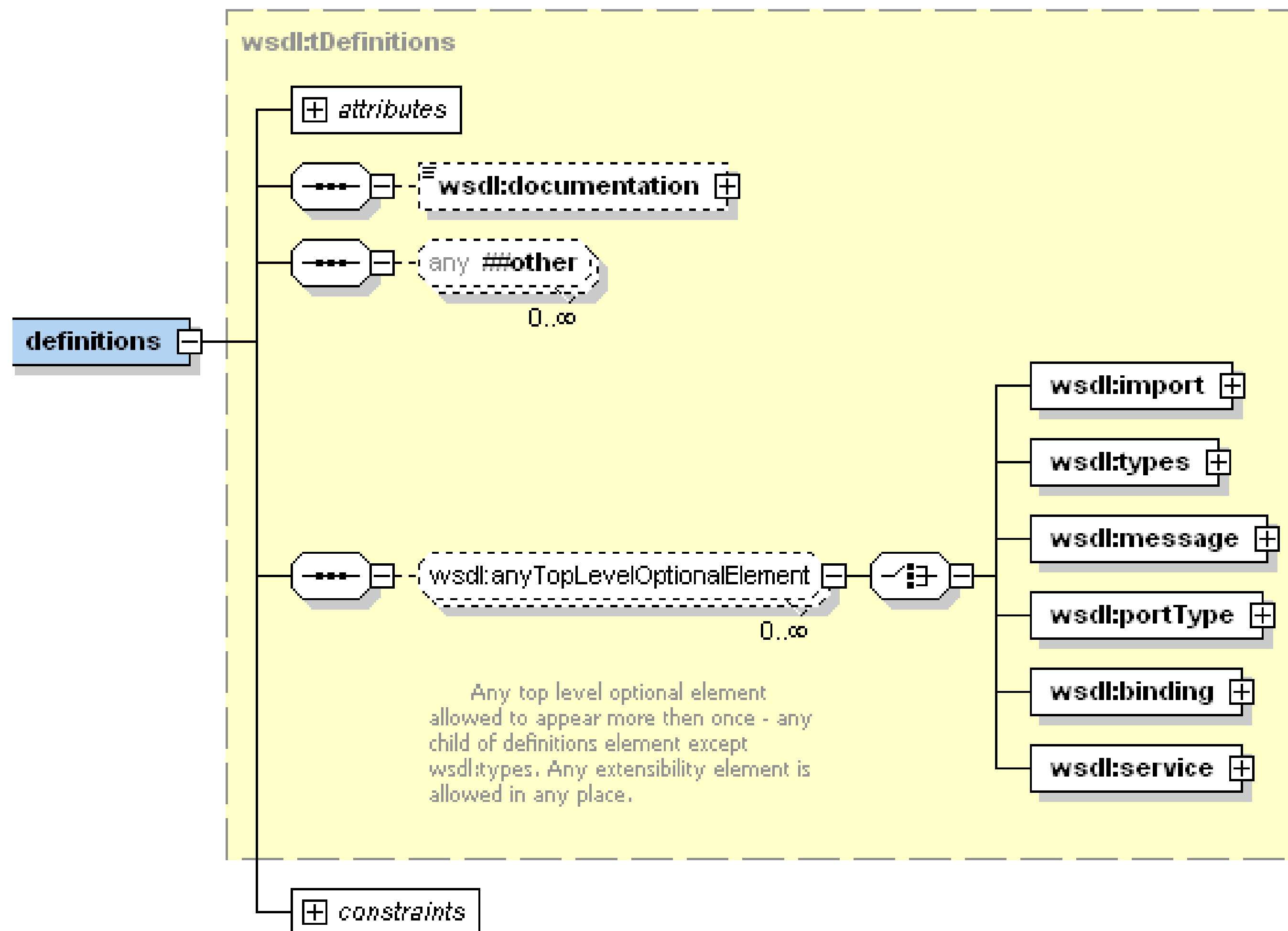
Balises

- Une description WSDL est un document XML qui commence par la balise definition et contient les balises suivantes :
  - types: cette balise décrit les types utilisés
  - message: cette balise décrit la structure d'un message échangé
  - portType: cette balise décrit abstraitement le service web sous forme d'un ensemble d'opérations (~ interface du service web)
    - *operation: cette balise décrit une opération réalisée par le service web. Une opération reçoit des messages et envoie des messages.*
  - binding : cette balise décrit la liaison entre un protocole (http) et la description abstraite du service (= le portType).
  - service: cette balise décrit un service comme un ensemble de ports.
    - *port: cette balise décrit un port de communication au travers duquel il est possible d'accéder à un ensemble d'opérations. Un port référence un Binding*

a
b
s
t
r
a
i
t

c
o
n
c
r
e
t

Les balises
(graphique XML Spy du wsdl.xsd)



types

- **Exemple de description du type `personne`**

```
<wsdl:types>
        <xs:schema
                targetNameSpace="http://www.exemple.fr/personne.xsd"
                xmlns:xs="http://www.w3.org/2001/XMLSchema">
          <xs:element name="personne">
                <xs:complexType>
                  <xs:sequence>
                        <xs:element name="nom" type="xs:string" />
                        <xs:element name="prenom" type="xs:string" />
                  </xs:sequence>
                </xs:complexType>
          </xs:element>
        </xs:schema>
</wsdl:types>
```

types (suite)

- Pas obligé de définir les types au sein même du `.wsdl`, on peut en importer

```
<types>
  <xsd:schema>
      <xsd:import namespace="http://session.mpg/"
      schemaLocation="http://localhost:8080/locationDuXSD"/>
  </xsd:schema>
</types>
```

message

- **Les messages sont envoyés entre le service et son client (et réciproquement !)**
  - **ex: une opération reçoit des messages et envoie des messages.**
- **Un message peut être avoir plusieurs paramètres appelés** parts
- **Les paramètres sont typés**

Paramètres de message

- **Paramètre de type simple**

```
<wsdl:message name="personneMsg">
   <wsdl:part name="nom" type="xsd:string" />
   <wsdl:part name="prenom" type="xsd:string" />
</wsdl:message>
```

- **Paramètre de type complexe (type défini dans un schéma)**

```
<wsdl:message name="personneMsg">
   <wsdl:part name="personne" element="exemple:personne" />
</wsdl:message>
```

portType

- **Description abstraite du service Web sous forme d'un ensemble d'opérations**
- **Un portType a un nom (attribut `name` de la balise `portType`)**

```
<wsdl:portType name=”gererPersonnes" >
      <wsdl:operation name="getPersonne" >
            …
      </wsdl:operation>
             <wsdl:operation name=“setPersonne" >
            …
      </wsdl:operation>
</wsdl:portType>
```

operation

- WSDL définit 4 catégories d'opération :
  - One-Way : une opération qui reçoit des messages mais n'en émet pas
  - Request-response : une opération qui reçoit des messages puis renvoit des messages
  - Solicit-response : une opération qui envoie des messages puis en reçoit
  - Notification : une opération qui envoie des messages mais n'en reçoit pas

operation

- Quelque soit le catégorie d'opérations, la définition est sensiblement la même :
- Une opération :
  - Reçoit des messages : <wsdl:input …>
  - Envoie des messages : <wsdl:output …> ou <wsdl:fault …>
- La catégorie de l'opération détermine la présence et l'ordre des input/outputs/fault

operation

```
<wsdl:operation name="operation_name">
  <wsdl:input name="nom_optionel" message="nom_message" />
</wsdl:operation>

<wsdl:operation name="operation_name">
  <wsdl:input name="nom_optionel" message="nom_message" />
  <wsdl:output name="nom_optionel" message="nom_message" />
  <wsdl:fault name="nom_optionel" message="nom_message" />*
</wsdl:operation>

<wsdl:operation name="operation_name">
  <wsdl:output name="nom_optionel" message="nom_message" />
  <wsdl:input name="nom_optionel" message="nom_message" />
  <wsdl:fault name="nom_optionel" message="nom_message" />*
</wsdl:operation>
```

binding

- Liaison d’une description abstraite (portType) à un protocole
- Chaque opération d’un portType peut être liée de manière différente (à un protocole distinct)
- Protocoles standardisés pour les liaisons
  - SOAP
  - HTTP
  - MIME

binding

- **Une liaison a**
  - **Un nom : attribut (optionnel)** `name` **de la balise binding**
  - **Un type : attribut** `type` **de la balise binding**
    - ***Le type identifie le portType (i.e., le nom du type de la liaison est le nom du portType)***

```
<wsdl:binding name="binding_name"
      type="nom du portType" >
...
</wsdl:binding>
```

Le cas du binding SOAP

- **Pour préciser que la liaison est de type SOAP, il faut inclure la balise suivante :**

```
<soap:binding transport="uri" style="soap_style" />
```

- **L'attribut *transport* de la balise définit le protocole qui encapsule les messages SOAP**
  - **Ex : pour HTTP, la valeur de l'attribut `transport` est `http://schemas.xmlsoap.org/soap/http`**

- **L'attribut *style* identifie le mode de traduction d'une liaison en messages SOAP (comment créer les messages SOAP à partir des définitions WSDL des opérations ?)**
  Deux valeurs possibles :
  - `rpc`
  - `document`

binding SOAP

- Pour chaque opération du portType :
  - Il faut préciser l'URI de l'opération : soapAction
  - On peut (pas obligatoire) repréciser la façon dont sont créés les messages SOAP : `style`

- Pour chaque message input/output/fault de chaque opération, il faut définir le mode d'usage (encodage d'un message WSDL en un message SOAP)
  Deux valeurs possibles de l'attribut `use` de la balise `body`
  - `encoded`
  - `literal`

Pour aller plus loin…

http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/

service

- **Un service est accessible à un port**
- **Un port : point d'accès au service**
  - **Référence une liaison (attribut binding dont la valeur doit être le nom de la liaison)**
  - **A une adresse (qui correspond à l'adresse http)**

```
<wsdl:service name="MonService">
   <wsdl:port     binding="intf:MonServiceSoapBinding">
         <soap:address
         location="http://mda.lip6.fr:8080/axis/services/MonService"/>
   </wsdl:port>
</wsdl:service>
```

Résumé du vocabulaire WSDL

- **Modèle logique (abstrait)**
  - Un message = description abstraite des données transmises lors d'un échange
  - Une opération = groupement logique de messages
  - Un portType = une collection d'opérations
- **Correspondance logique-physique : binding**
  - description concrète, i.e., protocole et format de données qui implantent un portType
- **Modèle physique**
  - Un port = point d 'accès au service défini par une liaison et une adresse réseau
  - Un service = un ensemble de ports

Exemple

```
<?xml version="1.0" ?>
  <definitions name="CurrencyExchangeService"
   targetNamespace="http://www.xmethods.net/sd/CurrencyExchangeService.wsdl »
   xmlns:tns="http://www.xmethods.net/sd/CurrencyExchangeService.wsdl »
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/ »
   xmlns="http://schemas.xmlsoap.org/wsdl/">
  <message name="getRateRequest">
   <part name="country1" type="xsd:string" />
   <part name="country2" type="xsd:string" />
  </message>

  <message name="getRateResponse">
   <part name="Result" type="xsd:float" />
  </message>

  <portType name="CurrencyExchangePortType">
   <operation name="getRate">
        <input message="tns:getRateRequest" />
        <output message="tns:getRateResponse" />
        </operation>
  </portType>
```

Exemple (suite)

```
<binding name="CurrencyExchangeBinding" type="tns:CurrencyExchangePortType">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
  <operation name="getRate">
    <soap:operation soapAction="" />
         <input>
                 <soap:body use="encoded"
                 namespace="urn:xmethods-CurrencyExchange »
                 encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
         </input>
         <output>
                 <soap:body use="encoded"
                 namespace="urn:xmethods-CurrencyExchange »
                 encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
         </output>
    </operation>
</binding>

<service name="CurrencyExchangeService">
  <documentation>Returns the exchange rate between the two
    currencies</documentation>
    <port name="CurrencyExchangePort" binding="tns:CurrencyExchangeBinding">
                 <soap:address location="http://services.xmethods.net:80/soap" />
    </port>
</service>
</definitions>
```

Exercice

Analyse d'un fichier WSDL

`MaCalculetteBean.wsdl`

Développement de Web Services

Avec JavaEE

Les WS basés sur un stateless session EJB

Étapes de réalisation d'un WS
(basé sur un stateless session EJB)

- 1. Faire un code EJB stateless
- 2. Ajouter les tags
- 3. Créer l'archive
- 4. Déployer l'EJB

Exemple de l’addition (options par défaut)

```
@WebService
@Stateless (mappedName="additionWS")
public class AdditionBean implements AdditionItf {
  @WebMethod
    public int add(int nombre1, int nombre2) {
    return nombre1+nombre2;
    }
}
```

Devient le nom du **portType**.
Le nom du **service** est alors `AdditionBeanService`.
Le nom du **port** est alors `AdditionBeanPort`

Home | Version | Logout | He

User: admin | **Domain:** myDomain | **Server:** localhost

Sun Java™ System Application Server Admin Console

Common Tasks

- Application Server
- Applications
  - Enterprise Applications
  - Web Applications
  - EJB Modules
  - Connector Modules
  - Lifecycle Modules
  - Application Client Modules
- Web Services
  - MiageWs
  - AdditionBean
  - echoCompoWS
  - timeCompoWS
  - echoWS
- JBI
  - Service Assemblies
  - Components
  - Shared Libraries
- Custom MBeans
- Resources
- Configuration

Web Services

General | Publish | Monitor | Transformation

Web Service - AdditionBean

The Test button is unavailable for a web service if it is a secure web service or if the web service is disabled.

Test

Name:	AdditionBean
Endpoint Address URI:	/AdditionBeanService/AdditionBean
Application:	
WSDL:	View WSDL
Webservices.xml:	Webservices.xml
Implementation Type:	EJB
Implementation Class Name:	
Deployment Descriptors:	ejb-jar.xml sun-ejb-jar.xml

Exemple de l'addition utilisant des options

```
@WebService(
  name="nEssai",          //nom du PortType
  serviceName="snEssai",      //nom du service
  targetNamespace="http://localhost:8080/snEssai/nEssai")
@Stateless (mappedName="additionWSoption")
public class AdditionBean implements AdditionItf {
  @WebMethod(operationName="addInt")
     public int add(int nombre1, int nombre2) {
    return nombre1+nombre2;
     }
}
```

Les WS utilisant l’API JAX-WS

Le codage d’un service Web

Étapes de réalisation d’un WS (basé sur l’API JAX-WS)

- 1. Coder la classe du service (POJO !!)
  - => Il n’y a pas d’interface !! Pas EJB !!
- 2. Compiler la classe
- 3. Utiliser l’outil `wsgen` pour générer les stub et les fichiers nécessaires au WS
- 4. Créer une archive `war` avec jar
- 5. Déployer le fichier `war` comme si c’était un module d’application Web

Règles d'écriture d'un WS (API JAX-WS)

- **La classe du service (le POJO)**
  - **doit être annoté par `WebService` ou `WebServiceProvider`**
  - **ne doit pas être `final` ou `abstract`**
  - **doit avoir un constructeur public sans paramètre**
- **Les méthodes métiers publiées de la classe**
  - **doivent être annotées par `WebMethod` (`javax.jws.WebMethod`)**
  - **ne doivent pas être `final` ou `static`**
  - **doivent être publiques**
- **On ne doit pas re/définir la méthode `finalize`**
- **On peut définir les callback `@preDestroy` et `@postConstruct`**
  - **`@PostConstruct` est invoqué par le container avant que le WS ne réponde aux requêtes**
  - **`@PreDestroy` est invoqué lors de l'arrêt du service**

Un exemple avec JAX-WS : echo

```
package webservice;
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public class echoWS    // donne son nom au PortType
{
  public echoWS () {}

  @WebMethod
  public String echo(String chaine) {
    return chaine;
  }
}
```

Un exemple avec JAX-WS : préparation

- **Créer un répertoire `serveur`**
- **Dans ce répertoire (`cd serveur`), créer quatre répertoires :**
  - **`src` (on y range les sources `.java`)**
  - **`classes` (contiendra les `.class` après la compilation des sources)**
  - **`srcGenerated` et `generated` (contiendront les fichiers générés)**

Un exemple avec JAX-WS : la compilation

- **Compilation : `javac` à partir du répertoire `src`**
  - `javac -d ../classes package/*.java`
  - **Ex de Echo : `javac -d ../classes webservice/*.java`**

- **Génération des talons et autres fichiers :**
  - **Utilitaire `wsgen` lancé à partir du répertoire `classes`**

```
wsgen -d dest_dir -s src_dir -cp class_dir -wsdl
                package.la_classe
```

  `-d` ➔ **où ranger les fichiers `.class` générés**

  `-s` ➔ **où ranger les fichiers `.java` générés**

  `-cp` ➔ **où trouver le fichier .class de la classe**

  `-wsdl` ➔ **pour générer un fichier wsdl (-servicename, -portname)**

  Ex de Echo :

```
wsgen -d ../generated -s ../srcGenerated -cp . -wsdl
  webservice.echoWS
```

Un exemple avec JAX-WS : le packaging

- Construction de l'archive `.war`
  - Les classes doivent être dans `WEB-INF/classes`
- Le déploiement : comme tout `.war`
  - Avec `asadmin (Glassfish)` ou
  - Avec l'IHM

Un exemple avec JAX-WS : le test

- **Exercice 1**
  - **Voir les commandes de la GUI Web du serveur SunApp**
  - **Récupérez le WSDL et analysez-le**
    - ***Quel est le nom du service ? de l'interface (PortType) ?***
    - ***Quelle(s) est (sont) le(s) nom(s) des opérations que l'on peut invoquer ?***
    - ***etc***
  - **Faire un test du service (bouton Test de la GUI)**
    - ***Quelle est l'URL du service ?***
- **Exercice 2 : récupérer les info du service de votre voisin (remplacer localhost par l'@ IP de sa machine)**

  => on a toutes les info pour écrire un client

Les WS utilisant l’API JAX-WS

Le codage d’un client statique

(l’interface est connue à la compilation)

JAX-WS : le client statique

- **Le client utilise des stub pour communiquer avec le WS.**

=> Il faut d'abord générer côté client les classes concernant le WS : cf la commande `wsimport`

- **Puis on peut compiler et exécuter comme tout client java**
- **Préambule : Configuration de la variable `classpath`**

  Avoir les deux librairies suivantes

  - `<JAVAEE_HOME>/lib/appserv-ws.jar`
  - `<JAVAEE_HOME>/lib/activation.jar`

1 - Génération de fichiers côté client

- **Préparation : créer un répertoire**
  - `src` (on y range les sources générés et le source client)
  - `build` (on y range les .class générés et le .class client)
- **Génération : `wsimport`, à partir du répertoire `src`**

```
wsimport -d dest_dir -s src_dir le_fichier_wsdl -p package
```

`-d` ➔ où mettre les fichiers `class` générés

`-s` ➔ où mettre les fichiers `java` générés

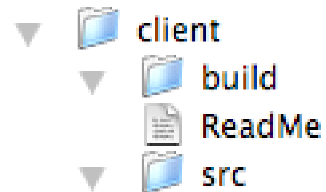
`-p` ➔ nom du package généré pour ranger les classes importées

  - Le fichier wsdl peut être un fichier ou une URL, À PLACER ENTRE GUILLEMETS !
  - `wsimport` génère des noms de classes identiques à ceux du service. ATTENTION AUX CONFLITS DE NOM !!! Il faut utiliser l'option –p

```
wsimport -d ../build -s .
  "http://localhost:8080/echoWS/echoWSService?WSDL"
  -p importservice
```

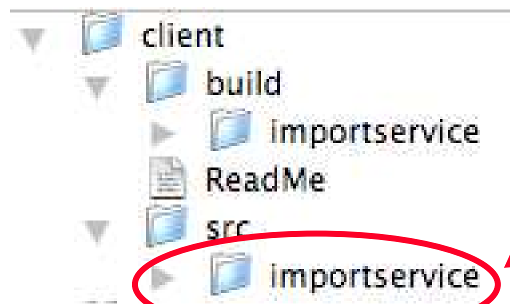
Illustration du `wsimport`

- **Avant**



- `wsimport -d ../build -s . "http://localhost:8080/echoWS/echoWSService?WSDL" -p importservice`

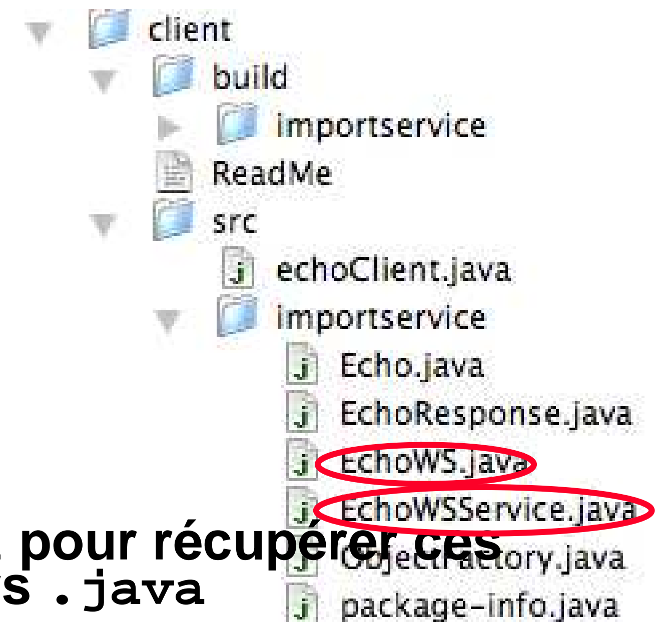
- **Après**



2 - Écriture du client

- **Le `.java` client doit**
  - **Être rangé dans le même package que le package généré**
  - **Importer le package généré**

- **Il utilise les fichiers `.java` générés par la lecture du `.wsdl`**
  - **~ l'interface du service**
    - ***= nom du portType (ex : EchoWS)***
  - **Le service (ex : EchoWSService)**
    - ***Pour instancier***
    - ***Pour avoir une référence de port***



=> Pour toute écriture de client, consulter le `wsdl` pour récupérer ces noms et pour identifier correctement les fichiers `.java`

=> Regarder les deux fichiers correspondants

2 - Écriture du client (suite)

- Soit `XXX` le nom du PortType (~ l'interface) et `XXXService` le nom du service
- 1. Instancier la classe `XXXService` (celle qui hérite de la classe `Service`) pour récupérer une référence sur le WS

```
static XXXService service;
service = new XXXService(
        new URL(URL_du_wsdl),
        new QName(targetNameSpace, "XXXService"));
```

- 2. Récupérer une référence de port qui permettra l'invocation des opérations
  - Utiliser la méthode de la classe `XXXService` du style `getXXXXPort()` => regarder dans la classe pour avoir la syntaxe exacte
- 3. Invoquer les méthodes de l'interface => regarder le fichier `XXX.java` pour avoir la syntaxe exacte

```
String response = port.laMethode(leParam);
```

Un exemple avec JAX-WS : un client statique

```
import java.net.URL;
import javax.xml.ws.WebServiceRef;
import javax.xml.namespace.QName;
import importservice.*;
public class echoClient {
   static EchoWSService service;
   public static void main(String[] args) {
   try {
         EchoWSService service = new EchoWSService(
               new URL("http://localhost:8080/echoWS/echoWSService?WSDL"),
               new QName("http://webservice/", "echoWSService"));
         System.out.println("Retrieving the port from the
                     following service: "+ service);
         EchoWS port = service.getEchoWSPort();
         System.out.println("Invoking the echo operation on the port.");
         String response = port.echo(args[0]);
         System.out.println("Echo value ==> "+response);
         } catch (Exception e) {
               e.printStackTrace();}
   }
}
```

3 - La compilation et l'exécution

- Compilation, à partir du répertoire `src`

```
javac -d ../build echoClient.java
```

- Exécution, à partir du répertoire `build`

```
java echoClient "un echo qui va bien"
```

  - Résultat :

```
Retrieving the port from the following service:
mpg.importservice.EchoWSService@163765
Invoking the echo operation on the port.
echo value ==> un echo qui va bien
```

Les WS utilisant l'API JAX-WS

Le codage d'un client dynamique

(interface non connue à la compilation)

Introduction au client dynamique

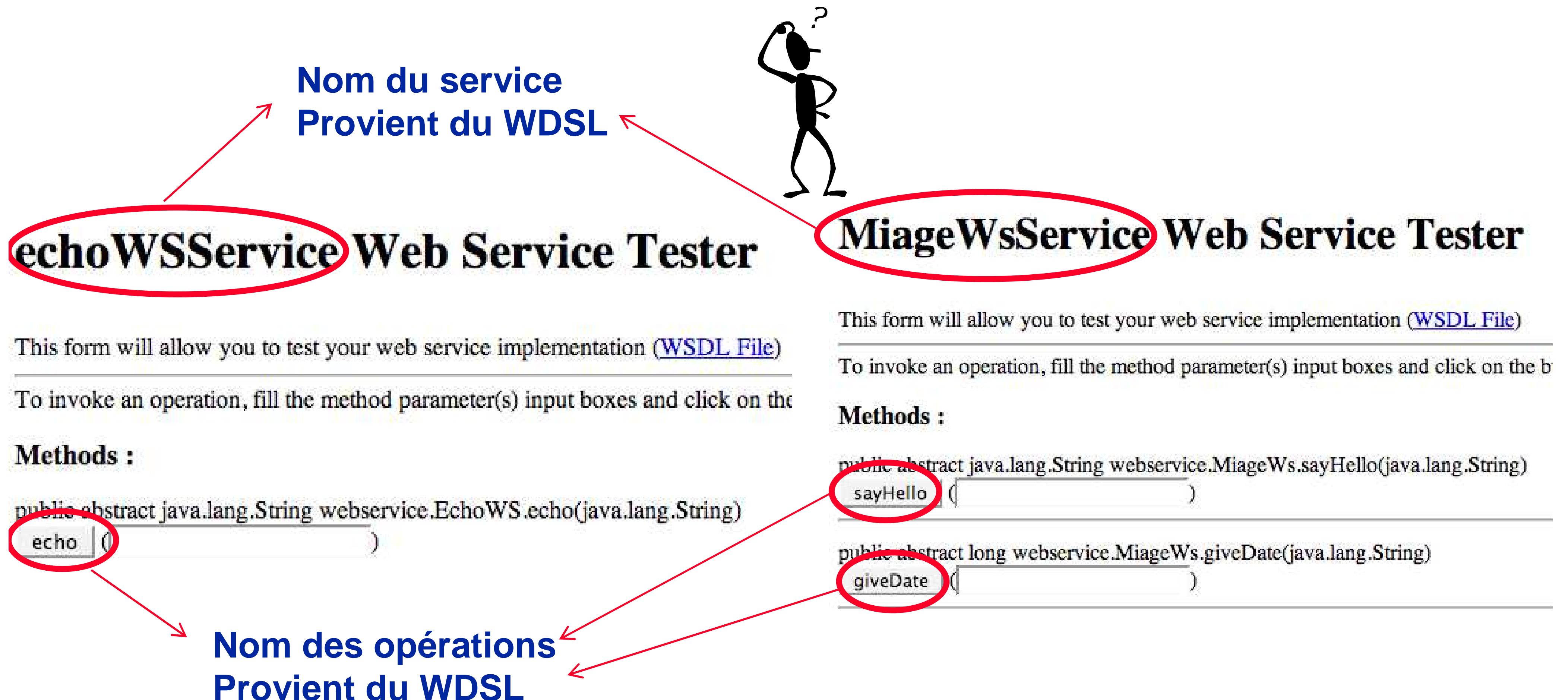
- Dans les clients développés précédemment (client statique), le développeur :
  - Connaît et analyse le `WSDL` et les stubs (fichiers java générés avec `wsimport` à partir du `WSDL`)
  - Génère à la main les fichiers java (avec `wsimport` à partir du `WSDL`)

  => on code « en dur » dans le client le targetName, le nom du service, l'Url, les noms de méthode, etc
- Liaison dynamique :

  => Programmer de façon générique l'invocation de service
  - Ne pas générer à la main (ne pas faire le `wsimport`)
  - Ne pas étudier le `WSDL`
  - Ne pas coder « en dur »

Motivation par l'exemple

- Si vous deviez programmer la page Web `test` de la console d'admin utilisée pour tester les services Web



Motivation par l'exemple

- **Comment écrire cette jsp `test` qui n'est pas dédiée à un WS et qui pour n'importe quel WS à tester doit :**
  - **Afficher le nom du service**
    - ***Change à chaque WS que l'on teste***

    => on ne peut pas le coder "en dur" dans la jsp, elle n'est pas dédiée à un WS
  - **Invoquer la méthode du WS que l'on teste**
    - ***Quel nom ? Il peut changer à chaque utilisation de la jsp***

    => la jsp doit doit "fabriquer" les invocations

Principe

- La jsp `test` (= le programme client) dispose du `WSDL` et le parcourt pour en extraire dans des variables les informations utiles (dont le nom du service le nom de la méthode, etc.)
  - Utilisation d'une API (DOM ou SAX)

Comment code-t-on l'invocation de méthode si on n'a pas son nom (puisqu'il est dans une variable) ?

=> Pas possible de faire une invocation standard

- Le développeur doit écrire le message SOAP lui-même
  - Utilisation de `javax.xml.soap`

Un exemple JAX-WS : un client dynamique

- Exemple de l'exo du service d'affichage de bienvenue
- Le msg SOAP de requête est (extrait) :

```
<SOAP-ENV:Body>
  <serviceBonjour:sayHello
                             xmlns:serviceBonjour="http://webservice/">
        <arg0>Marie-Pierre</arg0>
  </serviceBonjour:sayHello>
</SOAP-ENV:Body>
```

=> on va programmer la fabrication de ce msg

- Étape 1 : Initialiser les info utiles en parsant le WSDL avec DOM

```
// ces init remplacent le parse du WSDL avec DOM
   String targetNamespaceDuWSDL = new String("http://webservice/");
   String nomMethode = new String("sayHello");
   String nomParam1 = new String("arg0");
   String valeurParam1 = new String("Marie-Pierre");
   String location = new
                     String("http://localhost:8080/MiageWs/MiageWsService");
   String nomLogique = new String("serviceBonjour");
```

Exemple (suite)

- **Étape 2 : Création du msg SOAP**

```
MessageFactory factory = MessageFactory.newInstance();
SOAPMessage message = factory.createMessage();
SOAPBody body = message.getSOAPBody();
```

- **Étape 3 : Création de la requête (dans <body>)**

```
QName serviceName = new Qname
                (targetNameSpaceDuWSDL, nomMethode, nomLogique);
SOAPElement methodElement = body.addChildElement(serviceName);
QName arg0name = new QName(nomParam1);
SOAPElement arg0 = methodElement.addChildElement(arg0name);
arg0.addTextNode(valeurParam1);
message.saveChanges();
```

On peut afficher le msg créé (pour voir) :

```
message.writeTo(System.out); /* dans un try/catch */
```

Exemple (suite)

- Étape 4 : envoi du msg et réception réponse
  - Étape 4.1 : se connecter à la fabrique de connexion

```
SOAPConnectionFactory soapCF =
                SOAPConnectionFactory.newInstance();
SOAPConnection connection =
                 soapCF.createConnection();
```

  - Étape 4.2 : effectuer l’envoi et recevoir la réponse

```
URL endpoint = new URL(location);
SOAPMessage response = connection.call(message, endpoint);
connection.close();
```

On peut afficher le msg reçu (pour voir) :

```
 response.writeTo(System.out); /* dans un try/catch */
```

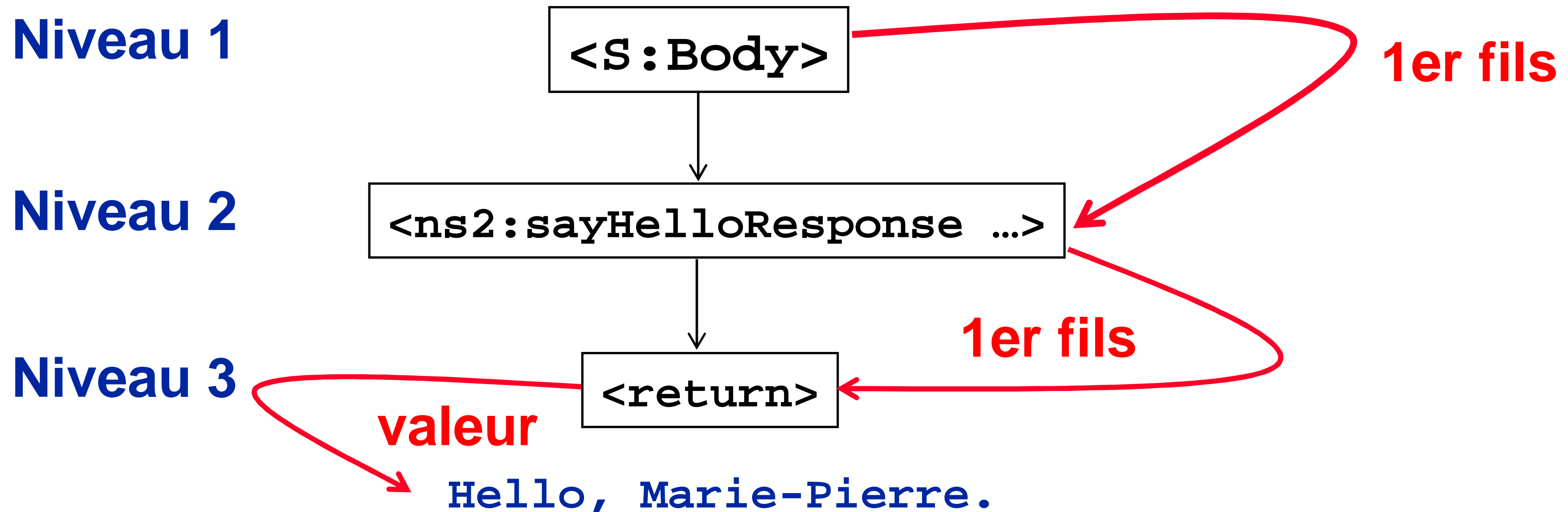
Exemple (suite)

- **Étape 5 : parser le msg de réponse pour en extraire le contenu (résultat de la requête)**
  - **Comprendre la structure DOM du msg de réponse : analyse de l'extrait de ce msg qui contient le retour de l'exécution de la méthode :**

```
<S:Body>
  <ns2:sayHelloResponse xmlns:ns2="http://webservice/">
      <return>Hello, Marie-Pierre.</return>
  </ns2:sayHelloResponse>
</S:Body>
```

Représentation DOM (simplifiée)

- **Les éléments du msg sont des objets représentés dans une structure arborescente**



Étape 5 : le code

```
//se positionner sur l'élément de Niveau 1
SOAPBody soapBody = response.getSOAPBody();

//se positionner sur l'élément de Niveau 2,
//1er fils de l'élément courant
SOAPBodyElement bE = (SOAPBodyElement) soapBody.getFirstChild();

//se positionner sur l'élément de Niveau 3,
//1er fils de l'élément courant
SOAPBodyElement bodyElement = (SOAPBodyElement)bE.getFirstChild();

//Récupérer la valeur de l'élément courant et l'afficher
String result = bodyElement.getValue();
System.out.println("Retour de sayHello: "+result);
```