

## Services Web

### 04–Protocole SOAP

Mohamed ZAYANI

ISET-SFAX- 2024/2025

# Plan

## 3. Protocole SOAP

**1–Rôle et  
Fonctionnement**

**2–Structure d'un  
message SOAP**

**3–Bibliothèque  
JAX-WS**

# Le protocole SOAP

## ➤ Rôle

- Assurer les appels de procédures à distance

## ➤ Fonctionnement côté client

- Ouverture d'une connexion HTTP
- La requête SOAP est un document XML décrivant:
  - ✓ Une méthode distante à invoquer
  - ✓ Les paramètres de la méthode

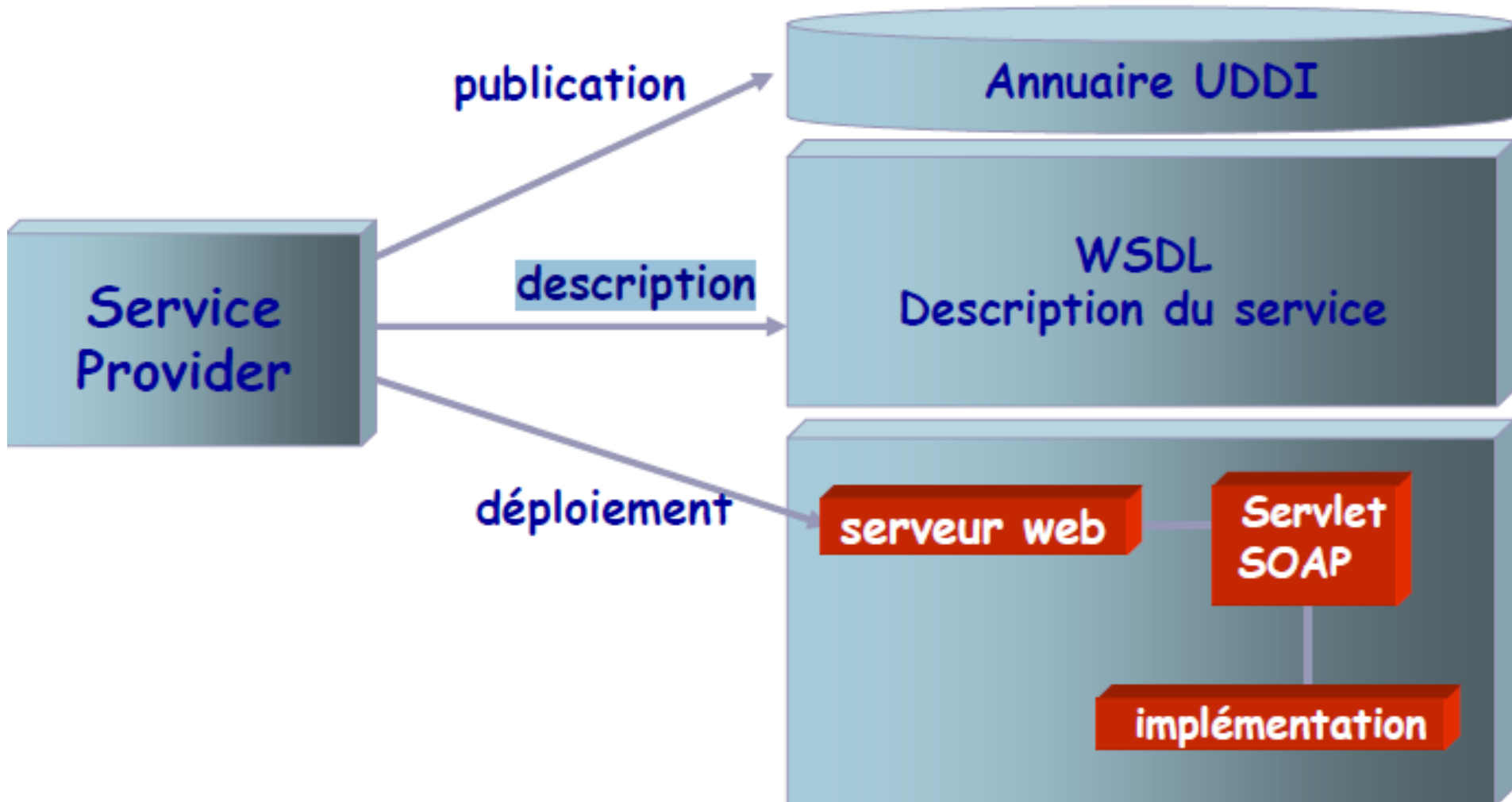
## ➤ Fonctionnement côté serveur SOAP

- Récupérer la requête
- Exécuter la méthode concernée
- Renvoyer une réponse SOAP (document XML) au client

# Le langage WSDL

- Un fichier WSDL est une interface qui cache l'implémentation du service, permettant une utilisation indépendante:
  - De plate forme utilisée
  - Du langage utilisé
- Un fichier WSDL est au format XML qui regroupe toutes les informations nécessaires pour interagir avec le Web Service:
  - Les méthodes
  - Les paramètres et les valeurs retournées
  - Le protocole de transport utilisé
  - La localisation du service
- Deux types de documents WSDL:
  - Un document pour décrire l'interface du service
  - Un document pour décrire l'implémentation du service

# Le fournisseur du service



# Structure d'un message SOAP



```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding">
  <soap:Header>
    <!-- en-tête -->
  </soap:Header>
  <soap:Body>
    <!-- corps -->
  </soap:Body>
</soap:Envelope>
```

# Mise en œuvre des WS avec JAX-WS

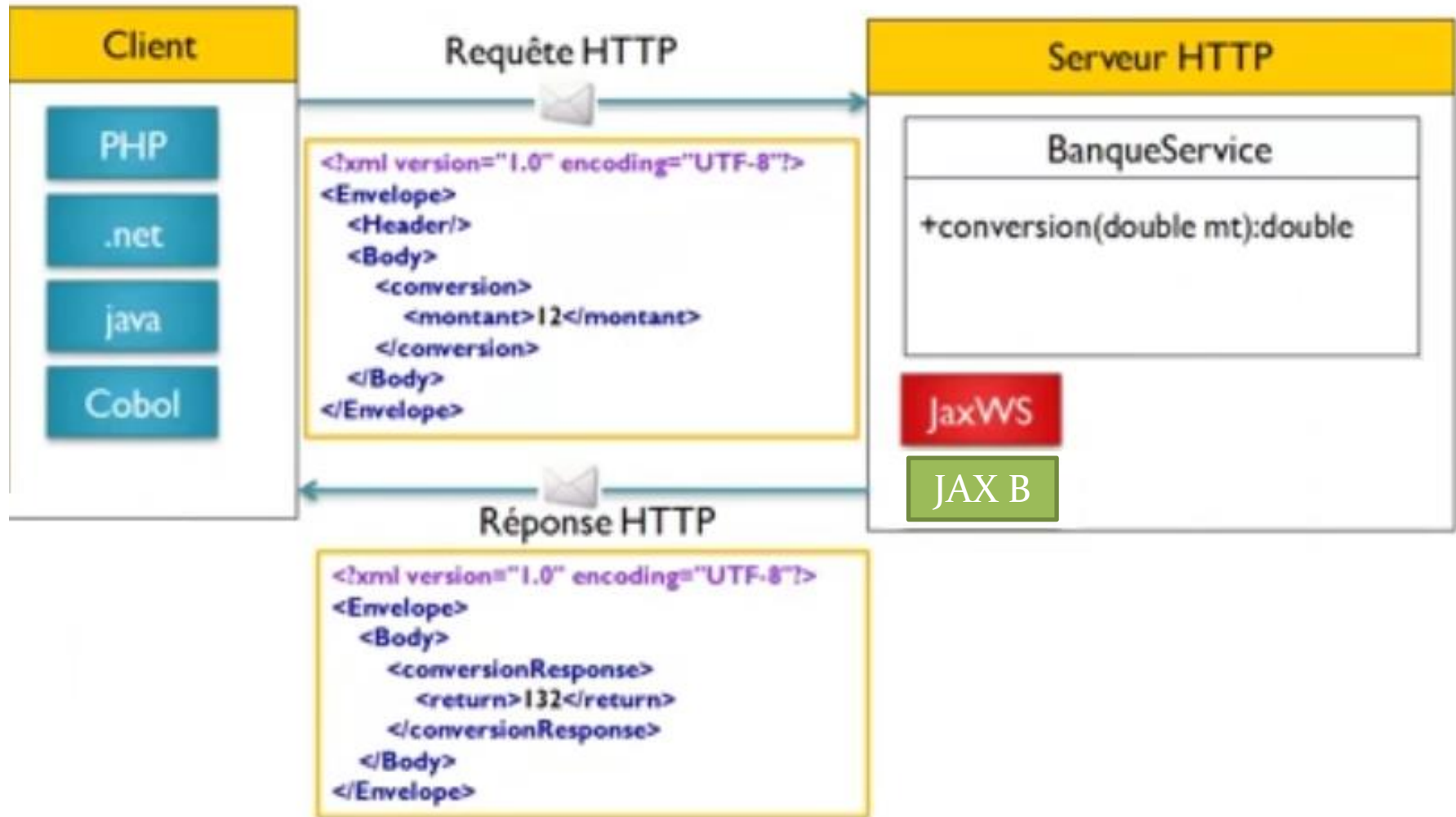
- **JAX-WS (Java Api for Xml pour les Web Services)**
  - C'est une **API** intégrée à partir du **JDK 1.6** pour créer des WS en utilisant **les annotations**.
  - Pour les dernières versions de JDKs, JAX-WS n'est plus présente: Elle est définie dans une API à part (dans le projet jakarta)
- Une nouvelle appellation de JAX-RPC
- Fournit un ensemble d'annotations pour mapper la correspondance JAVA-WSDL
- Il suffit d'annoter directement les classes Java qui vont représenter les services web

```
<dependency>  
  <groupId>jakarta.xml.ws</groupId>  
  <artifactId>jakarta.xml.ws-api</artifactId>  
  <version>4.0.0</version>  
</dependency>
```



Le document WSDL est **auto-généré** par le serveur d'application au moment du déploiement

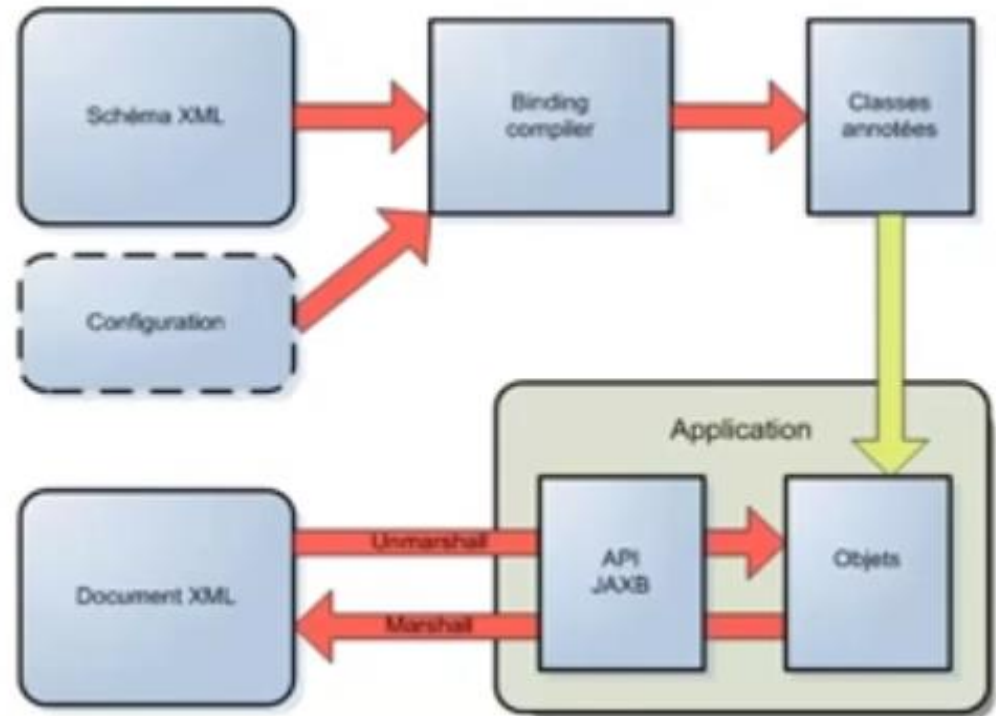
# 1. Exemple d'architecture globale





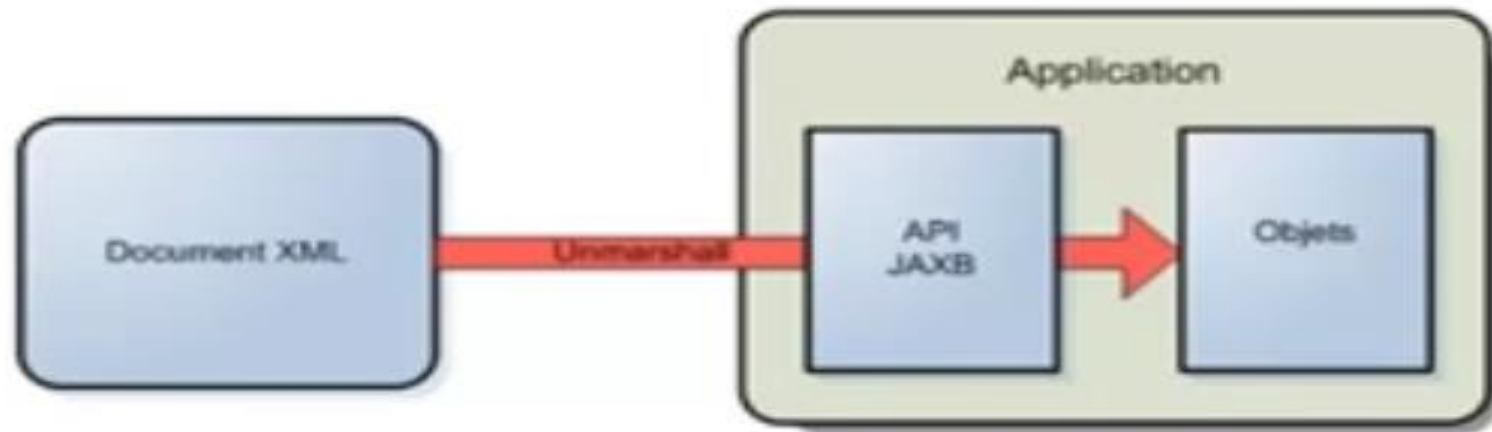
# JAX-WS / JAXB

- C'est une **API** utilisée par JAX-WS pour réaliser la correspondance entre un document XML et objets JAVA
- JAXB permet de mapper des objets JAVA dans un document XML et vice versa
- JAXB permet aussi de générer des classes JAVA à partir d'un schéma XML et vice versa

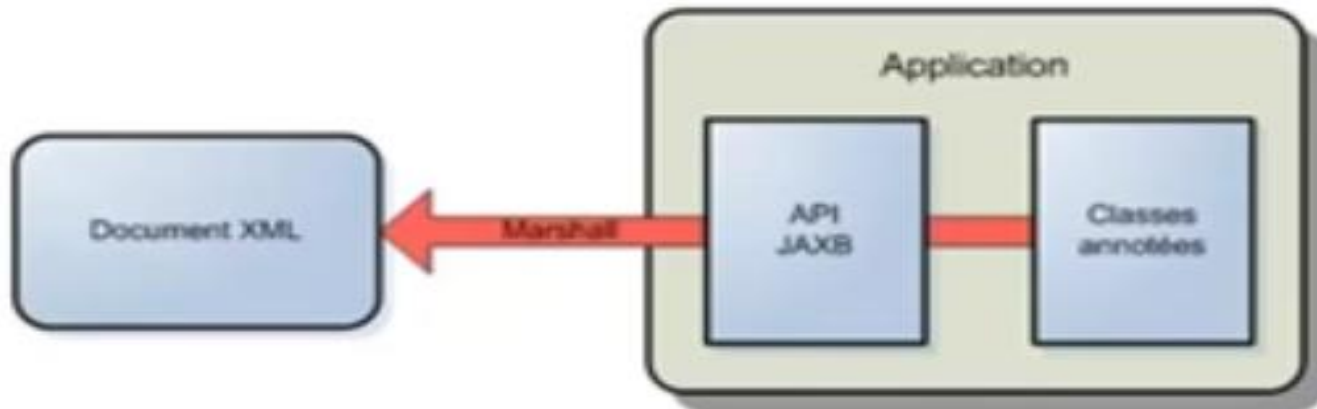


# Le principe de JAXB

- Le mapping d'un document XML à des objets (unmarshal)



- La création d'un document XML à partir d'objets (marshal)



# Générer XML à partir d'objets JAVA avec JAXB

```
package ws;
import java.io.File; import java.util.Date;
import javax.xml.bind.*;
public class Banque {
public static void main(String[] args) throws Exception {
    JAXBContext context=JAXBContext.newInstance(Compte.class);
    Marshaller marshaller=context.createMarshaller();
    marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,true);
    Compte cp=new Compte(1,8000,new Date());
    marshaller.marshal(cp,new File("comptes.xml"));
}}
```



```
package ws;
import java.util.Date;
import javax.xml.bind.annotation.*;
@XmlRootElement
public class Compte {
    private int code;
    private float solde;
    private Date dateCreation;
    // Constructeur sans paramètre
    // Constructeur avec paramètres
    // Getters et Setters
}
```

Fichier XML Généré : comptes.xml

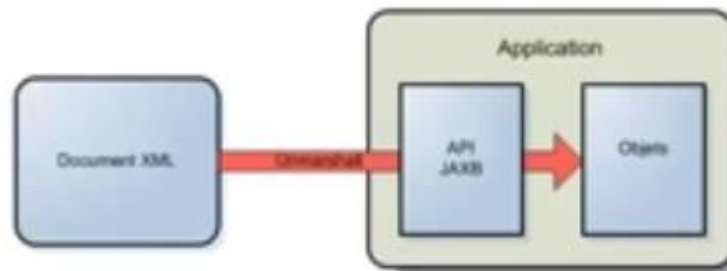
```
<?xml version="1.0" encoding="UTF-8"?>
<compte>
  <code>1</code>
  <dateCreation>
    2014-01-16T12:33:16.960Z
  </dateCreation>
  <solde>8000.0</solde>
</compte>
```

# Générer des objets JAVA à partir d'un XML avec JAXB

```
package ws;
import java.io.*;
import javax.xml.bind.*;
public class Banque2 {
public static void main(String[] args) throws Exception {
    JAXBContext jc=JAXBContext.newInstance(Compte.class);
    Unmarshaller unmarshaller=jc.createUnmarshaller();
    Compte cp=(Compte) unmarshaller.unmarshal(new File("comptes.xml"));
    System.out.println(cp.getCode()+"-"+cp.getSolde()+"-
"+cp.getDateCreation());
}
}
```

Fichier XML source : comptes.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<compte>
  <code>1</code>
  <dateCreation>
    2014-01-16T12:33:16.960Z
  </dateCreation>
  <solde>8000.0</solde>
</compte>
```



# Etapes d'utilisation d'un web service JAX-WS

## 1. Créer le service Web

- Développer le WS (avec les annotations JAX-WS)
- Déployer ou publier le WS:
  - Avec un serveur d'application JEE (Jboss,..)
  - ou avec un conteneur WS (JAXWS, AXIS,..)

## 2. Tester le service Web avec un analyseur SOAP

- Oxygen, SoapUI,...

## 3. Créer les clients

- Un client Java
- Un client .Net
- Un client php

# 1.1 Définir un WS dans une classe JAVA

© BanqueService.java ×

```
1 package org.soa.tp2;
2 import jakarta.jws.WebMethod;
3 import jakarta.jws.WebParam;
4 import jakarta.jws.WebService;
5 @WebService(serviceName = "BanqueWS")
6 public class BanqueService {
7     no usages
8     @WebMethod (operationName="ConversionEuroToDT")
9     public double conversion (@WebParam(name="montant") double mt)
10     { return mt*2.5; }
11     no usages
12     @WebMethod public String test()
13     { return "test"; }
14 }
```

**Indiquer qu'il s'agit d'un WS  
nommé« BanqueWS »  
par défaut, il prend le nom de la classe**

**Pour déclarer une  
méthode**

**Optionnelle pour  
personnaliser le  
nom du paramètre**


# Notion de skeleton – stub

- Un stub (souche) est un proxy pour un objet distant qui s'exécute sur l'ordinateur client.
- Un skeleton (squelette) est un proxy pour un objet distant qui s'exécute sur le serveur
- Le stub passe les invocations de méthode (et leurs arguments associés) aux squelettes, qui les transmettent au serveur approprié.
- Les squelettes renvoient les résultats de la méthode du serveur aux clients via le stub.



# Proxy stub et skeleton

- Un Stub : c'est en général un objet qu'on a en local mais qui ne contient aucun code.
- En fait il contient des descriptions de services que tu peux appeller.
- Il représente une copie d'un objet donc, mais qui est distant.
- le stub s'occupe d'appeler le véritable objet qui est distant



Un stub est un proxy côté client par opposition au skeleton qui est un proxy côté serveur.



# 1.2 Publier le WS à partir d'un serveur JAVA

© ServeurJaxws.java x

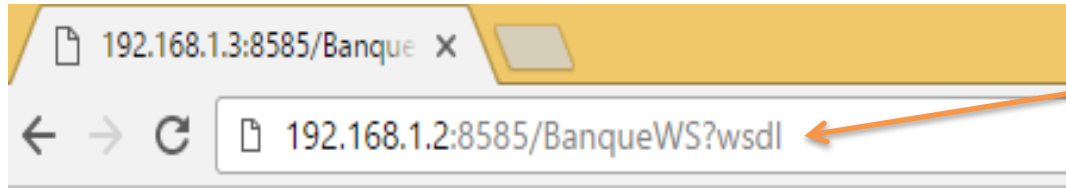
```
1 package org.soa.tp2;
2 import jakarta.xml.ws.Endpoint;
3 public class ServeurJaxws
4 {
5     public static void main(String[] args)
6     {
7         String adresseIp="172.16.40.22";
8         String url ="http://" + adresseIp + ":8585/";
9         // publier le service web
10        Endpoint.publish(url, new BanqueService());
11        System.out.println(url); }
12 }
13
```

**Classe Serveur**

**@IP Serveur + port**

**Créer un petit serveur web pour publier le WS, le serveur ouvre un service d'écoute et il attend qu'on appelle le WS**

# 1.4 Interroger le serveur via un navigateur Web



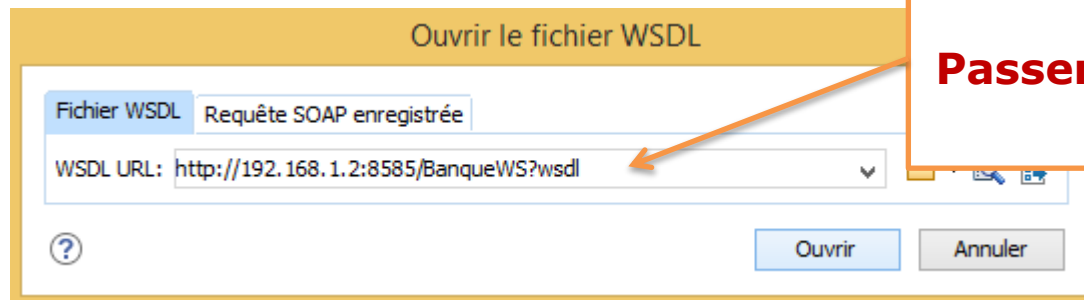
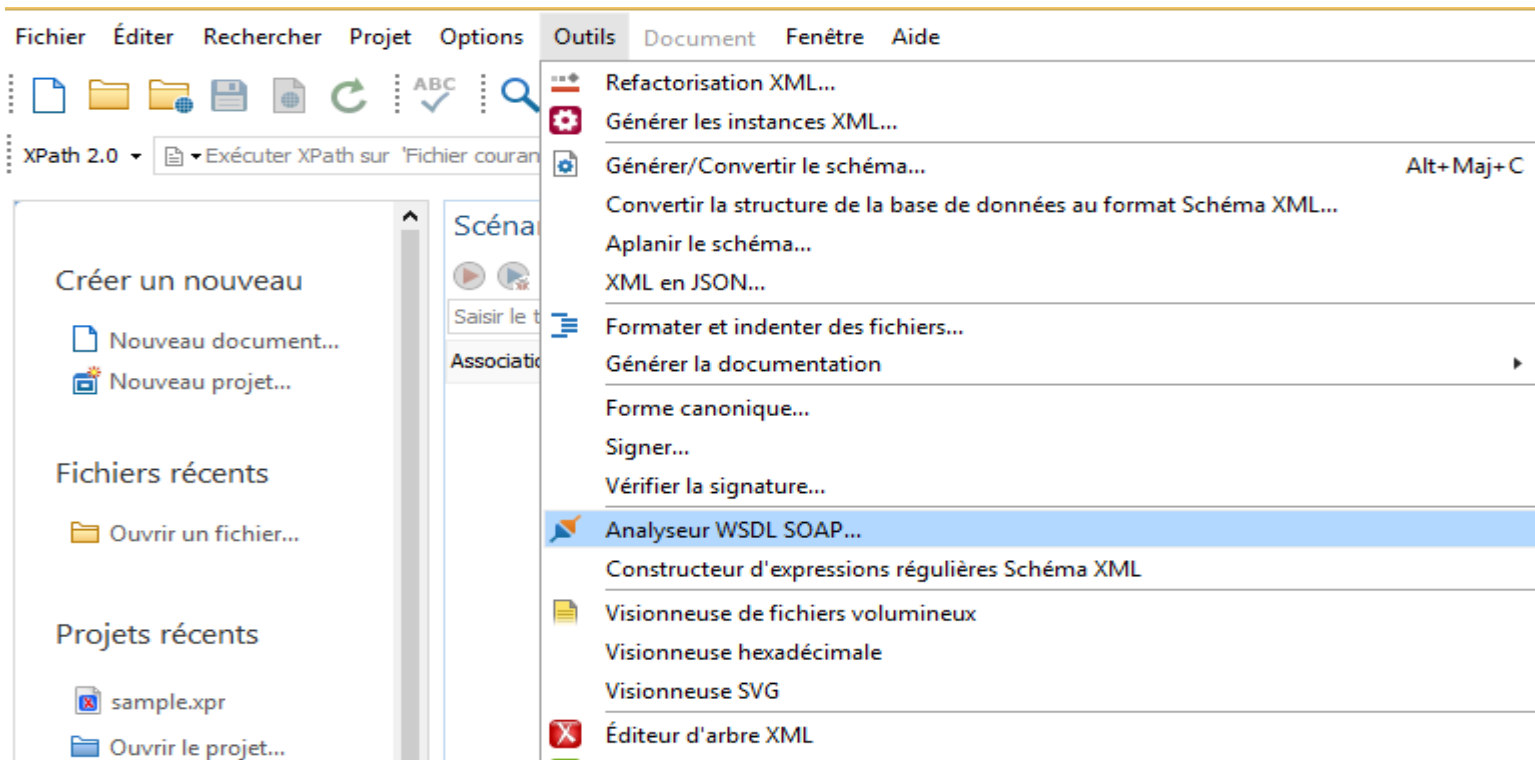
**Demander le fichier WSDL  
du WS qui représente  
l'interface du WS**

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<!--  
    Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.2.4-b01.  
-->  
▼<!--  
    Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.2.4-b01.  
-->  
▼<definitions xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0  
    xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy" xmlns:wsam="http://www.w3.org/2007/05/addr  
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://schemas.xmlsoap.org/wsdl/" targetNamespace=  
    ▼<types>  
        ▼<xsd:schema>  
            <xsd:import namespace="http://ws/" schemaLocation="http://1  
            </xsd:schema>  
        </types>  
    ▼<message name="ConversionEuroToDT">  
        <part name="parameters" element="tns:ConversionEuroToDT"/>  
    </message>  
    ▼<message name="ConversionEuroToDTResponse">  
        <part name="parameters" element="tns:ConversionEuroToDTResponse"/>  
    </message>
```

**Exemple:  
Méthode  
« ConversionEuroToDT »**

## 2. Tester les méthodes du WS via Oxygen



**Passer l'url du WSDL du WS**

## 2. Tester les méthodes du WS via Oxygen

### Analyseur WSDL SOAP

WSDL

Services: BanqueWS

Ports: BanqueServicePort

Opérations: ConversionEuroToDT

Actions

URL: http://192.168.1.2:8585/

Action SOAP:

Version: ☒ 1.1 ☐ 1.2

Requête Fichiers joints

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body>
    <ns0:ConversionEuroToDT xmlns:ns0="http://ws/">
      <montant>2</montant>
    </ns0:ConversionEuroToDT>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

**Passer la valeur 2 pour le montant**

Envoyer Paramètres de la requête: Ouvrir Enregistrer Régénérer

☐ Ouvrir la réponse dans l'éditeur

Réponse

```
<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:ConversionEuroToDTResponse xmlns:ns2="http://ws/">
      <return>5.0</return>
    </ns2:ConversionEuroToDTResponse>
```

**Recevoir le résultat: 5.0**

## 3.1. Créer une application Client JAVA

- Définir un plugin **jaxws** pour générer les proxys à partir du WSDL:

```
<plugin>
  <groupId>com.sun.xml.ws</groupId>
  <artifactId>jaxws-maven-plugin</artifactId>
  <version>4.0.1</version>
  <configuration>
    <wsdlUrls>
      <wsdlUrl>http://172.16.40.22:8585/BanqueWS?wsdl</wsdlUrl>
    </wsdlUrls>
    <packageName>org.soa.tp2</packageName>
    <sourceDestDir>
      ${project.build.sourceDirectory}/
    </sourceDestDir>
  </configuration>
</plugin>
```

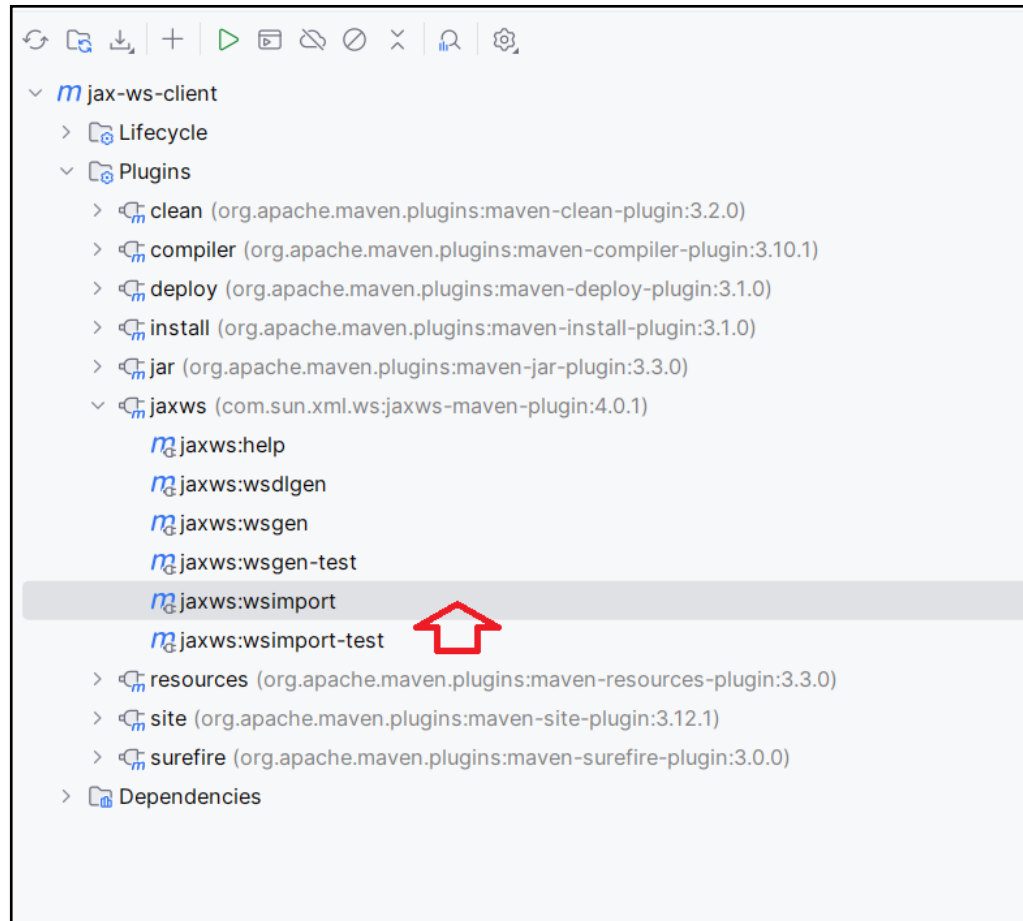
## 3.2. Créer une application Client JAVA

- Déclarer la dépendance de jakarta pour l'api JAX-WS:

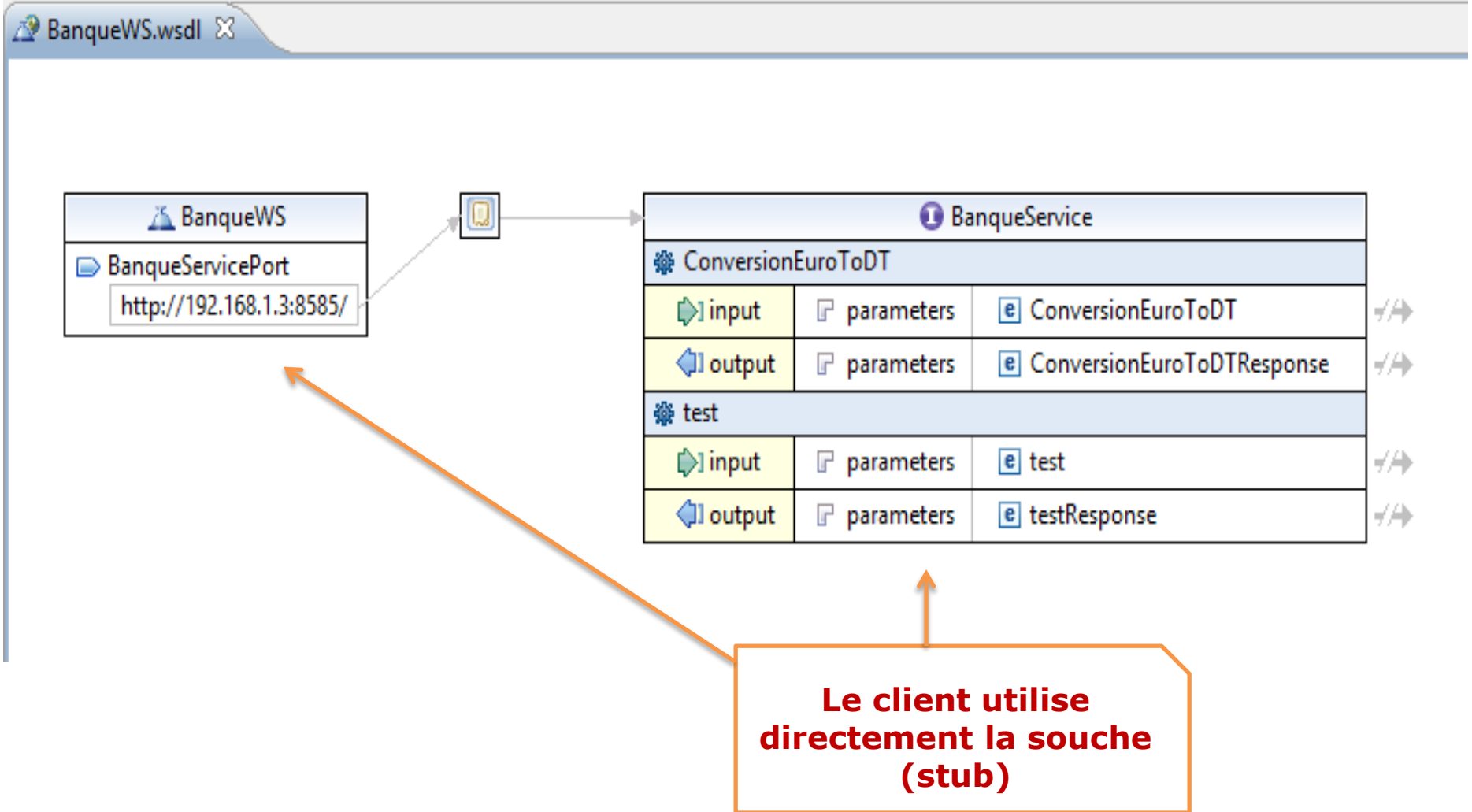
```
<dependency>  
  <groupId>jakarta.xml.ws</groupId>  
  <artifactId>jakarta.xml.ws-api</artifactId>  
  <version>4.0.0</version>  
</dependency>
```

## 3.3. Créer une application Client JAVA

- Générer les proxys côté client en utilisant la tâche « wsimport » du plugin « jaxws »:



## 3.2. Aperçu sur la souche (stub)





## 3.2 Consommer un WS par un client JAVA

ClientWS.java

```
package WS;

public class ClientWS
{
    public static void main(String[] args)
    {
        BanqueService stub = new BanqueWS().getBanqueServicePort();
        System.out.println(stub.conversionEuroToDT(50.0));
    }
}
```

**Invoker la méthode  
« conversionEuroToDT » via le stub**