Web services

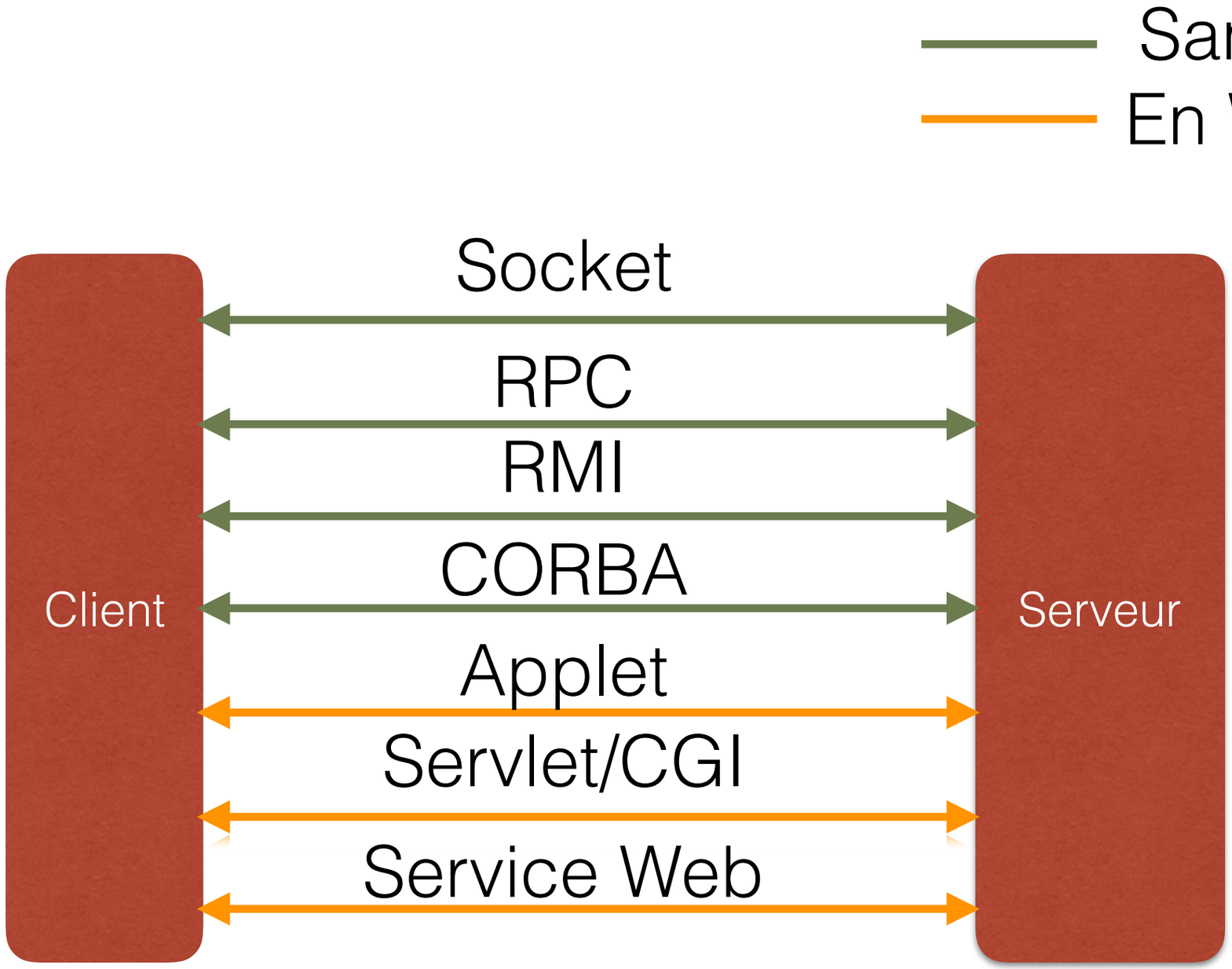
M.lahmer

SOA,ROA
XML
SOAP, WSDL, UUID
RPC / SOAP
AXIS,CXF JAXRS, JAXWS

Plan de la présentation

- Architecture répartie
- Introduction aux services Web
- Approche orienté service : SOA
  - SOAP
  - WSDL
  - UDDI
- Approche orienté ressource : ROA
  - Ressource
  - Méthodes
  - Format de représentation
- Les API java pour les services web

Architecture distribuée



Web Service

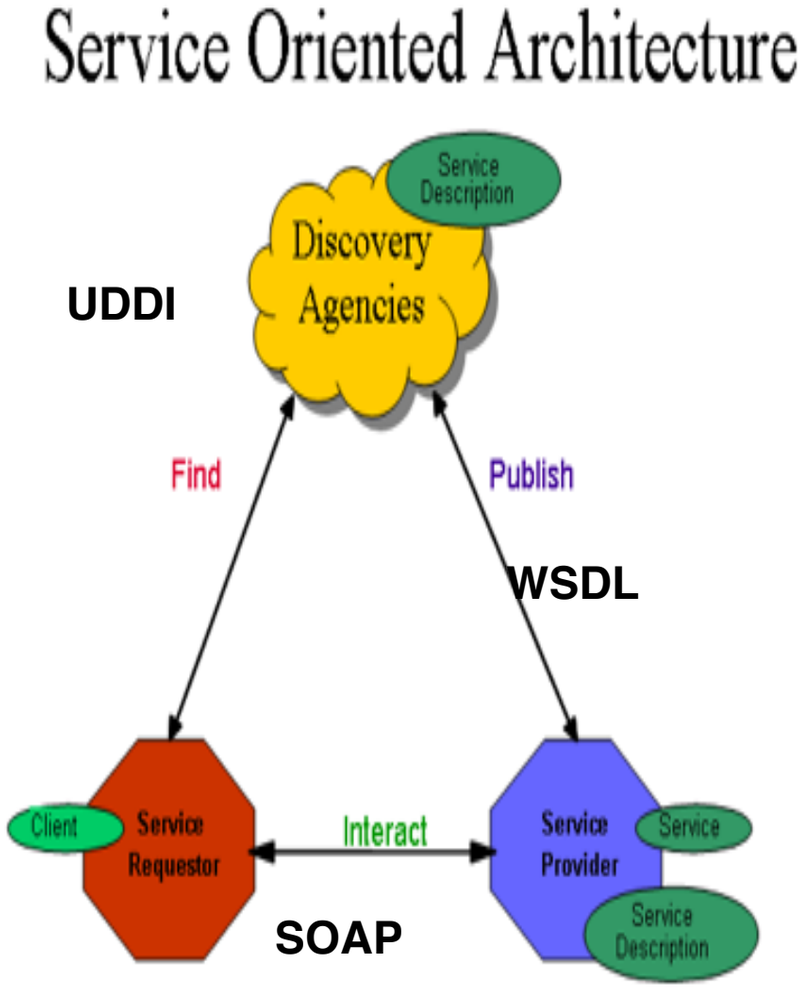
« A web service is a software application identified by a URI, whose interfaces and binding are capable of being defined, described, and discovered by XML artifacts, and supports direct interactions with other software applications using XML-based messages via Internet-based protocols »

Définition tiré du W3C
http://www.w3c.org/2002/ws/arch/2/08/wd-wsa-arch-20020821.html#webservice

- XML est utilisé pour décrire les messages échangés entre le client et le serveur
- Protocole de transport Internet (http, smtp, ftp, …)
- Deux Types d'architecture de services web
  - **Orienté service** SOA (standardisé)
  - **Orienté ressource** REST (non standardisé)

Architecture orientée service

- Un service résout un problème
- Les services peuvent coopérer pour résoudre un problème complexe
- Le client c'est celui qui invoque le service
- Le fournisseur c'est celui qui fournit le service
- L'annuaire c'est celui qui détient la description et les informations sur le service
- Le client et le serveur peuvent être développés par des langages différents



SOA Mode opératoire

- Un langage pour décrire le service web **WSDL** (Web Service Description Language)
- Un protocole pour structurer les messages échangés **SOAP** (Simple Object Access Protocol)
- Un protocole de Transport Standard pour transporter les messages
  - Http pour les messages synchrones
  - Smtp pour les messages asynchrones
- Un annuaire dans lequel est publié/localisé le service web **UDDI** (Universal Description Discovery and Integration)
  - il joue le même rôle que JNDI pour ejb, rmiRegistry pour mi ou cosNaming pour CORBA

WSDL

- C'est un langage de définition des interfaces des services (le contrat)
- Il représente la définition d'un services Web vue par le fournisseur
- Il doit contenir toutes les information nécessaire au client pour consommer le service (auto-suffisant)
- Utilise une grammaire XML de description d'interface des services
- Un service selon WSDL
  - Un ensemble d'opérations
  - Des formats des messages Typé nécessaire à chaque opération
- Un fichier WSDL contient
  - **les méthodes**
  - **les paramètres et les valeurs retournées**
  - **le protocole de transport utilisé**
  - **la localisation du service**

Types d'opération WSDL

Structure du WSDL

- **Définitions abstraites**
  - **Types**
    Un type décrit la structure de donnée transmise dans un message
  - **Messages**
    L'ensemble des données transmises au cours de l'opération (Requête / Réponse).
  - **PortTypes**
  - description des opérations du endpoint sous la forme d'échanges de messages. Ceci correspond à l'interface du service.
- **Descriptions concrètes**
  - **Bindings**
    Décrit la façon avec laquelle est lié un PortType à un Port selon un protocole réel
  - **Services**
    description des endpoints du service (binding et uri)

Les types WSDL

- **Un Type respecte le schéma XSD, il est utilisé lorsque la méthode retourne un objet**
- **Sinon on utilise les types xsd**

```
<types>
<xsd:schema targetNamespace="urn:xml-soap-address-demo"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<xsd:element name="to" type="my:mailformat" id="to"></xsd:element>
<xsd:element name="from" type="my:mailformat" id="from"></xsd:element>
<xsd:element name="mail">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="my:from" minOccurs="1" maxOccurs="1" />
<xsd:element ref="my:to" minOccurs="1" />
<xsd:element name="sujet" type="xsd:string" minOccurs="1" maxOccurs="1"></xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:simpleType name="mailformat">
<xsd:restriction base="xsd:string">
<xsd:pattern value="[0-9a-zA-Z]+[@][a-zA-Z]+\.[a-z]{3}">
</xsd:pattern>
</xsd:restriction>
</xsd:simpleType>
</types>
```

Exemple de fichier WSDL

- Prenant comme exemple le service **String sayHello(String nom)**

```
<definitions name="HelloService"
   targetNamespace="http://www.examples.com/wsdl/HelloService.wsdl"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://www.examples.com/wsdl/HelloService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <message name="SayHelloRequest">
      <part name="firstName" type="xsd:string"/>
   </message>

   <message name="SayHelloResponse">
      <part name="greeting" type="xsd:string"/>
   </message>
  <portType name="Hello_PortType">
      <operation name="sayHello">
         <input message="tns:SayHelloRequest"/>
         <output message="tns:SayHelloResponse"/>
      </operation>
   </portType>
```

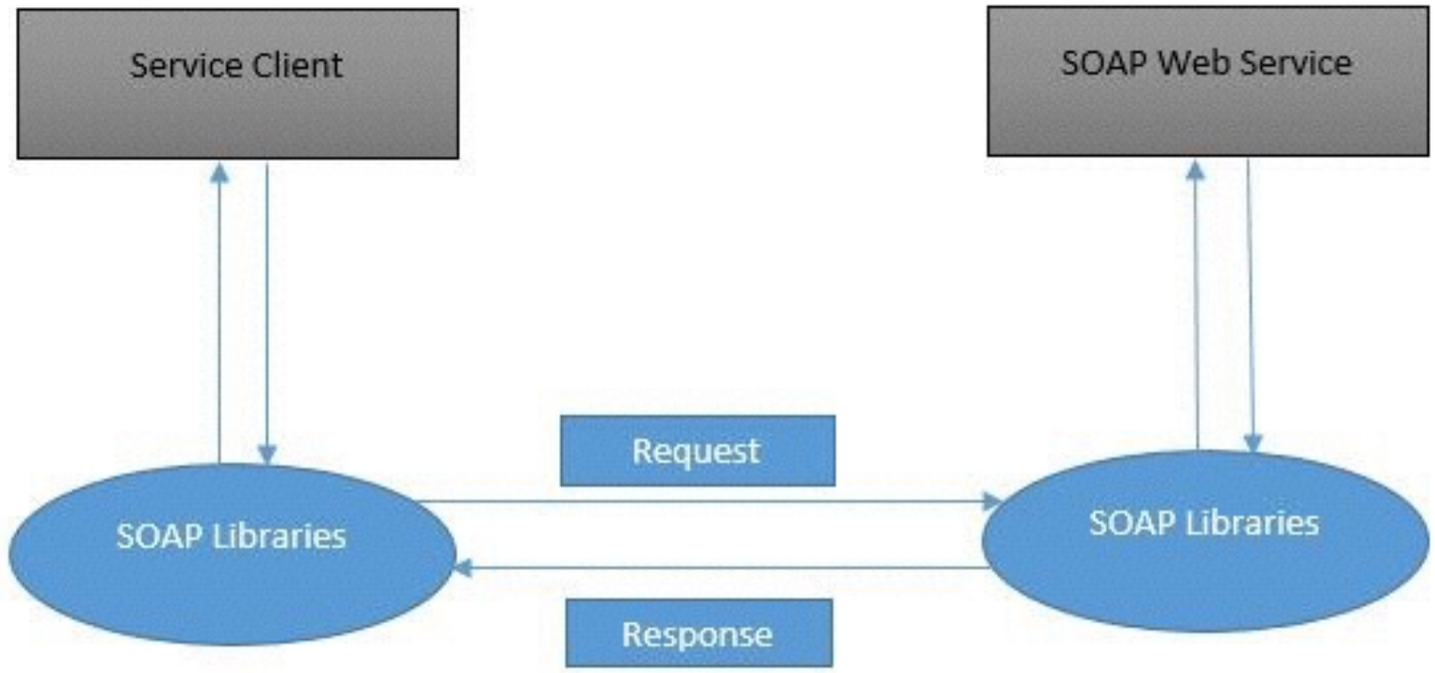
Exemple de fichier WSDL

```
<binding name="Hello_Binding" type="tns:Hello_PortType">
      <soap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="sayHello">
         <soap:operation soapAction="sayHello"/>
         <input>
            <soap:body
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="urn:examples:helloservice"
               use="encoded"/>
         </input>
         <output>
            <soap:body
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="urn:examples:helloservice"
               use="encoded"/>
         </output>
      </operation>
   </binding>
 <service name="Hello_Service">
      <documentation>WSDL File for HelloService</documentation>
      <port binding="tns:Hello_Binding" name="Hello_Port">
         <soap:address
            location="http://www.examples.com/SayHello/" />
      </port>
   </service>
</definitions>
```

SOAP

- Protocc
(Reque

- Les me
envelop



- Basé sur XML et indépendant tu protocole de transport utilisé mais http est le préféré

Enveloppe SOAP

```
<soap-env:Envelope
xmlns:soap-env="http://www.w3.org/2001/12/soap-envelope"
soap:soap-enc="http://www.w3.org/2001/12/soap-encoding">
    <soap-env:Header>
       <!—Optionnelle sert comme description —>
    </soap-env:Header>

    <soap-env:Body xmlns:myns="http://www.exemple.org/
services">
       <!—Le message—>
    </soap-env:Body>

     <soap-env:Fault>
       <!—Erreur —>
    </soap-env:Fault>
</soap-env:Envelope>
```

Styles SOAP

- Il existe deux styles de services web reposant sur SOAP : RCP et Document
- Pour chaque style il existe deux types d'encodage de messages : Encoded et Literal
- Dans le style RPC/Encoded seules les types de base XSD sont utilisés pour coder les messages
- Dans le style Document/Literal chaque élément qui correspond à un paramètre ou à la valeur de retour est décrit dans un schéma XML.
- Les messages de type RPC/Literal sont encodés comme des appels RPC avec une description des paramètres et des valeurs de retour décrites chacune avec son propre schéma XML
- Les combinaison utilisées sont RPC/Encoded ou Document/Literal

SOAP Document/Literal

le service est `String sayHello(String Name)`

- Request SOAP

```
<SOAP-ENV:Body xmlns:m=« http://www.exemple.org/hello" >

      <m:sayHello>
         <m:Name>ESTM<m:Name>
      </m:sayHello>

</SOAP-ENV:Body>
```

- Response SOAP

```
<SOAP-ENV:Body xmlns:m="http://www.exemple.org/hello" >

      <m:sayHelloResponse>
         <m:Response>Hello ESTM</m:Response>
      </m:sayHelloResponse>

   </SOAP-ENV:Body>
```

SOAP RPC/Encoded

Le service est `int add(int a, int b)`

- Request SOAP

```
<SOAP-ENV:Body xmlns:m=« http://www.exemple.org/hello" >

    <m:add soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/
    encoding/">
          <m:a xsi:type="xsd:int">10<m:a>
           <m:b xsi:type="xsd:int">20<m:b>
   </m:add>

</SOAP-ENV:Body>
```

- Response SOAP

```
<SOAP-ENV:Body xmlns:m="http://www.exemple.org/hello" >

      <m:addResponse>
         <m:Response xsi:type="xsd:int">30</m:Response>
      </m:addResponse>
</SOAP-ENV:Body>
```

APIs Java pour SOAP

- JAX-WS Java for XML Web Service, successeur de JAX-RPC est une implémentation sun standardisée
- Intégrée dans JDK 1.6 et donc pas besoin de conteneur
    - SAAJ (SOAP with Attachment API for Java) : permet l'envoi et la réception de messages respectant les normes SOAP et SOAP with Attachment
    - JAXB pour automatiser le mapping d'un document XML objets Java
- AXIS 1 et 2 de apache
- CXF implémentation apache pour SOA et ROA
- JBossWS implémentation IBM

JAX-WS

- Développement basé sur POJO (Plain Old Java Object)
- Utilisation des annotations pour la déclaration du service et Style SOAP
- Un service web peut être développé à partir
  - D'un fichier wsdl et on utilise l'outils wsimport pour générer les classes
  - D'un POJO avec annotation, le fichier wsdl est automatiquement généré (wsdlgen)

JAX-WS Server sans conteneur

```java
package estm.ws.demo;
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;
@WebService
@SOAPBinding(style=Style.RPC)
public interface Hello {
    @WebMethod
    public String sayHello(String arg);
}
```

```java
package estm.ws.demo;
import javax.jws.WebService;
@WebService(endpointInterface="estm.ws.demo.Hello")
public class HelloImpl implements Hello {
    @Override
    public String sayHello(String arg) {
       // TODO Auto-generated method stub
       return "hello "+arg;
    }}
```

JAX-WS Client sans wsimport

```
import java.net.MalformedURLException;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import estm.ws.demo.Hello;

public class Main {

   public static void main(String[] args) throws Exception {
      // TODO Auto-generated method stub
   URL url=new URL("http://localhost:9090/ws/hello?wsdl");
   QName qname = new QName("http://demo.ws.estm/",
"HelloImplService");
      Service service=Service.create(url, qname);
      Hello hello=service.getPort(Hello.class);
      System.out.println(hello.sayHello("Lahmer"));
   }
}
```

JAX-WS Client avec wsimport

```
wsimport -keep http://localhost:9090/ws/hello?wsdl -d $HomeClient/
src
```

```
import java.net.MalformedURLException;
import java.net.URL;

public class Main {

   public static void main(String[] args) throws Exception {
      // TODO Auto-generated method stub
      HelloImplService service=new HelloImplService()
      Hello hello=service.getHelloImplPort();
      System.out.println(hello.sayHello("Lahmer"));
   }
}
```

JAX-WS avec conteneur

- Pour utiliser jax-ws, il faut ajouter dans **web.xml** la servlet et et le Listener
  - com.sun.xml.ws.transport.http.servlet.WSServlet
  - com.sun.xml.ws.transport.http.servlet.WSServletContextListener

```
<listener>
  <listener-class>
     com.sun.xml.ws.transport.http.servlet.WSServletContextListener
  </listener-class>
</listener>
<servlet>
   <servlet-name>hello</servlet-name>
   <servlet-class>
     com.sun.xml.ws.transport.http.servlet.WSServlet
   </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>hello</servlet-name>
  <url-pattern>/hello</url-pattern>
</servlet-mapping>
```

JAX-WS avec conteneur

- On suppose que le service web est définie par l'interface Hello et est implémenter par HelloImpl dans le package estm.demo
- Créer le fichier de mapping ***sun-jaxws.xml*** dans le dossier *"WEB-INF"* de la webapp

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime"
version="2.0">
  <endpoint
     name="HelloWorld"
     implementation="estm.demo.HelloImpl"
     url-pattern="/hello"/>
</endpoints>
```

JAX-WS avec conteneur

```
Client jax-ws Servlet
@WebServiceRef(wsdlLocation="http://localhost:8080/hello?wsdl")
@WebServlet("/jaxwsClient")
public class jaxwsClient extends HttpServlet {
```

```
   private static final long serialVersionUID = 1L;
      protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {

      HelloImplService sw=new HelloImplService();
      Hello h=sw.getHelloImplPort();
response.getWriter().append("Served at: « +h.sayHello("estm"));

   }
```

Axis 2

- Apache eXtensible Interaction System : Solution apache développée en java qui supporte aussi bien les services web SOAP (SOAP 1.2 & WSDL 2.0) que REST

- Supporte les services web synchrone et asynchrone (bloquant/ non bloquant)

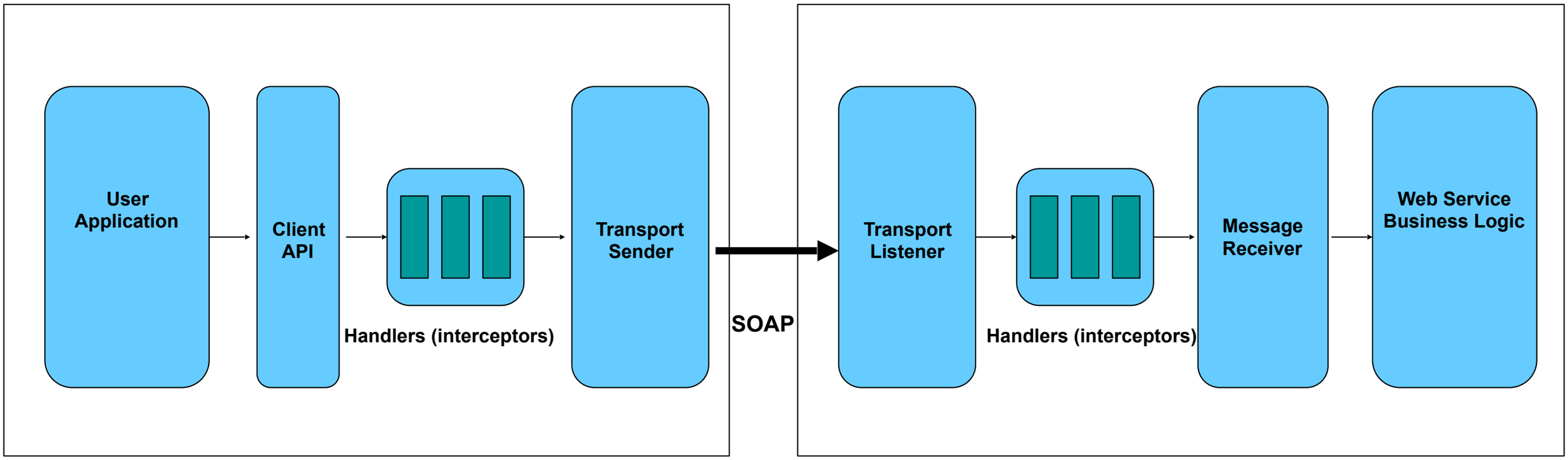
- Les styles SOAP rpc/encoded ou Document/encoded ne sont plus supportés

- Supporte une large implémentations pour le mapping xsd<->beans java parmi lesquelles XMLBean, ADB et JAXB

- Possède des outils en mode commande WSDL2Java et Java2WSDL

- Offre le Monitoring TCP et SOAP

Axis 2 Architecture



- Axis 2 peut traiter les messages SOAP aussi bien pour le client (émetteur) que le serveur (récepteur)

- le client génère le message soap, le Handler peut ajouter des actions supplémentaires telles que le cryptage avant de le soumettre au transport sender qui va l'envoyer.

- Le Listener détecte l'arrivée du message le soumet au Handler. Une fois traité il le message est envoyé au Dispatcher qui l'aiguille vers l'application concernée.

Axis 2 le Serveur

On crée l'interface POJI et POJO : Implservice.java Iservice.java Dans le projet Web dynamic.
Générer le wsdl par eclipse ou en utilisant java2wsdl

```
public interface Iservice {
public String sayHello(String lang);
}

public class Implservice implements Iservice {

   @Override
   public String sayHello(String lang) {
      return iff lang. equals(«ar »)?»SALAM »: » Hello»;
   }
}
```

Axis 2 Client lourds

```
wsdl2java.sh -uri http://localhost:8080/axis2Server/services/
Implservice?wsdl -u -o $CLientProjet
-->
ExtensionMapper.java
ImplserviceService.java
ImplserviceServiceCallbackHandler.java
ImplserviceServiceStub.java
SayHello.java
SayHelloResponse.java
void main(String[] args) throws Exception {
   ImplserviceServiceStub service=new ImplserviceServiceStub();
ImplserviceServiceStub.SayHello request=new
ImplserviceServiceStub.SayHello();
   request.setLang("ar");
   ImplserviceServiceStub.SayHelloResponse
response=service.sayHello(request);
   System.out.println(response.getSayHelloReturn());
   }
```

Axis 2 Client léger

Les fichiers générés depuis eclipse

Implservice.java

ImplserviceProxy.java

ImplserviceService.java

ImplserviceServiceLocator.java

ImplserviceSoapBindingStub.java

La page jsp

```
<%
ImplserviceProxy stub=new ImplserviceProxy();
%>
<%=stub.sayHello("ar") %>
```

Ressource Oriented Web Service REST

- Les services REST sont utilisés pour développer des architectures orientées ressources (ROA)

- REST est l'acronyme de REpresentational State Transfert

- Concept introduit en 2000 dans la thèse de Roy FIELDING

- Ce n'est pas un format pas un protocole est non plus un standard

- Permet l'envoi de messages sans enveloppe SOAP et dans un encodage libre (XML, JSON, binaire, simple texte)

- Une Ressource « Any information that can be named can be a resource »

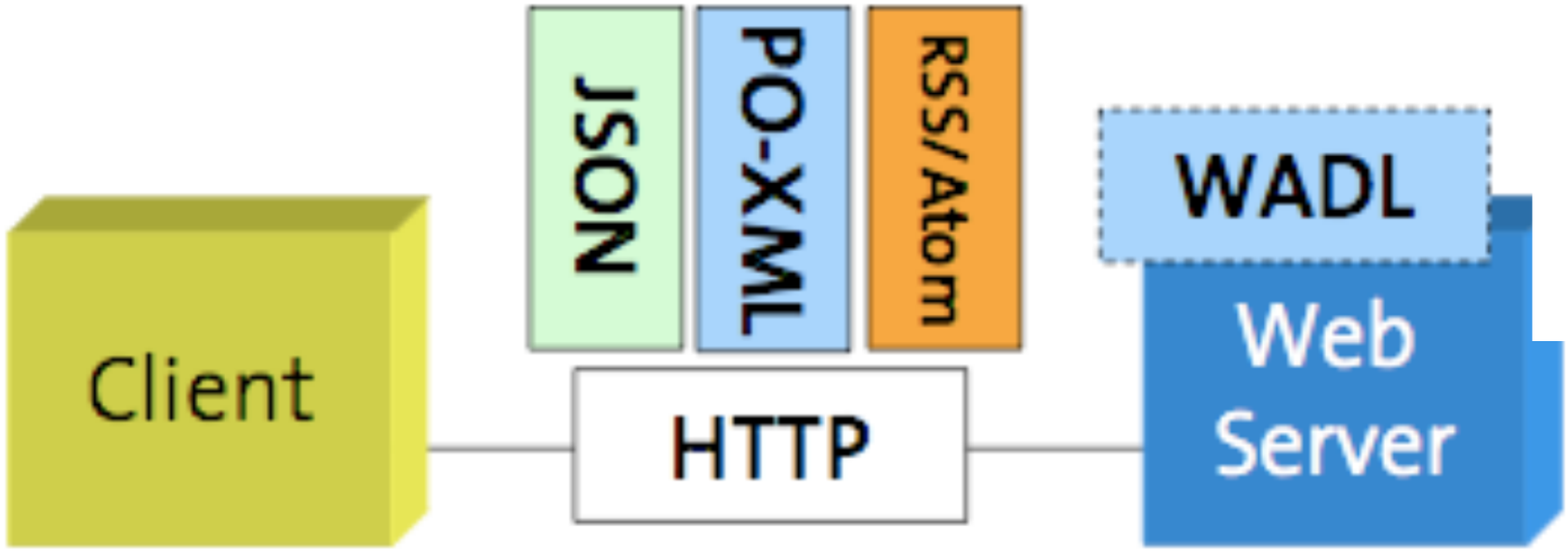
REST principe

- Chaque Ressource est identifiée par une URI http://api.domain.com/users
- L'accès à une ressource se fait par l'une des méthodes HTTP:
    - GET : Récupérer une ressource http://api.domain.com/users/lahmer
    - POST : Ajouter une ressource http://api.domain.com/users/estm
    - PUT : Modifier une ressource http://api.domain.com/users/estm?nom=new&..
    - DELETE : Supprimer une ressource http://api.domain.com/users/estm/
- Les données retournées peuvent prendre plusieurs format
    - Text
    - XML
    - JSON
    - Binaire

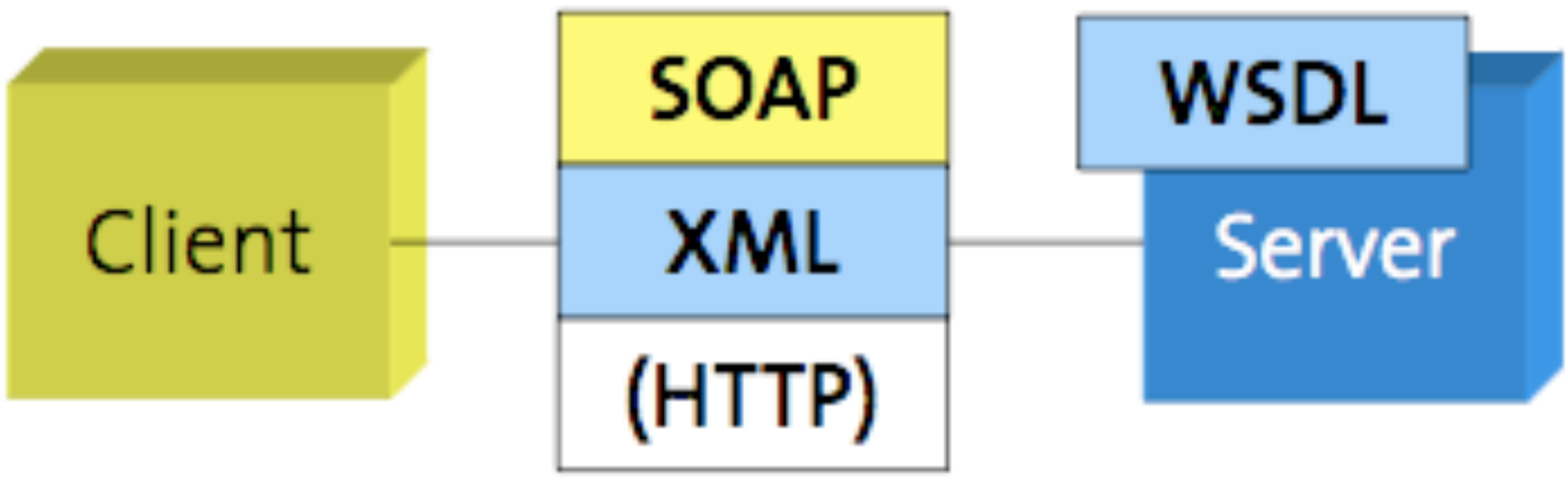
REST vs. SOAP

REST vs. SOAP

RESTful Web Services (2007)



WS-* Web Services (2000)



WADL

- WADL(Web Application Description Language) langage de description XML de services de type REST
- Permet une description de services par éléments de type: ressource, méthode, paramètre, requête, réponse

<application>

<doc jersey:generatedBy="Jersey: 2.22.2 2016-02-16 13:32:17 »/>

<doc jersey:hint="This is simplified WADL with user and core resources only. To get full WADL with extended resources use the query parameter detail. Link: http://localhost:8080/jaxrs/rest/application.wadl?detail=true »/><grammars/>

<resources base=« http://localhost:8080/jaxrs/rest/« >

<resource path="/services">

<method id="getHello" name=« GET">

<response>**<representation mediaType="text/plain"/>**</response>

</method>**</resource></resources></application>**

Java API pour Restful Web Services

- **Oracle Jersey 2.0:** framework pour faire des services web RESTful
- **RestEasy:** Projet Jboss pour la création des services web.
- **Apache CXF**
- **Restlet:** un des premiers framework implémentant REST pour Java
- API JAX-RS : Java API for Restful Web Services
- Interface de programmation Java permettant de créer des services Web avec une architecture REST.
- Fournie avec Java EE 6 https://jax-rs-spec.java.net/
- Version: dernière version JAX-RS 2.0

Rest en JAX-RS Côté Serveur

- On crée un projet JEE web dynamic puis on inclut dans le lib les jars jax-rs.

- Dans le web.xml, on ajoute la servlet org.glassfish.jersey.servlet.ServletContainer

```
<servlet>
  <servlet-name>RestServlet</servlet-name>
  <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
   <init-param>
     <param-name>javax.ws.rs.Application</param-name>
         <param-value>estm.jaxrs.rest</param-value>
          </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
   <servlet-mapping>
 <servlet-name>RestServlet</servlet-name>
<url-pattern>/rest/*</url-pattern>
 </servlet-mapping>
```

Rest en JAX-RS Côté Serveur

- Créer la classe service POJO avec annotation @GET, @POST, @PUT, @DELETE

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
@Path("/services")
public class jaxrsServices {
@GET
@Produces(value="text/plain")
public String getHello()
{
    return "SALAM";}}
```

- Créer la classe de configuration à partir de Jersey 2.0

```
import org.glassfish.jersey.server.ResourceConfig;
public class rest extends ResourceConfig {
  public rest()
  {
      packages("estm.jaxrs");
  }
}
```

Rest en JAX-RS Côté Client

- Plusieurs types de clients peuvent être implémentés en utilisant
  - HttpURLConnection (java.net)
  - HttpClient de apache org.apache.http.impl.client
  - jax-rs client javax.ws.rs.client.Client;

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

   Client client=ClientBuilder.newClient();
  WebTarget target = client.target("http://localhost:8080/jaxrs/rest/
services/");
        Response response = target.request().buildGet().invoke();
   System.out.println(response.readEntity(String.class));
   }
```

Annotations Jax-rs

@QueryParam

@QueryParam

Permet d'associer un paramètre de la requête à un champ ou un paramètre d'une méthode

```
@DefaultValue("defaultValue")
```

Permet de spécifier la valeur par défaut

```
@GET
@Path("/json")
@Produces(MediaType.APPLICATION_JSON)
public Client
getjsonHello(@DefaultValue("lahmer")@QueryParam("nom") String
nom){
   return L.get(nom);
}
```

L'accès se fait par URL: http://machine/rest/json?nom=lahmer

Annotations Jax-rs

@PathParam

@PathParam

Dans l'annotation « *@Path* », il est possible de mettre des parties variables entre accolades {}. Ils peuvent alors être récupérés comme paramètres de la méthode avec l'annotation « ***@PathParam*** »

```
@DELETE
@Path("/delete/{userId}")
public boolean deleteUser(@PathParam("userId") String
userId)
{..}
```

L'accès se fait par URL: http://machine/rest/delete/1

Annotations Jax-rs

@FormParam

@FormParam

Permet de récupérer directement les paramètres depuis un formulaire html.

<form action="add" methode="post">
<input name="nom" typte=text/>
....
</form>

```
@POST
@Path("/add/")
public boolean deleteUser(@FormParam("nom") String
nom)
{..}
```

URL: http://machine/rest/add