WEB SERVICES

version 0.2 (2016)

Pascal Poizat
Université Paris Ouest Nanterre la Défense
http://lip6.fr/Pascal.Poizat

I. Démarrage en douceur

1. pré-requis

- Java >= 1.6 (nous utiliserons la version 1.8)
- un IDE (IntelliJ, Eclipse ou NetBeans)

2. création d'un service simple

On commence par créer une interface décrivant le service. On retrouve l'idée des interfaces utilisées dans RMI.

```
package fr.paris10.miage.procs.exercice1;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

@WebService
@SOAPBinding(style = SOAPBinding.Style.RPC)
public interface HelloWorld {
    @WebMethod String bonjour(String name);
}
```

Noter ici les annotations :

- `@WebService` pour signaler que l'interface est une description (d'interface) de service
- `@SOAPBinding` pour préciser le type de *binding* SOAP utilisé (ici RPC)

Dans un second temps on crée une implantation de l'interface. Cela ressemble toujours beaucoup à RMI. Notez l'utilisation de l'annotation `WebService` pour donner l'interface implantée.

```
package hello;

import javax.jws.WebService;

@WebService(endpointInterface = "fr.paris10.miage.procs.exercice1.HelloWorld")
public class HelloWorldImpl implements HelloWorld {
    @Override
    public String bonjour(String name) {
```

```
        return "Bonjour "+name+"\n";
    }
}
```

Enfin, il faut lancer le service. Pour l'instant on utilise un lanceur (publisher).

```
package hello;

import javax.xml.ws.Endpoint;

public class HelloWorldLanceur {
    public static void main(String[] args) {
        Endpoint.publish("http://localhost:9999/ws/bonjour", new HelloWorldImpl());
    }
}
```

Un premier test. Il suffit d'accèder à l'URI http://localhost:9999/ws/bonjour?wsdl, ce qui peut être fait avec un navigateur ou bien la commande `curl` du shell.

Si tout va bien vous récupérez ceci :

```
<?xml version="1.0" encoding="UTF-8"?><!-- Published by JAX-WS RI
(http://jax-ws.java.net). RI's version is JAX-WS RI 2.2.9-b130926.1035
svn-revision#5f6196f2b90e9460065a4c2f4e30e065b245e51e. -->
<!-- Generated by JAX-WS RI (http://jax-ws.java.net). RI's version is JAX-WS RI
2.2.9-b130926.1035 svn-revision#5f6196f2b90e9460065a4c2f4e30e065b245e51e. -->
<definitions
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utili
ty-1.0.xsd" xmlns:wsp="http://www.w3.org/ns/ws-policy"
xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://exercice1.procs.miage.paris10.fr/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://exercice1.procs.miage.paris10.fr/"
name="HelloWorldImplService">
<types></types>
<message name="bonjour">
<part name="arg0" type="xsd:string"></part>
</message>
<message name="bonjourResponse">
<part name="return" type="xsd:string"></part>
</message>
<portType name="HelloWorld">
<operation name="bonjour">
<input
wsam:Action="http://exercice1.procs.miage.paris10.fr/HelloWorld/bonjourRequest"
message="tns:bonjour"></input>
<output
wsam:Action="http://exercice1.procs.miage.paris10.fr/HelloWorld/bonjourResponse"
message="tns:bonjourResponse"></output>
</operation>
```

```
</portType>
<binding name="HelloWorldImplPortBinding" type="tns:HelloWorld">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http"
style="rpc"></soap:binding>
<operation name="bonjour">
<soap:operation soapAction=""></soap:operation>
<input>
<soap:body use="literal"
namespace="http://exercice1.procs.miage.paris10.fr/"></soap:body>
</input>
<output>
<soap:body use="literal"
namespace="http://exercice1.procs.miage.paris10.fr/"></soap:body>
</output>
</operation>
</binding>
<service name="HelloWorldImplService">
<port name="HelloWorldImplPort" binding="tns:HelloWorldImplPortBinding">
<soap:address location="http://localhost:9999/ws/bonjour"></soap:address>
</port>
</service>
</definitions>
```

Il s'agit d'un fichier WSDL (Web Service Description Language), dans sa version 1.1, nous verrons son rôle et son contenu dans la suite.

IMPORTANT : n'oubliez pas de vous assurez que le service est lancé avant de l'utiliser et de l'arrêter lorsque vous n'en avez plus besoin.

3. création d'un client simple

La création d'un client est simplifiée grâce aux informations contenues dans le fichier WSDL que nous avons vu ci-dessus. Traditionnellement on peut soit utiliser une copie locale (au niveau du code client) de ce fichier, soit l'accèder en ligne (si le service Web tourne, il est possible d'ajouter `?wsdl` à son URI pour obtenir le fichier WSDL le décrivant). Côté classes Java, ici le client a directement accès à la description du service (l'interface `hello.HelloWorld`). Nous verrons dans la suite comment lever cette hypothèse.

```
package helloclient;

import hello.HelloWorld;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import java.net.URL;

public class HelloWorldClient {

    public static void main(String[] args) throws Exception {
        URL url = new URL("http://localhost:9999/ws/bonjour?wsdl");
        QName qname = new QName("http://hello/", "HelloWorldImplService");
        Service service = Service.create(url, qname);
```

```
        HelloWorld hello = service.getPort(HelloWorld.class);
        System.out.println(hello.bonjour("Bob"));
    }
}
```

Ici Java nous aide avec des classes qui font le gros du travail (c'est l'intergiciel) pour nous : `URL` (représentation d'une URL), `QName`(noms qualifiés tels que décrits dans la spécification XML, réutilisée par les spécifications de services Web SOAP/WSDL) et `Service`(accès à un service Web en indiquant l'URL d'un fichier WSDL le décrivant et un nom qualifié de service).

4. auto-évaluation

1. Crééz un service Web qui permette de faire des additions et des soustractions
2. Faites un programme de test pour ce service. Vous pouvez utiliser ou vous inspirer du squelette TestNG suivant.

```
package calculette;

import org.testng.annotations.*;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;
import java.net.URL;

import static org.testng.Assert.*;

public class CalculetteTest {

    private Calculette calc;

    @BeforeClass
    public void setUp() throws Exception {
      // à compléter
    }

    @AfterClass
    public void tearDown() throws Exception {
        calc = null;
    }

    @Test
    public void testAjouter() throws Exception {
        assertEquals(calc.ajouter(2,3),5);
    }

    @Test
    public void testSoustraire() throws Exception {
        assertEquals(calc.soustraire(2,3),-1);
    }
}
```

II. Développement bottom-up

On part ici de code Java qu'on expose ensuite comme service Web. Pour comprendre cela, nous allons partir d'une étude de cas relativement simple : celle d'un système automatisé d'amendes (SAAM).

1. SAAM - Classes de données

Les données manipulées par le service sont de trois types :

- types de base Java : rien de particulier à faire
- classes Java internes à l'implantation du service (pas en E/S des opérations exportées par le service) : rien de particulier à faire
- classes Java utilisées par l'interface du service : ces classes doivent pouvoir être sérialisées (en XML, nous verrons comment par la suite), il faut donc s'assurer que cela est possible en ayant des constructeurs sans paramètres, des getters et/ou des setters (on verra pourquoi par la suite). De plus, les clients devront y avoir accès (pour compiler), ce qui peut être fait en les mettant en ligne telles quelles ou dans une archive (jar, war, etc) et en utilisant l'option -cp de java/javac.

A noter, qu'ici on ne parle pas de données qui seraient elles-même des services (ce qu'on verra éventuellement dans par la suite aussi).

1. crééz les classes `Personne`, `Voiture` et `Amende`.

```
package radars;

import java.io.Serializable;

public class Personne implements Serializable {
    private String nom;
    private String prenom;
    private String adresse;

    public Personne() { super(); }

    public Personne(String nom, String prenom, String adresse) {
        this.nom = nom;
        this.prenom = prenom;
        this.adresse = adresse;
    }

    public String getNom() { return nom; }
    public String getPrenom() { return prenom; }
    public String getAdresse() { return adresse; }

    public void setNom(String nom) { this.nom = nom; }
    public void setPrenom(String prenom) { this.prenom = prenom; }
    public void setAdresse(String adresse) { this.adresse = adresse; }

    @Override
    public String toString() { return nom + " " + prenom; }
```

```
}
```

```
package radars;

import java.io.Serializable;

public class Voiture implements Serializable {
    private String immatriculation;
    private String modele;
    private Personne proprietaire;

    public Voiture() { super(); }

    public Voiture(String immatriculation, String modele) {
        this.immatriculation = immatriculation;
        this.modele = modele;
        this.proprietaire = null;
    }
    public void setProprietaire(Personne p) { this.proprietaire = p; }
    public void setImmatriculation(String immatriculation) { this.immatriculation =
immatriculation; }
    public void setModele(String modele) { this.modele = modele; }

    public String getImmatriculation() { return immatriculation; }
    public String getModele() { return modele; }
    public Personne getProprietaire() { return proprietaire; }

    @Override
    public String toString() {
        String rtr = modele + " " + immatriculation;
        if(proprietaire != null) {
            rtr += "\n" + "possédée par " + proprietaire;
        }
        else {
            rtr += "\n" + "pas de propriétaire connu";
        }
        return rtr;
    }
}
```

```
package radars;

import java.io.Serializable;

public class Amende implements Serializable {
    private static int numero_ = 0;
    private int numero;
    private String immatriculation;
    private int tarif;

    public Amende() { super(); }

    public Amende(String immatriculation, int tarif) {
        numero_++;
```

```
        numero = numero_;
        this.immatriculation = immatriculation;
        this.tarif = tarif;
    }

    public int getNumero() {
        return numero;
    }
    public String getImmatriculation() {
        return immatriculation;
    }
    public int getTarif() { return tarif; }

    public void setNumero(int numero) { this.numero = numero; }
    public void setImmatriculation(String immatriculation) { this.immatriculation =
immatriculation; }
    public void setTarif(int tarif) { this.tarif = tarif; }

    @Override
    public String toString() {
        return String.format("Amende n°%d de %d EUR pour
%s",numero,tarif,immatriculation);
    }
}
```

2. SAAM - Service

1. commencez par écrire l'interface du service, `SystemeAmendes`contraintes: binding = SOAP RPC opérations =
   - **enregistrer(in v:Voiture)**, qui permet d'enregistrer une voiture,
   - **signaler(in immatriculation:string, in modele:string, in tarif:int):int**, qui permet de signaler une faute et qui renvoit le numéro d'amende (-1 si erreur),
   - **lister(in immatriculation:string): Amende[]**, qui permet d'obtenir la liste des amendes (non payées) d'une voiture,
   - **payer(in numero:int, in nom:string, in prenom:string)**, qui permet de payer une amende
2. écrivez une implantation de ce service, `SystemeAmendesImpl`
3. écriver un lanceur pour le service puis vérifiez avec le lien http://localhost:9999/ws/saam?wsdl son descriptif WSDL.Le fichier WSDL contient les informations nécessaires à la descrption d'un service, pour la version 1.1 :
   - les types de données complexes (ne correspondant pas aux types de base Java traduits en types de base XML) sont définis dans le fichier WSDL directement ou via l'importation de schémas externes (XSD), ici http://localhost:9999/ws/saam?xsd=1
   - les `message`[1] sont des types de données (XML) qui permettent d'échanger des informations entre client et serveur. Il sont constitués de parties nommées et typées (au sens XML)
   - les `operation` sont les opérations disponibles pour les clients. On leur associe (selon les paramètres) des messages en entrée et/ou en sortie
   - les `portType` sont des ports où un ensemble d'opérations sont disponibles
4. l'opération `lister` utilise un tableau pour retourner la liste des amendes non payées. On aurait pu imaginer utiliser `java.util.List`. Modifiez votre programme en conséquence. Qu'observez-vous ?

Pour plus de détails sur les normes vues juqu'ici :

- WSDL 1.1, WSDL 2.0
- SOAP 1.2
- XSD 1.1 partie 1 et XSD 1.1 partie 2

IMPORTANT : nous ne pouvons pas rentrer ici dans les détails de ces normes mais retenez un point clé : il est courant que telle ou telle version d'un protocole ou d'un framework (comme JAXB qui sert pour faire le lien etre représentation XML des données et Java) ne soit pas compatible avec telle ou telle version d'un autre protocole ou outil. Assurez vous bien quand vous développez des services d'avoir une compatiblité entre les versions de XSD (représentation des données), SOAP (protocole d'appel de méthode objet par échange de messages XML), WSDL (description de service), JAX-WS RI/JAXB (traductions Java <-> XML), pile de service utilisée (spécification: Jax-RPC/Jax-WS ou implantation: Axis1/Axis2/CXF/JBoss) et serveur d'application associé (Tomcat ou JBoss par exemple).

3. SAAM - Client

1. écrivez un client ou un testeur pour votre service Web. Les actions à faire sont les suivantes:
    1. enregistrement d'une Clio 3 immatriculée AB 123 CD possédée par Bob Sponge
    2. enregistrement d'un Punto immatriculée AB 124 CD possédée par Capitaine Haddock
    3. signalement d'une infraction à 90€ pour une Clio 3 immatriculée AB 123 CD (renvoit numéro n1)
    4. signalement d'une infraction à 90€ pour un Tracteur immatriculé AB 123 CD (fausse plaque, renvoit -1)
    5. signalement d'une infraction à 140€ pour une Clio 3 immatriculée AB 123 CD (renvoit numéro n2)
    6. listing des amendes de la voiture AB 123 CD (nombre : 2)
    7. paiement de l'amende n1 par Capitaine Haddock (non pris en compte)
    8. listing des amendes de la voiture AB 123 CD (nombre : 2)
    9. paiement de l'amende n1 par Bob Sponge
    10. listing des amendes de la voiture AB 123 CD (nombre : 1)
2. L'objectif est d'arriver à l'affichage suivant pour le service :
```
Voiture enregistrée
Clio 3 AB 123 CD
possédée par Sponge Bob

Voiture enregistrée
Punto AB 124 CD
possédée par Haddock Capitaine

Amende enregistrée
Amende n°1 de 90 EUR pour AB 123 CD

Amende enregistrée
Amende n°2 de 140 EUR pour AB 123 CD

Amende payée
Amende n°1 de 90 EUR pour AB 123 CD
```
et à l'affichage suivant pour le client :
```
amende n°1 enregistrée
amende n°-1 enregistrée
amende n°2 enregistrée
```

```
Listing des amendes pour AB 123 CD
Amende n°1 de 90 EUR pour AB 123 CD
Amende n°2 de 140 EUR pour AB 123 CD

Listing des amendes pour AB 123 CD
Amende n°1 de 90 EUR pour AB 123 CD
Amende n°2 de 140 EUR pour AB 123 CD

Listing des amendes pour AB 123 CD
Amende n°2 de 140 EUR pour AB 123 CD
```

3. vous aurez surement noté que les classes de données contiennent des constructeurs sans arguments, des getters et des setters (sinon c'est que vous avez mal lu le texte qui précède). Commencez par supprimer les getters non nécessaires à la compilation. Qu'observez-vous ? Essayez aussi avec les setters et les constructeurs sans arguments.

4. SAAM - Client avec wsimport

Jusqu'ici nous avons supposé que le client avait accès à certaines des classes définies par le service (`Personne`, `Voiture`, `Amende`, etc.). En fait il est possible, à partir du moment où un descriptif WSDL (accompagné éventuellement de fichiers XSD) est disponible, de générer le code nécessaire au client. Pour cela, on utilise l'outil **wsimport**.

1. créez un nouveau projet pour un client SAAM.
2. lancez wsimport sur le descriptif de votre service (qui doit tourner, ou sinon passez le par file://)
   ```
   $ wsimport -keep -d . http://localhost:9999/ws/saam?wsdl
   ```
   L'option -keep permet de conserver les fichiers source (.java) générés (important pour que vous regardiez le contenu au point suivant) et l'option -d de préciser où générez les fichiers. Vous devriez déjà être habitués à ces options si vous avez une connaissance de base de Java RMI.
   Note : wsimport peut être appelé depuis l'IDE en général (action contextuelle sur fichier WSDL).
3. un répertoire (package) a été créé. Observez le contenu (et le contenu des fichiers). Il diffère du code Java que vous utilisiez jusqu'ici. En effet, celui-ci a été généré par JAX-WS RI (qui est utilisé par wsimport).
4. écrivez maintenant un client en utilisant le code généré et non le code du répertoire du service Web. En raison du code généré vous devrez surement procéder à quelques modifications (par exemple, au niveau des appels de constructeurs et de la gestion des tableaux). Que se passe-t-il avec l'affichage des amendes (méthode `toString`) ?

5. SAAM - Publisher avec wsgen

TODO

Tout comme nous ne sommes pas rentrés dans le cadre de ce premier cours dans les détails de JAX-WS (par exemple les nombreuses annotations possibles), nous ne rentrerons pas ici dans les

détails des options et annotations permettant de paramétrer le fonctionnement de JAX-WS RI ou JAXB. Des liens vous sont donnés en fin de document.

III. Développement top-down

On part ici d'une spécification du service Web sous la forme d'un document WSDL (ainsi qu'éventuellement de documents XML Schema décrivant les données manipulées). Nous utiliserons ici à nouveau **wsimport** qui sert à générer des classes Java à partir d'un document WSDL.

Ici les documents XSD et WSDL seront donnés. En général ce n'est pas forcément le cas (et pour cause). On peut utiliser n'importe quel éditeur pour les créer mais les frameworks dédiés services des IDE (voir en références) sont ici d'une grande aide.

1. Données échangés (incl. messages)

On commence par définir les types de données. Pour cela on utilise un (ou plusieurs) fichiers XSD externes. Ceci nous permet de réutiliser les descriptions indépendemment des services Web et de WSDL (par exemple avec JAXB pour générer des classes Java). Les types de messages quant à eux sont spécifique aux services et donc on choisit de les mettre dans le fichier WSDL (directement associés aux opérations en WSDL 2.0, associés aux opérations par l'entreprise de balises `message` en WSDL 1.1).

On va réaliser un service d'épicerie (simplissime). L'épicerie contient des produits (id, libellé). Le service doit permettre d'enregistrer des produits, d'augmenter leur quantité en stock, de connaître la quantité en stock d'un produit, de lister les produits disponibles et de lister le stock.

Les données sont décrites dans `epicerie.xsd`.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:tns="http://epicerie/"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           version="1.0"
           targetNamespace="http://epicerie/">
    <xs:complexType name="Produit">
```

```
        <xs:sequence>
            <xs:element name="id" type="xs:ID"></xs:element>
            <xs:element name="libelle" type="xs:string"></xs:element>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="StockProduit">
        <xs:sequence>
            <xs:element name="produit" type="xs:IDREF"></xs:element>
            <xs:element name="quantite" type="xs:int"></xs:element>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ListeProduit">
        <xs:sequence>
            <xs:element name="item" type="tns:Produit" minOccurs="0"
maxOccurs="unbounded" nillable="true"></xs:element>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ListeStock">
        <xs:sequence>
            <xs:element name="item" type="tns:StockProduit" minOccurs="0"
maxOccurs="unbounded" nillable="true"></xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Il faut noter ici que le but n'est pas de décrire **tous** les types de données manipulés par le service, mais **uniquement ceux qui seront visibles par les clients** (en entrée ou sortie d'une opération). Ainsi, on a décrit le type `StockProduit` car on envisage une opération qui retourne le stock total de l'épicerie (une liste de `StockProduit`). S'il avait été question uniquement de retourner la quantité d'un produit donné par exemple on aurait pu éviter cette définition. De même pour `Produit`.

2. Descriptif de service

On définit ici l'interface "fonctionnelle" du service (types de port, opérations, messages) et les informations de mise en oeuvre (types de bindings SOAP utilisés pour les opérations, déploiement, etc). Noter les types de données pour les messages (voir la remarque ci-dessus).

On obtient le fichier d'interface `epicerie.wsdl`.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:tns="http://epicerie/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             targetNamespace="http://epicerie/" name="EpicerieImplService">
    <types>
        <xsd:schema>
            <xsd:import namespace="http://epicerie/"
schemaLocation="epicerie.xsd"></xsd:import>
        </xsd:schema>
    </types>
    <message name="enregisterInputMessage">
        <part name="produit" type="tns:Produit"></part>
```

```
    </message>
    <message name="enregisterOutputMessage"></message>
    <message name="ajouterInputMessage">
        <part name="idProduit" type="xsd:IDREF"></part>
        <part name="qte" type="xsd:int"></part>
    </message>
    <message name="ajouterOutputMessage"></message>
    <message name="obtenirQuantiteInputMessage">
        <part name="idProduit" type="xsd:IDREF"></part>
    </message>
    <message name="obtenirQuantiteOutputMessage">
        <part name="return" type="xsd:int"></part>
    </message>
    <message name="listeProduitsIntputMessage">
    </message>
    <message name="listeProduitsOutputMessage">
        <part name="return" type="tns:ListeProduit"></part>
    </message>
    <message name="listStockInputMessage">
    </message>
    <message name="listeStockOutputMessage">
        <part name="return" type="tns:ListeStock"></part>
    </message>
    <portType name="Epicerie">
        <operation name="enregister">
            <input message="tns:enregisterInputMessage"></input>
            <output message="tns:enregisterOutputMessage"></output>
        </operation>
        <operation name="ajouter" parameterOrder="idProduit qte">
            <input message="tns:ajouterInputMessage"></input>
            <output message="tns:ajouterOutputMessage"></output>
        </operation>
        <operation name="obtenirQuantite">
            <input message="tns:obtenirQuantiteInputMessage"></input>
            <output message="tns:obtenirQuantiteOutputMessage"></output>
        </operation>
        <operation name="listeProduits">
            <input message="tns:listeProduitsIntputMessage"></input>
            <output message="tns:listeProduitsOutputMessage"></output>
        </operation>
        <operation name="listeStock">
            <input message="tns:listStockInputMessage"></input>
            <output message="tns:listeStockOutputMessage"></output>
        </operation>
    </portType>
    <binding name="EpicerieImplPortBinding" type="tns:Epicerie">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
style="rpc"></soap:binding>
        <operation name="enregister">
            <soap:operation soapAction=""></soap:operation>
            <input>
                <soap:body use="literal" namespace="http://epicerie/"></soap:body>
            </input>
            <output>
                <soap:body use="literal" namespace="http://epicerie/"></soap:body>
            </output>
        </operation>
```

```
        <operation name="ajouter">
            <soap:operation soapAction=""></soap:operation>
            <input>
                <soap:body use="literal" namespace="http://epicerie/"></soap:body>
            </input>
            <output>
                <soap:body use="literal" namespace="http://epicerie/"></soap:body>
            </output>
        </operation>
        <operation name="obtenirQuantite">
            <soap:operation soapAction=""></soap:operation>
            <input>
                <soap:body use="literal" namespace="http://epicerie/"></soap:body>
            </input>
            <output>
                <soap:body use="literal" namespace="http://epicerie/"></soap:body>
            </output>
        </operation>
        <operation name="listeProduits">
            <soap:operation soapAction=""></soap:operation>
            <input>
                <soap:body use="literal" namespace="http://epicerie/"></soap:body>
            </input>
            <output>
                <soap:body use="literal" namespace="http://epicerie/"></soap:body>
            </output>
        </operation>
        <operation name="listeStock">
            <soap:operation soapAction=""></soap:operation>
            <input>
                <soap:body use="literal" namespace="http://epicerie/"></soap:body>
            </input>
            <output>
                <soap:body use="literal" namespace="http://epicerie/"></soap:body>
            </output>
        </operation>
    </binding>
    <service name="EpicerieImplService">
        <port name="EpicerieImplPort" binding="tns:EpicerieImplPortBinding">
            <soap:address
location="http://localhost:9999/ws/carrouf"></soap:address>
        </port>
    </service>
</definitions>
```

Il faut ici savoir que WSDL supporte 4 type d'opérations :

- One-way: réception d'un message (par le service)
- Request-response: réception d'un message (par le service) et envoi d'un message corrélé (au client)
- Solicit-response: envoi d'un message (par le service) et réception d'un message corrélé (depuis le client)
- Notification: envoi d'un message (par le service).

Ici nous n'utilisons que des request-response (même si l'opération implantée n'a parfois pas d'argument ou de retour, nous en discuterons).

3. Création du service

1. générez le code Java à partir du fichier WSDL en utilisant wsimport`mbp-pascal:src`
```
pascalpoizat$ ls epicerie/
Epicerie.class          ListeStock.class        StockProduit.class
Epicerie.java           ListeStock.java         StockProduit.java
EpicerieImplService.class   ObjectFactory.class     epicerie.wsdl
EpicerieImplService.java    ObjectFactory.java      epicerie.xsd
ListeProduit.class      Produit.class           package-info.class
ListeProduit.java       Produit.java            package-info.java
```

2. complétez ce code en implantant l'interface `Epicerie` et en créant un lanceur
3. lancer le service et vérifiez qu'il est disponible

4. Création du client

Ici l'approche ne diffère pas de ce que nous avons vu précédemment en II.4.

1. faites un client pour tester votre service (avec ou sans wsimport).
   Le scenario est le suivant côté serveur : `creation produit 1 : Camembert`
```
creation produit 2 : Pain
ajout stock 1 : 20
ajout stock 2 : 10
```
   et côté client : `creation produit 1 : Camembert`
```
creation produit 2 : Pain
ajout stock 1 : 20
ajout stock 2 : 10
liste des produits
- Camembert (id=1)
- Pain (id=2)
liste des stocks
- Camembert : 20
- Pain : 10
```

IV. Services Statefull

TODO

V. Implantations de JAX-WS

JAX-WS a plusieurs implantations. Jusqu'ici nous avons utilisé celle disponible dans le JDK (1.6 ou plus). Mais cette implantation reste minimale par rapport à l'ensemble des standards WS-* disponibles (sécurité, transactions, etc.).

De nombreuses autres implantations sont disponibles, parmi lesquelles :

- Apache Axis2
- Apache CXF
- Metro

Vous trouverez aisément des comparatifs sur Internet (ce n'est pas l'objectif ici). Dans la suite nous utiliserons Apache AXIS2 (implante JAX-WS) et Apache CXF (implante JAX-WS et JAX-RS)

1. Installation de Tomcat

Tomcat est un serveur d'application. Il implante les technologies Java Servlet et JSP. Dans la suite nous donnons la procédure d'installation et de test, les numéros de version sont pour information (il y a parfois des incompatibilités entre versions des différents logiciels).

1. installer Tomcat (8.0.18)
   positionner la variable d'environnement `CATALINA_HOME`
   rajouter `$CATALINA_HOME/bin` dans le chemin d'accès
   lancer Tomcat avec `catalina.sh start`
   accéder à la page suivante : http://localhost:8080/
   si tout va bien vous voyez la page d'accueil de Tomcat
   stopper Tomcat avec `catalina.sh stop`.
2. paramétrage de l'IDE
   votre IDE doit connaître quel serveur d'application est installé afin de pouvoir par exemple déployer automatiquement les applications
   dans IntelliJ : préférences -> Application Servers -> rajouter Tomcat en précisant le chemin d'accès
   dans Eclipse : voir ce lien et **bon courage** car les plugins apparaissent/disparaissent selon les versions (Luna 1a 4.4.1 semble ok), et question stabilité ce n'est pas non plus la panacée.
3. création d'un projet avec l'IDE
   dans IntelliJ : choisir un projet Web Application (pas Web Service) et vérifiez que vous avez sélectionné le bon SDK et le bon Tomcat dans les paramètres
   dans Eclipse : choisir un projet Dynamic Web Project et vérifiez que vous avez sélectionné le bon SDK et le bon Tomcat dans les paramètres puis créez un nouveau fichier JSP nommé index.jsp dans les deux cas, complétez `web/index.jsp` ```html <%@ page contentType="text/html;charset=UTF-8" language="java" %> Hello JSP Hello !

- <%= i %> <% } %>

```

1. lancer l'exécution de votre projet (vérifiez que le bon serveur d'application est sélectionné)
   la page http://localhost:8080/ affiche les nombres de 0 à 9

Remarque : le but n'est pas ici de faire un cours sur Tomcat (voir sa documentation) ni sur JSP/Servlet (voir cours de développement Web suivi précédemment). Il s'agit juste de s'assurer que vous avez les bons outils pour travailler.

## 2. Installation d'Axis2

Axis2 peut être utilisé de deux façons. Soit comme un serveur stand-alone (indépendant), soit sous forme d'archive WAR comme extension à un serveur d'application (Tomcat par exemple). C'est le second choix que nous ferons. Voir la documentation d'Axis2 pour le premier choix.

1. installer Apache Axis2 (version WAR 1.6.2)
   stoppez Tomcat (si ce n'est pas déjà fait)
   copiez `axis2.war` dans `$CATALINA_HOME/webapps`
   relancer Tomcat
   accédez à la page suivante : http://localhost:8080/axis2/
   si tout va bien vous voyez la page d'accueil d'Axis
   faites une validation avec le lien disponible
   un service Web est disponible : http://localhost:8080/axis2/services/Version?wsdl
   stoppez Tomcat.
2. paramétrage de l'IDE
   votre IDE peut avoir besoin de savoir où Axis2 est installé
   **dans Eclipse :** c'est dans Préférences -> Axis2 Préférences, indiquez le répertoire `$CATALINA_HOME/webapps`

Dans la suite nous allons voir comment construire des services Web sur Axis2 avec l'aide d'un IDE. Il est aussi possible de le faire à la main en créant soi-même une archive contenant le code du service et un fichier de description.

**ATTENTION :** Eclipse Luna, à la date de rédaction de ces notes, supporte Axis 2 niveau 1.2 et 1.3, déployés sur un serveur qui implante la spécification Java Servlet 2.2. En cas de besoin, ne pas hésiter à regresser sur des versions inférieures (par exemple passer de Eclipse Luna à Eclipse Kepler).
```

VI. Utilisation d'un IDE

1. Eclipse

Si vous utilisez Eclipse vous pouvez créér des projets de type Web Service. L'IDE vous aide alors qu'il s'agisse d'approche bottom-up ou top-down. Commençons par le top-down.

1. créez un projet Java
2. ajoutez un fichier epicerie.xsd (XML Schema) et copiez-coller le contenu vu précédemment
3. ajoutez un fichier epicerie.wsdl (WSDL File) et au choix copiez-collez le contenu vu précédemment ou regardez comment l'IDE vous aide à remplir le fichier (vous pourrez aussi le faire en exercice d'auto-évaluation donc faites au plus simple)
4. créez un projet Dynamic Web Project `EpicerieWeb` comme précédemment mais vérifiez bien que la Dynamic Web Version est inférieure à 3.0 (c'est ok avec Axis ou CXF mais Axis2 n'aime pas apparemment **va savoir pourquoi**)
5. menu contextuel sur epicerie.wsdl -> Web Services -> Generate Java Bean Skeleton
6. vérifiez le paramétrage : Tomcat 8.0, Axis 2, EpicerieWeb, génération d'interface et de classes pour les types XSD
7. si tout va bien, votre service est déployé ici : http://localhost:8080/EpicerieWeb/services/EpicerieImplService?wsdl

Vous pouvez bien sur implanter rééllement votre service (en ajoutant du code directement dans le skeleton généré à partir du WSDL `EpicerieImplServiceSkeleton` ou par utilisation d'un objet métier externe), modifier ses paramètres (par exemple, adresse de déploiement, binding SOAP, etc). Le but n'est pas ici de faire une documentation complète d'Eclipse. Vous pourrez pratiquer sur les exercices de TD.

Question client, vous pouvez modifier votre client précédent (il y a plusieurs différences dues au framework utilisé, à l'adresse de déploiement par défaut, etc) ou bien Eclipse peut vous aider (rechercher en ligne si vous êtes intéressés).

Mais vous aurez surement noté jusqu'ici que le serveur d'application était en fait lancé à chaque fois que vous lanciez ou arrêtiez le projet. Qu'en est-il si l'on désire mettre le service Web en production ?

1. vérifiez que Tomcat est arrêté (dans Eclipse et au dehors)
2. exportez le projet Epicerie comme une archive WAR en indiquant le répertoire de déploiement `$CATALINA_HOME/webapps/`
3. lancer Tomcat
4. vérifiez que tout va bien : http://localhost:8080/EpicerieWeb/services/EpicerieImplService?wsdl

Passons au bottom-up. C'est encore plus simple.
Il suffit d'avoir une classe métier puis d'utiliser le menu contextuel pour en faire un service Web.

1. crééz un Dynamic Web Project
2. crééz une classe HelloWorld qui réalise l'opération vue en début du cours
3. utilisez l'IDE pour faire un service Web au dessus de cette classe
4. vérifiez que tout va bien en examinant le WSDL en ligne et en écrivant un client.

Vous aurez l'occasion d'utiliser Eclipse lors de TD, c'est pourquoi, à nouveau, nous ne rentrerons pas plus dans les détails techniques ici (la documentation est votre amie).

2. IntelliJ

Si vous utilisez IntelliJ ...

TODO

VII. Services REST

Nous allons maintenant voir un autre type de services Web, les services REST(ful). En effet, jusqu'ici nous avons vu en fait les services SOAP (WSDL). Les deux types de services ont leurs avantages et leurs inconvénients.

SOAP (Simple Object Access Protocol) :

- du RPC comme déjà vu en cours
- services stateless ou statefull
- interfaces expressives (WSDL)
- support de standards de sécurité, transactions, etc. (SOAP 1.2)

REST (Representational State Transfer) :

- différent du RPC sans sa philosophie

- mieux supporté en cas de ressources limitées
- basé sur le requêtage HTTP standard, GET / PUT / POST / DELETE
- services stateless
- interfaces simples (Create / Retrieve / Update / Delete)
- sécurité via HTTPS

1. Philosophie d'une API REST

On commence en toute logique par définir une API, sauf qu'ici nous n'avons pas d'interface Java (bottom-up) ou de description WSDL (top-down) mais on se base à la place sur la notion de ressource et de correspondance entre opérations sur cette ressource et les opérations CRUD.

- CREATE - création d'une ressource : POST
- RETRIEVE - obtention d'une ressource : GET (idempotence)
- UPDATE - mise à jour d'une ressource : PUT
- DELETE - suppression d'une ressource : DELETE

Il est facile de ne pas respecter cette "sémantique" de base. Cela conduit à des API non REST dans l'esprit, contre-intuitives, et pouvant poser des problèmes techniques (robots créants des ressources par effet de bord). Ainsi, pour créer un produit, on n'utilisera pas

```
GET /addproduct?id="1"&label="Camembert" HTTP/1.1
```

mais plutot ceci

```
POST /products HTTP/1.1
Host: serveur
Content-Type: application/xml
<?xml version="1.0"?>
<product>
  <id>1</id>
  <label>Camembert</label>
</product>
```

Note : nous verrons que l'on n'est pas obligé d'utiliser XML (JSON est une alternative).

On peut ensuite accéder au produit par exemple avec

```
GET /products/1 HTTP/1.1
```

De même pour une mise à jour, on n'utilise pas

```
GET updateproduct?id=1&newlabel="Gruyere" HTTP/1.1
```

mais plutôt

```
PUT /products/1 HTTP/1.1
Host: serveur
Content-Type: application/xml
<?xml version="1.0"?>
<product>
  <id>1</id>
  <label>Gruyere</label>
</product>
```

2. CXF

Mauvaise nouvelle pour certains, pour utiliser CXF il faut utiliser **maven**. La première chose à faire est d'avoir un fichier de configuration de maven correct. Un exemple de tel fichier est donné ci-dessous (**usage limité aux étudiants de la MIAGE**).

Attention, si vous utilisez un IDE, assurez-vous que maven utilise bien ce fichier (`$HOME/.m2/settings.xml` par défaut, mais paramétrable dans les IDE).

```
<settings>
    <mirrors>
        <mirror>
            <id>nexus</id>
            <name>Repository Miage</name>
            <url>http://miage11.u-paris10.fr:8081/nexus/content/groups/public
            </url>
            <!-- <url>https://teamcity-systeme.lip6.fr/nexus/content/groups/public
-->


<mirrorOf>external:*,!svnkit,!sonatype-oss-snapshots,!fornax,!phloc.com,!eclipse-bu
ndles,!sonatype,!lip6,!luna,!sonatype-oss
            </mirrorOf>
        </mirror>
    </mirrors>
    <servers>
        <server>
            <id>deployment</id>
            <username>deployment</username>
            <password>muges269:lass</password>
        </server>
    </servers>
    <profiles>
        <profile>
```

```
            <id>miage11</id>
            <activation>
                <activeByDefault>true</activeByDefault>
            </activation>
            <repositories>
                <repository>
                    <id>nexus</id>
                    <name>Repository MIAGE11</name>

<url>https://miage11.u-paris10.fr/nexus/content/groups/public</url>
                    <layout>default</layout>
                    <releases>
                        <enabled>true</enabled>
                        <updatePolicy>always</updatePolicy>
                        <checksumPolicy>warn</checksumPolicy>
                    </releases>
                    <snapshots>
                        <enabled>true</enabled>
                        <updatePolicy>always</updatePolicy>
                        <checksumPolicy>fail</checksumPolicy>
                    </snapshots>
                </repository>
                <repository>
                    <id>lip6</id>
                    <name>Repository LIP6</name>

<url>https://teamcity-systeme.lip6.fr/nexus/content/groups/public/</url>
                    <layout>default</layout>
                    <releases>
                        <enabled>true</enabled>
                        <updatePolicy>always</updatePolicy>
                        <checksumPolicy>warn</checksumPolicy>
                    </releases>
                    <snapshots>
                        <enabled>true</enabled>
                        <updatePolicy>always</updatePolicy>
                        <checksumPolicy>fail</checksumPolicy>
                    </snapshots>
                </repository>
                <repository>
                    <id>sonatype-oss</id>
                    <name>Repository Sonatype</name>
                    <url>https://oss.sonatype.org/content/groups/public/</url>
                    <releases>
                        <enabled>true</enabled>
                        <updatePolicy>always</updatePolicy>
                        <checksumPolicy>warn</checksumPolicy>
                    </releases>
                    <snapshots>
                        <enabled>true</enabled>
                        <updatePolicy>always</updatePolicy>
                        <checksumPolicy>fail</checksumPolicy>
                    </snapshots>
                </repository>
            </repositories>
        </profile>
    </profiles>
```

```
</settings>
```

2. Simuler un client REST

1. Téléchargez et installez CXF
2. Dans `samples/jax_rs/basics`, lancer la construction du projet avec `mvn clean install`
3. Lancez le serveur avec `mvn -Pserver` et notez bien l'adresse du serveur (normalement `localhost:9000`
4. Simulez un client en lançant **dans une autre fenêtre* telnet sur le serveur avec `telnet localhost 9000` ou installez et utilisez le plugin REST client de Firefox.`GET /customerservice/customers/123`

```
 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Customer>
    <id>123</id>
    <name>John</name>
</Customer>
```

```
 PUT /customerservice/customers
Content-Type: application/xml
<Customer>
    <id>123</id>
    <name>Mary</name>
</Customer>
```

```
 GET /customerservice/customers/123
 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Customer>
    <id>123</id>
    <name>Mary</name>
</Customer>
```

```
 GET /customerservice/customers/124
 Status Code: 204 No Content
Content-Length: 0
```

```
 POST /customerservice/customers
Content-Type: application/xml
<Customer>
    <id>124</id>
    <name>Bob</name>
</Customer>
```

```
 GET /customerservice/customers/124


<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Customer>
    <id>124</id>
    <name>Bob</name>
</Customer>
```

5. Faites quelques autres expérimentations, par exemple pour récupérer le contenu de commandes ou pour supprimer des clients

2. Utiliser CXF pour faire un client REST

TODO

3. Utiliser CXF pour faire un service REST

TODO

VIII. Utilisation d'un IDE (2)

TODO

IX. Orchestration de services

TODO

références additionnelles

JAX-WS / SOAP + WSDL

- JAX-WS Hello World Example - RPC Style les étapes 1., 2. et 3. du I. sont reprises de ce document
- Globinch JAX-WS tutorial tutoriel sur le développement top-down et bottom-up de services JAX-WS
- Design and develop JAX-WS 2.0 web services tutoriel sur le développement bottom-up de services JAX-WS
- Java API for XML-Based Web Services (JAX-WS) 2.0 spécification JAX-WS 2.0

JAX-RS / REST

- RESTful Web Services: The Basics principes de développement REST, le début du cours sur REST en est repris
- JAX-RS: The Java API for RESTful Web Services spécifications JAX-RS

Choix (spécifications et implantations, bindings, etc.)

- Web services hints and tips: JAX-RPC versus JAX-WS
  Part 1, Part 2, Part3, Part4, Part5 JAX-RPC ou JAX-WS ?
- Globinch SOAP Binding : Difference Between Document and RPC Style Web Services Bindings SOAP RPC ou Document ? (discute aussi RPC/encoded vs RPC/literal)
- Wikipedia: Web Services Description Language WSDL 1.1 ou 2.0 ?
- REST vs SOAP SOAP ou REST ? (si vous aimez les Oran-Outangs et les Pieuvres)

Outils (aide au développement de services)

- wsgen et wsimport wsgen génère des artefacts logiciels à partir d'une implantation de service (.class)
  wsimport génère des artefacts logiciels à partir d'un descriptif de service (WSDL)
- Apache Tomcat serveur d'application, implante les technologies Java Servlet et JSP
- Apache Axis2 conteneur de services Web, implante JAX-WS
- Apache CXF, tutoriel JAX-WS avec CXF framework de services Web, implante JAX-WS et JAX-RS
- Eclipse Web Services Project développement de Web services avec Eclipse
  la documentation est disponible dans l'Eclipse Help
- NetBeans Web Services Learning Trail développement de Web services avec NetBeans

---

1. l'absence de 's' n'est pas une faute d'orthographe, il s'agit du nom des balises dans le WSDL. Cependant, avec ces informations, le fichier WSDL ne contiendrait qu'une partie de l'information nécessaire au déploiement et à la communication avec un service Web. Quid de la façon dont les messages sont échangées (via le protocole SOAP en mode RPC par exemple), du protocole de transport sous-jacent (HTTP, SMTP, etc), de l'adresse de déploiement (localhost:9999/ws/saam) et du nom du service (SystemeAmendesImplService) ? Si vous regardez bien, le fichier WSDL contient aussi ces informations. A noter WSDL est maintenant en version 2.0 avec une variation dans le contenu, entre autres la suppression des `message` et l'utilisation directe de types XSD (voir les liens pour plus d'information). □