Services Web SOAP : technologies de base

UV IDL

Correction

Objectifs

L'objectif de ce TP est de découvrir quelques technologies sous-jacentes aux services Web. Pour cela, vous utiliserez une plate-forme Java EE et, en particulier, le serveur d'applications Glassfish.

À la fin de l'activité, l'étudiant doit être capable de :

- écrire un code Java pour appeler un service Web SOAP existant ;
- développer en Java un service Web SOAP qui offre des méthodes ayant des types complexes dans la signature des méthodes ;
- expliquer de manière simple et sans ambiguïté le rôle du SEI dans le cadre des services Web SOAP et de le générer en utilisant les outils disponibles dans le monde Java ;
- expliquer de manière simple et sans ambiguïté le rôle des technologies WSDL, SOAP et JAXB dans le cadre des services Web SOAP.

1 Préliminaires

1.1 Les outils

Pour réaliser le TP, vous utiliserez une machine virtuelle VMWare pré-configurée avec les outils nécessaires au TP. Il s'agit de la machine virtuelle *Ubuntu14_BD_PROG*. Elle contient, entre autres, l'IDE Eclipse dans sa version Java EE et un serveur Glassfish sur lequel vous déploierez les services Web développés. Le document « Description de l'environnement de travail », disponible sur Moodle, introduit brièvement la virtualisation de manière générale puis présente les caractéristiques de la machine virtuelle que nous avons mis à votre disposition ainsi que son utilisation.

Pour les services Web, Glassfish utilise Metro, l'implémentation de référence libre des spécifications Java pour le développement de services Web. Vous trouverez plus d'informations sur Metro à l'adresse `metro.java.net`.

1.2 Brève introduction des technologies

Si vous avez déjà travaillé avec des services Web et que vous comprenez les technologies utilisées, passez à la partie suivante.

L'annexe A.1, présente un résumé des technologies impliquées dans la programmation des services Web ainsi que des spécifications s'y référant. Lisez-le attentivement ; il s'agit d'un extrait du livre :

- Goncalves, A., *Beginning Java EE 6 Platform with Glassfish 3. From Novice to Professional*, APress, 2009.

2 Utilisation d'un service Web SOAP existant

Dans cette première partie du TP nous allons développer un logiciel consommateur d'un service Web SOAP existant. Il s'agit d'un service Web disponible en ligne à l'adresse `www.webservicex.net/ws/WSDetails.aspx?WSID=64&CATID=12`.

Accédez à cette adresse via le navigateur Web pour avoir des informations sur le service Web à utiliser. Dans la page que vous avez accédé vous trouverez la description du service Web, l'URL où se trouve le WSDL correspondant au service et une autre où on peut tester l'utilisation de celui-ci.

Exercice 1 (*Génération du SEI (« Service Endpoint Interface »)*)

▷ **Question 1.1 :**
D'après les informations données sur la page de description du service Web, quel est l'objectif du service ?

Il s'agit d'un service qui permet, à partir d'une adresse IP, de savoir quel est le pays d'origine de l'adresse.

Accédez maintenant à la page contenant le fichier WSDL du service. Nous verrons par la suite les différents éléments du fichier mais pour l'instant regardez uniquement l'élément *wsdl :service* ; tout fichier WSDL doit avoir un élément de ce type et il donne le nom du service Web. Cet élément est constitué de plusieurs sous éléments *wsdl :port*. Nous nous intéressons à celui dont le nom est *GeoIPServiceSoap*.

Nous avons vu en cours que les appels à un service Web se font via un SEI (*« Service Endpoint Interface »*). Il s'agit d'un logiciel qui converti les appels au service en messages SOAP et vice-versa.

Pour générer le SEI pour le service Web que nous voulons appeler, nous allons utiliser l'outil *wsimport* (`https://metro.java.net/1.2/docs/wsimport.html`) qui est inclut dans la plate-forme Java SE.

▷ **Question 1.2 :**
Ouvrez un terminal dans la machine virtuelle et utilisez la commande *wsimport* pour générer le SEI du service Web. Parmi les classes qui sont générées, deux correspondent aux éléments WSDL dont on vient de parler. Quelles sont ces classes ? Elles serviront explicitement au client pour appeler le service. Les autres sont des classes « utilitaires ».

Il s'agit des classes qui portent le même nom que les éléments du fichier WSDL : `GeoIPService.java` et `GeoIPServiceSoap.java`.
wsimport http ://www.webservicex.net/geoipservice.asmx ?WSDL génère les .class uniquement. L'ensemble de ces classes qui constitue le SEI du service Web.

▷ **Question 1.3 :**
Utilisez *wsimport* pour générer non seulement les binaires mais aussi les sources.

wsimport -keep -s src http ://www.webservicex.net/geoipservice.asmx ?WSDL génère les fichiers .class et garde les fichiers .java dans le répertoire `src`.

Exercice 2 (*Appel au service Web SOAP*)

Une fois le SEI généré, le consommateur du service peut l'utiliser pour appeler le service.

▷ **Question 2.1 :**
Créez un projet Eclipse contenant une seule classe (par ex. `GeoIPFinder`) qui prend en paramètre une adresse IP. Cette classe ayant besoin du SEI pour utiliser le service Web, vous devez ajouter dans le classpath du projet le répertoire où se trouve le SEI.

L'ajout du répertoire avec les sources du SEI peut se faire en sélectionnant le projet puis, via le menu contextuel, *Properties > Build Path > Link Source*. Ou en les copiant « à la main ».

▷ **Question 2.2 :**
Utilisez les deux classes identifiées dans la question 1.2 pour appeler le service Web. Piste : utilisez le constructeur sans paramètre...

```
public class GeoIPFinder {
  public static void main(String[] args) {
    if (args.length < 1){
      System.out.println("java usage : GeoIPFinder ipAdress");
      System.exit(-1);
    }
    String ipAddress = args[0];
    GeoIPService geoIPService = new GeoIPService();
    GeoIPServiceSoap geoIPServiceSoap = geoIPService.getGeoIPServiceSoap();
    GeoIP geoIP = geoIPServiceSoap.getGeoIP(ipAddress);
    System.out.println(geoIP.getCountryName());
  }
}
```

▷ **Question 2.3 :**
Testez votre consommateur. Vous pouvez utiliser la commande *ping* sur un terminal pour avoir une adresse IP. Par exemple, `ping google.com`.

Dans cette partie vous avez créé un consommateur d'un service Web existant. Pour ce faire, vous avez utilisé l'outil *wsimport*, disponible avec Java SE, pour générer le SEI correspondant au

service à utiliser. Vous avez aussi « fait connaissance » des deux éléments du fichier WSDL qu'il faut connaître pour utiliser un service Web : l'élément *service* et *port*.

3 Programmation d'un premier service Web

Comme déjà mentionné dans la section 1, nous utiliserons Glassfish comme infrastructure d'exécution des services Web. Cela veut dire que lorsqu'on développe un service, il faut le déployer (le « déposer ») là où l'infrastructure peut le récupérer pour l'exécuter.

Dans la machine virtuelle que nous utilisons pour le TP, l'IDE Eclipse est configuré pour piloter directement un serveur Glassfish. Vous pouvez donc, entre autres, le démarrer, l'arrêter et déployer et enlever des applications sur le (du) serveur.

Pour démarrer le serveur Glassfish, assurez vous que la perspective Java EE d'Eclipse est active. Dans cette perspective vous devez avoir un onglet *Servers* visible dans la partie basse d'Eclipse dans lequel la ligne *Glassfish 4.0 at localhost [Stopped]* indique que le serveur est arrêté. Démarrez-le à l'aide du menu contextuel [1] ou de l'icône représentant une flèche blanche entourée d'un cercle vert. Vérifiez qu'il a été correctement démarré en accédant par le menu contextuel à *Glassfish > View Admin Console*. S'il l'a été, un navigateur s'ouvre avec l'adresse `localhost:4848` : il s'agit de la page d'administration du serveur Glassfish.

Exercice 3 (*Le classique* `HelloWorld`)

Dans cet exercice vous allez créer votre premier service Web en utilisant Eclipse et vous déploierez le service sur Glassfish. Pour créer le service, vous devez d'abord créer un projet qui contiendra le service Web. Votre projet sera de type *Dynamic Web Project* (*File -> New Project -> Web -> Dynamic Web Project*).

Ajoutez au projet une page JSP qui ne contient que la fameuse phrase **Hello World ! !** (menu contextuel de *Web Content New > JSP File*).

Exécutez votre application en sélectionnant le projet et via le menu contextuel *Run as > Run on Server*. Vous devriez voir votre application dans la console d'administration de Glassfish dans la rubrique *Applications*.

Ajoutez maintenant une classe `HelloWorld` contenant une méthode `sayHello()` qui retourne la chaîne de caractères *Hi guys ! !*. Pour indiquer que vous voulez que cette classe (et donc toutes ces méthodes publiques) soit un service Web SOAP, il suffit d'ajouter l'annotation `@WebService` devant la déclaration de la classe.

Il faut absolument que la classe soit dans un paquetage sinon le service ne veut pas se déployer sur Glassfish. Le nom du paquetage contenant la classe ne doit pas contenir des majuscules. L'outil *wsimport* les ignore lors de la génération du SEI. Dans la correction l'annotation `@WebMethod` est optionnelle.

```
@WebService
public class HelloWorld {
  @WebMethod//(operationName="hello")
  public String sayHello(){
```

1. Il s'agit du menu qui apparaît lorsqu'on clique sur le bouton droit de la souris.

```
    return "Hello guys!";
  }
}
```

Sauvegardez votre classe puis allez dans la console d'administration de Glassfish dans la rubrique *Applications*. Regardez maintenant votre application. Une nouvelle ligne apparaît avec un composant *HelloWorld*.

Si vous demandez l'exécution de l'application, c'est toujours le code de la page JSP qui s'exécute. Pour exécuter le code de votre service Web (la méthode `sayHello()`), il faut avoir un client qui l'appelle. Vous pouvez donc créer un client tel que vous l'avez fait dans la section 1 mais vous pouvez aussi utiliser le client générique disponible avec Glassfish. Vous pouvez accéder à ce client par la console d'administration de Glassfish, rubrique *Applications* puis *View Endpoint*. Lorsque vous testez, s'affichent aussi les messages SOAP qui sont envoyés et reçus du service Web.

Ce même client générique vous permet d'accéder au fichier WSDL correspondant à votre service (lien *View WSDL*)[2]. Si vous regardez les URLs du client générique et du fichier WSDL, il s'agit de la même adresse sauf à partir du caractère *?*.

▷ **Question 3.1 :**
En regardant le fichier WSDL, à partir de quelles informations du fichier a été générée cette adresse ?

C'est l'adresse donnée dans l'élément *port* du fichier. Elle a été construite à partir du nom de l'élément *service*.

Exercice 4 (*HelloWorld personalisé*)

Dans cet exercice vous devez ajouter au service Web une méthode `sayHelloTo(String name)` qui retourne la chaîne de caractères *Hello* suivie du nom passé en paramètre. Testez votre service en utilisant le client générique de Glassfish et écrivez ensuite votre propre client.

Un seul conseil, le nom du paquetage contenant la classe ne doit pas contenir des majuscules. L'outil *wsimport* les ignore lors de la génération du SEI.
Dans cette correction, l'implémentation `HelloWorldImpl` est séparée de la classe `HelloWorldv2` qui correspond au service Web. C'est intéressant d'en parler aux étudiants pour qu'ils appliquent la même démarche : on sépare l'implémentation du « contrat ». Ce sont des bonnes pratiques.

```
public class HelloWorldImpl {
  public String sayHello(){
    return "Hello guys!";
  }
  public String sayHelloTo(String name){
    return "Hello " + name;
  }
}

@WebService
public class HelloWorldv2 {
```

2. Ce fichier a été généré automatiquement par Glassfish à partir du code du service.

```
  HelloWorldImpl helloWorldImpl = new HelloWorldImpl();
  public String sayHello(){
    return helloWorldImpl.sayHello();
  }
  public String sayHelloTo(String name){
    return helloWorldImpl.sayHelloTo(name);
  }
}

import eu.telecom_bretagne.tp_soap.*;
import eu.telecom_bretagne.tp_soapv2.*;
public class HelloWorldClient {
  public static void main(String[] args) {
    HelloWorldService helloWorld = new HelloWorldService();
    HelloWorld helloWorldPort = helloWorld.getHelloWorldPort();
    System.out.println(helloWorldPort.sayHello());
    HelloWorldv2Service helloWorldv2 = new HelloWorldv2Service();
    HelloWorldv2 helloWorldv2Port = helloWorldv2.getHelloWorldv2Port();
    System.out.println(helloWorldv2Port.sayHelloTo("Lucas!"));
  }
}
```

Dans cette partie vous avez créé votre premier service Web de manière très simple : il suffit d'ajouter une annotation `@WebService` pour que toutes les méthodes publiques deviennent accessibles par SOAP. Vous avez aussi utilisé le client générique fourni par Glassfish et développé votre propre client. Pour cela vous avez généré le SEI de votre service en utilisant l'outil disponible dans Java SE et utilisé ce SEI pour appeler le service.

Le contrat du service, le fichier WSDL a été automatiquement généré par Glassfish lors du déploiement du service ; vous n'avez pas vraiment eu a vous en soucier. Dans la partie suivante, on va s'intéresser à WSDL pour comprendre ses éléments de base et être capables ensuite de l'utiliser à bon escient.

4 Le contrat de base des services Web : WSDL

Dans votre formation et/ou dans votre vie professionnelle vous avez certainement déjà manipulé ou entendu parler de la notion d'interface. En Java, une interface est déclarée par **`public interface`**. Lorsqu'une classe réalise une interface, on dit que la classe réalise le « contrat » défini dans l'interface parce qu'elle doit donner du code à toutes les méthodes de celle-ci. Mais l'utilisation des interfaces n'est pas obligatoire.

Dans le cadre de la SOA, la notion de « contrat » devient obligatoire : tout service doit déclarer son contrat. Il existe deux niveaux de contrat dans la SOA : le contrat de base et le contrat étendu qui définit des garanties de qualité de service.

Concernant le contrat de base, il est décomposé en deux parties :

— une partie indépendante de la plate-forme, qui définit les interfaces du service, leurs opérations et les messages qui circuleront sur le réseau et
— une partie dépendante de la plate-forme, qui établit des liaisons entre la partie indépendante et des protocoles et des serveurs sur lesquels le service est déployé.

Partie indépendante de la plate-forme. L'appel à un service par un consommateur se traduit par l'envoi d'une *requête* vers celui-ci. En général, le service renvoie le résultat de l'exécution dans une *réponse*. Chaque couple (requête, réponse) est appelé *opération* : une opération est exécutée par le service sur réception de la requête associée et peut-être chargée de renvoyer une réponse avec le résultat de l'exécution. Les requêtes et réponses doivent transporter les informations nécessaires à l'exécution de l'opération.

L'ensemble d'opérations d'un même service correspond à l'*interface*. Celle-ci ne détaille pas comment les opérations sont implémentées.

Le contrat doit donc spécifier la liste des opérations offertes par le service et leur regroupement en interfaces et, pour chaque opération, la requête et la réponse à envoyer. Il doit également préciser la *structure* et le *format* de chaque information véhiculée par les requêtes/réponses.

Partie dépendante de la plate-forme. L'appel d'une opération d'un service par un consommateur se traduit par l'envoi d'un *message* transportant sa requête. Le message est transporté sur le réseau via un *protocole de communication* formalisant et organisant les échanges sur le réseau. En retour, et lorsque cela sera nécessaire, le client recevra un autre message, qui sera également transporté via le même protocole ou un autre protocole de communication.

Enfin, le service doit être localisé sur une *ressource* physique donnée, en général, un serveur.

Dans les services Web SOAP. Dans le cadre des services Web, le contrat de base est défini dans un langage indépendant du langage de programmation des services et des consommateurs : XML. En particulier, langage utilisé est WSDL (*« Web Services Description Language »*). Il s'agit d'une spécification du W3C (`http://www.w3.org/TR/wsdl`).

Les éléments principaux dans une description WSDL qui sont **indépendants de la plate-forme** sont les suivants :

- *<type>* : définition des types pour les données véhiculées dans les messages. Il s'agit donc de la structure et format des données ;
- *<message>* : définition du nom du message et type de ses paramètres (ou du retour). Il s'agit des requêtes et réponses mentionnées précédemment ;
- *<operation>* : définition du nom et type d'une opération qui utilise les messages. Il existe quatre modèles d'opérations : *« one way »*, *« request-response »*, *« sollicite-response »* et *« notification »* ;
- *<portType>* : regroupe un ensemble d'opérations ; il s'agit de la notion d'*interface* mentionnée précédemment.

En ce qui concerne les **éléments dépendants** de la plateforme :

- *<binding>* : spécifie une liaison entre un *<portType>* et un protocole de transport ainsi que la manière d'encapsuler les opérations du *<portType>* lors de la construction d'un message SOAP ;
- *<port>* : spécifie un point d'entrée, une liaison entre un *<binding>* et une adresse réseau ;
- *<service>* : spécifie une collection de points d'entrée, d'éléments *<port>*. Dans le jargon des services Web, cette collection est aussi appelée SEI (*« Service Endpoint Interface »*) [3].

Vous allez voir ces éléments dans la suite de cette partie. Tout d'abord simplifiez le WSDL généré par Glassfish pour le service de `HelloWorld` en commentant la méthode `sayHello()` [4]. Redéployez

3. Et oui, le même terme que le code coté client pour appeler un service Web.

4. Une autre manière de dire qu'une méthode publique ne doit pas être accessible par service Web SOAP est d'ajouter à la méthode l'annotation `@WebMethod(exclude=true)`.

le service sur Glassfish et visualiser le fichier WSDL généré pour votre service.

Exercice 5 (*Principaux éléments du WSDL*)

Vérifiez que tous les éléments présentés précédemment sont bien déclarés dans le fichier WSDL.

▷ **Question 5.1 :**
Regardez l'élement *portType*. Combien d'opérations il inclut ? Quel est leur nom et à quoi correspond-t'il dans le code du service Web ?

Il contient un seul élément de type *operation*. Le nom correspond au nom de la méthode offerte par le service Web. Donc, lors de la génération du WSDL Glassfish déduit le nom des opérations à partir du nom des méthodes à publier.

Comme mentionné précédemment, une opération est caractérisée par le message de requête et de réponse qui doivent circuler dans le réseau lors de l'appel de l'opération par un client.

▷ **Question 5.2 :**
Quel est le nom de ces messages pour notre opération ? Utiliser le client générique de Glassfish pour vérifiez que le nom des messages est bien celui déclaré dans le WSDL.

Le nom est *sayHelloTo* et *sayHelloResponse*. Lors de l'utilisation du client générique on voit les messages SOAP construits. Dans ces messages on voit bien ces noms.

▷ **Question 5.3 :**
Quel est le protocole de transport utilisé pour véhiculer les requêtes et les réponses SOAP ?

Pour répondre il faut regarder l'élément *binding*. Cet élément dit, entre autres, que les messages SOAP sont transportés par HTTP.

▷ **Question 5.4 :**
Quelle est l'adresse réseau sur laquelle le service Web attend les messages SOAP ?

Pour répondre il faut regarder l'élément *service* dans lequel on a le sous-élément *port* qui déclare l'adresse.

▷ **Question 5.5 :**
Résumez en quelques phrases le rôle de chaque élément du fichier WSDL. Faites valider vos réponses par un enseignant.

En résumé, l'élément *service* déclare un *port* (une localisation) pour un *binding* (une manière de transporter des requêtes SOAP sur le réseau).
Un *binding* déclare l'ensemble de requêtes SOAP (de méthodes disponibles dans le service Web) qui sont concernées par le mode de transport. Cet ensemble est déclaré dans un *portType*.
Une requête SOAP (une méthode disponible dans le service Web) correspond à un élément *opération*. Enfin, cet élément définit les messages SOAP qui devront circuler par le réseau lors de l'appel à l'opération.

L'élément *type* n'a pas été abordé. Il suffit de dire que c'est l'élément dans lequel le type des paramètres et type de retour des méthodes du service sont définis.

Exercice 6 (*Personnalisation du WSDL*)

Cet exercice est optionnel. Passez plutôt à l'exercice suivant et revenez sur celui-ci lorsque vous l'aurez fini.

Dans une démarche « classique » de développement de services Web, le fichier WSDL est généré à partir de la classe contenant le code du service. Une fois la classe écrite, son déploiement résulte en la génération d'une première version du fichier WSDL ; c'est cela que vous avez fait jusqu'à présent.

Cette manière de procéder n'est pas « idéale » : si le nom de la classe qui réalise le service Web change, le fichier WSDL changerait aussi, le contrat vis-à-vis des clients changerait... Nous allons voir dans cet exercice comment rendre plus indépendant le WSDL généré de la classe du service.

▷ **Question 6.1 :**
Complétez l'annotation `@WebService` pour modifier le nom de l'élément *portType* (nouveau nom = « MyHelloWorld ») et vérifiez que la modification a bien lieu. Faites de même pour l'élément *service* (nouveau nom = « MyHelloWorldService »).

Il suffit d'ajouter les paramètres
`@WebService(name="MyHelloWorld",serviceName="MyHelloWorldService")`.
Attention, lorsqu'on change le nom du service, l'URL où est disponible celui-ci n'est plus valide. Il faut la modifier pour voir le nouveau fichier...

▷ **Question 6.2 :**
Un autre paramètre de l'annotation `@WebService` permet de modifier l'espace de nommage utilisé dans le fichier WSDL. Si vous ne savez pas ce que c'est, vous pouvez l'assimiler à la notion de paquetage en Java : les noms déclarés dans le fichier WSDL sont valides dans l'espace de nommage déclaré dans l'élément *definitions*.

Regardez dans le fichier WSDL quel est le nom de cet espace de nommage. À partir de quelle information Glassfish a pu le construire ? Modifiez le en modifiant l'annotation `@WebService` et vérifiez le que fichier a bien été modifié.

À partir du nom du paquetage où se trouve la classe qui réalise le service Web. Pour le modifier il suffit d'ajouter le paramètre `targetNameSpace="http://www.myhelloworld.org"`.

▷ **Question 6.3 :**
À l'image de ce que nous avons fait pour l'annotation `@WebService`, il existe deux autres annotations, `@WebMethod` et `@WebParam`, ayant des paramètres pour rendre le WSDL indépendant du nom des méthodes et des paramètres de celles-ci.

Utiliser ces annotations et leurs paramètres pour modifier le nom de l'opération associée à la méthode `sayHelloTo(String name)` de votre service et du paramètre de cette requête.

`@WebMethod(operationName="talkTo", action="talk_to` Le deuxième paramètre n'est pas

demandé dans la question. Il sert à fixer le nom du message *input* associé à l'opération. `@WebParam(name="who")` pour modifier le nom du paramètre associé à la méthode.

Exercice 7 (*Valider des cartes bancaires*)

Vous allez créer maintenant un nouveau service Web qui à partir des informations d'une carte bancaire dit si elle est valide ou pas. La signature de la méthode de validation sera la suivante : `public String validate(String number, String expiryDate)` :

- `String number` : numéro de la carte bancaire ;
- `String expiryDate` : date d'expiration.

▷ **Question 7.1 :**
Créez le service Web `CardValidator`. Vous supposerez qu'une carte est valide si son numéro est un nombre pair et que la date d'expiration (dd-mm-yyyy) n'est pas dépassée.

```
public class CardValidatorImpl {
  public String validate(String number, String expiryDate){
    // Convert string to number for the card number
    double num = Double.parseDouble(number);
    // Convert the expiryDate to a date
    SimpleDateFormat formatter = new SimpleDateFormat("dd-MM-yyyy");
    Date date;
    try {
      date = formatter.parse(expiryDate);
    // Date of today
    Date today = new Date();
    if (((num % 2) == 0) && (date.after(today)))
      return "Carte valide !";
    } catch (ParseException e) {
      e.printStackTrace();
    }
    return "Carte non valide!";
  }

@WebService(serviceName = "CardValidatorService", portName = "CardValidatorPort")
public class CardValidator {
  CardValidatorImpl cardValidatorImpl = new CardValidatorImpl();
  @WebMethod(operationName = "validateParams")
  @WebResult(name = "validation")
  public String validate(@WebParam(name = "cardNumber") String number,
      @WebParam(name = "expiryDate") String expiryDate) {
    return cardValidatorImpl.validate(number, expiryDate);
  }
```

Déployez le service sur Glassfish. Testez-le en utilisant le client générique de Glassfish.

▷ **Question 7.2 :**
Quelle est la forme de la requête SOAP ? Et de la réponse ?

Requête SOAP

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
            xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
    <S:Body>
      <ns2:validate xmlns:ns2="http://services.telecom_bretagne.eu/">
        <arg0>889</arg0>
        <arg1>10-10-2014</arg1>
      </ns2:validate>
     </S:Body>
  </S:Envelope>
```

Réponse SOAP

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
            xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header/>
  <S:Body>
    <ns2:validateResponse xmlns:ns2="http://services.telecom_bretagne.eu/">
      <return>Carte non valide!</return>
    </ns2:validateResponse>
  </S:Body>
</S:Envelope>
```

▷ **Question 7.3 :**
Créez maintenant un consommateur de ce service. Ce consommateur sera une simple classe Java exécutable comme vous avez fait dans les exercices précédents.

Attention, il faut penser à générer le SEI avec *wsimport*.

```
public class CardValidatorClient {
  public static void main(String args[]){
  CardValidatorService cardValidatorService = new CardValidatorService();
  CardValidator cardValidator = cardValidatorService.getCardValidatorPort();
  System.out.println(cardValidator.validateParams("88", "10-10-2014"));
```

▷ **Question 7.4 :**
Utilisez les annotations vues précédemment pour adapter le contenu du fichier WSDL (nom de l'opération et des paramètres et retour, espace de nommage, nom du service). Vérifiez que les modifications ont bien été prises en compte. Note : pour modifier le nom du retour de l'opération, utilisez l'annotation `@WebResult`.

Pour la vérification il faut utiliser le client générique de Glassfish. Voir correction exercice 7.1

▷ **Question 7.5 :**
Adaptez votre client Java pour qu'il utilise le service avec le nouveau contrat.

Il faut utiliser à nouveau l'outil *wsimport* pour générer le SEI et copier les classes générées. Ensuite, modifier le code du client en accord avec les modifications sur WSDL. La correction de l'exercice 7.3 inclut déjà les modifications.

Pour résumer, les annotations que vous avez utilisé pour rendre le WSDL indépendant de la classe du service sont les suivantes :

- `@WebService` indique qu'il s'agit d'un service Web : toutes les méthodes publiques seront accessibles via le service Web. Elle permet de fixer le nom du service, de son port et de l'espace de nommage ;
- `@WebMethod` indique que la méthode doit être accessible à partir du service Web ; elle est optionnelle et permet de changer le nom de la méthode et du message associé à la requête SOAP correspondant à la méthode ;
- `@WebParam` indique qu'il s'agit d'un paramètre d'une méthode accessible à partir du service Web ; elle est optionnelle et permet entre autres de modifier le nom des paramètres de la méthode.
- `@WebResult` indique qu'il s'agit du retour d'une méthode accessible à partir du service Web ; elle est optionnelle et permet entre autres de modifier le nom du retour de la méthode.

5 Le SEI : convertir des objets en documents XML et vice-versa

Comme déjà mentionné précédemment, tout consommateur d'un service Web a besoin d'un SEI pour appeler une méthode du service. Ce SEI converti les appels de méthode en appels vers le service Web correspondant. Il a été automatiquement généré par l'outil *wsimport* à partir du fichier WSDL du service.

Dans cette partie vous allez voir comment le SEI fonctionne. Pour cela vous utiliserez la technologie Java qui l'implante, JAXB (*« Java Annotation for XML Binding »*) afin de mieux comprendre le fonctionnement des services Web SOAP.

Lors de l'appel à un service Web, le SEI coté client converti l'appel en message SOAP. De même, la résultat de l'exécution de la méthode est traduit par le SEI coté serveur en un message SOAP. SOAP (*« Simple Object Access Protocol »*) est une est spécification du W3C (`www.w3.org/TR/soap/`) dont l'objectif est de définir un protocole d'échange de messages permettant d'invoquer des services à distance de manière indépendante de leur implantation. Il permet donc de faire du RPC (*« Remote Procedure Call »*) indépendamment de la plate-forme d'exécution des applications (indépendamment du langage, de l'OS ...). Pour s'assurer de cette indépendance, SOAP s'appuie sur un protocole de transport (le protocole HTTP, mais aussi SMTP ou POP) et sur XML comme langage de structuration des données envoyées sous forme de messages.

Les spécifications SOAP incluent :

- *« SOAP enveloppe specification »* : quelle méthode ? paramètre ? retour ? erreur ?
- *« Data encoding rules »* : règles de représentation des types de données.

Un message SOAP est un document XML qui contient les éléments suivants :

- `<Envelope>` : définit le message et l'espace de nommage XML utilisés dans le document ;
- `<Header>` (optionnel) : contient des informations optionnelles pour le message. En particulier, les informations relatives au contrat étendu des services Web sont véhiculées dans cet élément ;

— `<Body>` : contient le message échangé entre les clients et le service ;
— `<Fault>` (optionnel) : fournit de l'information sur des erreurs survenus pendant le traitement du message.

Exercice 8 (*Arguments et valeur de retour* complexes)

▷ **Question 8.1 :**
Utilisez le client générique de Glassfish pour exécuter la méthode de validation d'une carte bancaire. Regardez les messages SOAP échangés entre le client et le service. À partir de quelles informations du fichier WSDL les éléments des messages SOAP ont été construits par le SEI ?

Ces messages ont déjà été vus dans l'exercice 7. Ce qu'il faut voir c'est que les messages SOAP ont la forme d'un document XML. Les différents éléments sont obtenus à partir des informations du fichier WSDL. Il suffit par exemple de regarder le nom des éléments correspondant aux paramètres de la méthode pour se rendre compte.

Ajoutez une nouvelle méthode à votre service de validation de cartes bancaires. La nouvelle méthode génère une carte bancaire valide à chaque appel. La signature de méthode sera
`public CreditCard generateValidCard()` où `CreditCard` est une nouvelle classe dont les attributs seront les mêmes que les paramètres de la méthode `validate` précédente.

▷ **Question 8.2 :**
Modifiez votre service de validation en ajoutant la nouvelle méthode. Pensez à créer la classe `CreditCard` dans un paquetage `eu.telecom_bretagne.model` pour que vos sources soient bien organisées.

Ici il faut faire attention lors de la génération du SEI. En effet, *wsimport* génère toutes les classes dans le meme paquetage. Donc il faudra faire attention de déplacer la classe `CreditCard` générée dans son paquetage.
Attention, les annotations de la classe `CreditCard` sont la correction d'un exercice plus loin dans le sujet.

++++++++++++ CardValidatorImpl.java ++++++++++++

```
public CreditCard generateValidCard(){
  // Date of today
  DateFormat dateFormat = new SimpleDateFormat("dd-MM-yyyy");
  Date today = new Date();
  String todayString = dateFormat.format(today);
  return new CreditCard(80+2,todayString);
}
```

++++++++++++ CardValidator.java ++++++++++++

```
@WebResult(name = "yourCard")
public CreditCard generateValidCard() {
  return cardValidatorImpl.generateValidCard();
}
```

++++++++++++ CreditCard.java ++++++++++++

```
@XmlType(name = "creditCard", propOrder = {
```

```
      "number",
      "expiryDate"
  })
public class CreditCard {
  private double number;
  private String expiryDate;
  public CreditCard(){
  }
  public CreditCard(double number, String date){
    this.number = number;
    this.expiryDate = date;
  }
  @XmlElement(name="num")
  public double getNumber() {
    return number;
  }
  public void setNumber(double number) {
    this.number = number;
  }
  public String getExpiryDate() {
    return expiryDate;
  }
  public void setExpiryDate(String expiryDate) {
    this.expiryDate = expiryDate;
  }
}
```

Le résultat de l'exécution de la méthode est un objet de type `CreditCard`. À chaque appel à la méthode `generateValidCard`, un objet de ce type est échangé entre le consommateur et le service. Or, comme vous l'avez vu dans l'introduction de cette partie et comme il est présenté dans l'annexe A.1, les données échangées entre les consommateurs et les services sont des documents XML (messages SOAP). Il faut donc convertir un objet de type `CreditCard` en document XML. Cette converstion est faite par le SEI.

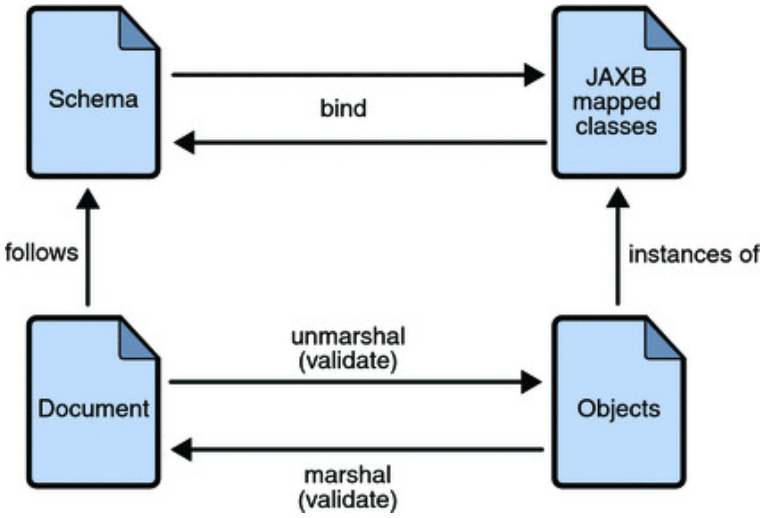
▷ **Question 8.3 :**
Testez le fonctionnement de la nouvelle méthode avec le client générique de Glassfish. Quelle est la forme de la réponse SOAP ?

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"
            xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <S:Body>
    <ns2:generateValidCardResponse
       xmlns:ns2="http://services.cardvalidator.telecom_bretagne.eu/">
      <yourCard>
        <expiryDate>29-09-2014</expiryDate>
        <number>82.0</number>
      </yourCard>
```

```
            </ns2:generateValidCardResponse>
          </S:Body>
        </S:Envelope>
```

Ce qu'on peut remarquer c'est que dans le message SOAP de réponse on a l'objet de type `CreditCard` converti en document XML.

Dans le monde Java, la technologie qui matérialise le SEI est JAXB. La figure ci-après résume le rôle de JAXB lors des appels aux services Web.



Concernant l'outil *wsimport*, son rôle est de générer les *JAXB mapped classes* à partir du fichier WSDL. En effet, si vous regardez le code de la classe `CreditCard` générée par *wsimport*, elle contient votre code mais aussi un ensemble d'annotations.

Vous trouverez dans `download.oracle.com/javaee/5/tutorial/doc/bnazy.html` et `download.oracle.com/javaee/6/api/javax/xml/bind/annotation/package-summary.html` des informations précises sur la signification des différentes annotations. Sachez uniquement que :

- `@XmlAccessorType` définit quels attributs de la classe seront sérialisés par JAXB dans chaque message SOAP. La valeur donnée signifie que tous les attributs privés seront sérialisés.
- `@XmlType` : donne le nom du type complexe qui correspond à cette classe et l'ordre dans lequel les éléments sérialisés vont l'être.
- Une autre annotation, `@XmlElement` permet de modifier le nom d'un attribut de la classe.

▷ **Question 8.4 :**
Modifier l'ordre dans lequel les attributs de la classe `CreditCard` seront envoyés dans la réponse SOAP.

Il faut faire la modification dans la classe coté serveur, puis déployer sur Glassfish. Ensuite, régénérer avec *wsimport*. Voir la correction dans la question 8.2

6 L'outil SoapUI

Dans cette partie nous allons voir les bases d'un outil largement utilisé dans le cadre des services Web SOAP dans les phases de développement. Il s'agit de SoapUI (`http://www.soapui.org/`),

un outil libre et gratuit. Pour l'utiliser vous pouvez soit l'installer sur votre machine (`http://sourceforge.net/projects/soapui/`), soit installer le plugin Eclipse correspondant (`http://www.soapui.org/IDE-Plugins/eclipse-plugin.html`). Ici nous supposerons que le plugin Eclipe a été installé.

L'objectif de l'outil est de vous permettre de *debugger* vos services Web. Pour cela, il vous permet, entre autres, de visualiser les messages SOAP envoyés.

Exercice 9 (*Validation d'une carte bancaire*)

Pour comprendre l'utilité d'un tel outil, nous allons modifier notre service de validation de cartes bancaires.

▷ **Question 9.1 :**
Ajoutez à votre service de validation de cartes bancaires une méthode `public String validateCard(CreditCard card)`. Elle envoie la chaîne de caractères *« Carte valide ! »* si la carte passée en paramètre est valide[5]. Sinon, elle envoie la chaîne *« Carte non valide ! »*.

++++++++ Classe CardValidatorImpl ++++++++++++++++++++++

```
  public String validate(CreditCard card){
    // Date of today
    Date today = new Date();
    DateFormat dateFormat = new SimpleDateFormat("dd-MM-yyyy");
    String expiryDateString = card.getExpiryDate();
    Date expiryDateDate;
    try {
      expiryDateDate = dateFormat.parse(expiryDateString);
      if (((card.getNumber() % 2) == 0) && (expiryDateDate.after(today)))
        return "Carte valide !";
    } catch (ParseException e) {
      e.printStackTrace();
    }
    return "Carte non valide!";
  }
}
```

+++++++++ Classe du service ++++++++++++++++++++++++++++++

```
  @WebMethod(operationName = "validateCard")
  @WebResult(name = "validation")
  public String validateCard(@WebParam(name = "card") CreditCard card) {
    return cardValidatorImpl.validate(card);
  }
```

Déployez le service sur Glassfish. Le paramètre de la méthode étant un type complexe, il n'est pas possible d'utiliser le client générique de Glassfish pour tester le fonctionnement du service. Par ailleurs, si on développe notre propre client, il ne sera pas possible de voir les messages SOAP envoyés entre le client et le service. Pour visualiser ces messages nous pouvons utiliser SoapUI.

5. Nous supposerons qu'une carte est valide si son numéro est pair et que la date d'expiration n'est pas dépassée.

Pour cela :

1. Ouvrez la perspective SoapUI dans Eclipse ;
2. Créez un projet de type SoapUI ;
3. Donnez l'URL de votre service de validation de cartes bancaires.

Avec ces informations SoapUI crée un *squelette de requête SOAP* par opération déclarée dans le fichier WSDL. Il suffit de donner les valeurs qu'on souhaite tester et demander l'envoi du message. Le message de retour sera affiché par SoapUI sur la droite de l'IDE.

Dans cette partie vous avez utilisé un outil de validation fonctionnelle dédié aux services Web. Il vous permet très simplement de construire des messages SOAP et de les soumettre aux services que vous voulez tester. Il s'agit de la fonctionnalité de base de SoapUI. N'hésitez pas à étudier les autres possibilités offertes par l'outil.

A Annexes

A.1 Introduction aux services Web SOAP

CHAPTER 14

SOAP Web Services

Formerly considered a buzzword, Service-Oriented Architecture (SOA) today forms part of day-to-day architectural life. However, sometimes it is confused with web services. SOA is an architecture principally based upon service-oriented applications that can be implemented with different technologies such as web services.

Web services are said to be "loosely coupled" because the client of a web service doesn't have to know its implementation details (such as the language used to develop it or the method signature). The consumer is able to invoke a web service using a self-explanatory interface describing the available business methods (parameters and return value). The underlying implementation can be done in any language (Visual Basic, C#, C, C++, Java, etc.). A consumer and a service will still be able to exchange data in a loosely coupled way: using XML documents. A consumer sends a request to a web service in the form of an XML document, and, optionally, receives a reply, also in XML.

Web services are also about distribution. Distributed software has been around for a long time, but unlike existing distributed systems, web services are adapted to the Web. The default network protocol is HTTP, a well-known and robust stateless protocol.

Web services are everywhere, and they can run on desktops or be used for business-to-business (B2B) integration so that operations that previously required manual intervention are performed automatically. Web services integrate applications run by various organizations through the Internet or within the same company (which is known as Enterprise Application Integration, or EAI). In all cases, web services provide a standard way to connect diverse pieces of software.

Understanding Web Services

Simply put, web services constitute a kind of business logic exposed via a service interface to a client application (i.e., a service consumer). However, unlike objects or EJBs, web services provide a loosely coupled interface using XML. Web service standards specify that the interface to which a message is sent should define the format of the message request and response, and mechanisms to publish and to discover web service interfaces (a registry).

In the Figure 14-1, you can see a high-level picture of a web service interaction. The web service can optionally register its interface into a registry (UDDI) so a consumer can discover it. Once the consumer knows the interface of the service and the message format, it can send a request and receive a response.

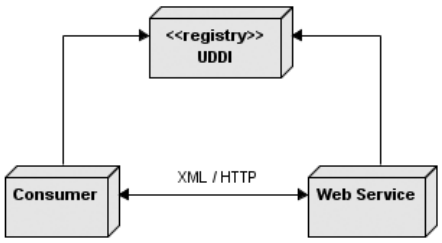


Figure 14-1. *The consumer discovers the service through a registry.*

Web services require several technologies and protocols to transport and to transform data from a consumer to a service in a standard way. The ones that you will come across more often are the following:

- Universal Description Discovery, and Integration (UDDI) is a registry and discovery mechanism, similar to the Yellow Pages, that is used for storing and categorizing web services interfaces.
- Web Services Description Language (WSDL) defines the web service interface, data and message types, interactions, and protocols.
- Simple Object Access Protocol (SOAP) is a message-encoding protocol based on XML technologies, defining an envelope for web services communication.
- Messages are exchanged using a transport protocol. Although Hypertext Transfer Protocol (HTTP) is the most widely adopted transport protocol, others such as SMTP or JMS can also be used.
- Extensible Markup Language (XML) is the basic foundation on which web services are built and defined (SOAP, WSDL, and UDDI).

With these standard technologies, web services provide almost unlimited potential. Clients can call a web service, which can be mapped to any program and accommodate any data type and structure to exchange messages through XML.

UDDI

Programs that interact with one another over the Web need to be able to find information that allows them to interconnect. UDDI provides this standard approach of locating information about a web service and how to invoke it.

UDDI is an XML-based registry of web services, similar to a Yellow Pages directory, where businesses can register their services. This registration includes the business type, geographical location, web site, phone number, and so on. Other businesses can then search the registry and discover information about specific web services. This information provides additional metadata about the service, describing its behavior and the actual location of the web service. It is in the form of a WSDL document. The client can then consume the WSDL document, which provides the information to bind and invoke the service.

WSDL

The UDDI registry points to a public WSDL file available on the Internet that can be downloaded by potential consumers. WSDL is the interface definition language (IDL) that defines the interactions between consumers and web services (see Figure 14-2). It is central to a web service as it describes the message type, port, communication protocol, supported operations, location, and what the client should expect in return. It defines the contract to which the web service guarantees it will conform. You can think of WSDL as a Java interface but written in XML.

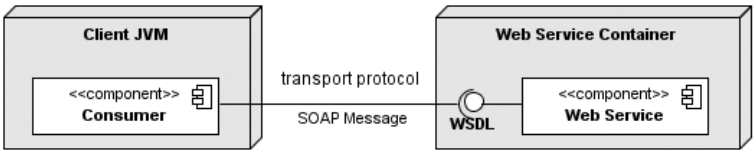


Figure 14-2. *WSDL interface between the consumer and the web service*

To ensure interoperability, a standard web service interface is needed for a consumer and a producer to share and understand a message. That's the role of WSDL. SOAP defines the way in which the message will be sent from one computer to another.

SOAP

SOAP is the standard web services application protocol. It provides the communication mechanism to connect web services exchanging formatted XML data across a network protocol, commonly HTTP. Like WSDL, SOAP heavily relies on XML because a SOAP message is an XML document containing several elements (an envelope, a header, a body, etc.).

SOAP is designed to provide an independent, abstract communication protocol capable of connecting distributed services. The connected services can be built using any combination of hardware and software that supports a given transport protocol.

Transport Protocol

For a consumer to communicate with a web service, a way to send messages to each other is needed. SOAP messages can be transported over a network using a protocol that both parties can support. Given that web services are used mostly on the Web, they usually use HTTP, but they can also use other network protocols such as HTTPS (Secure HTTP), TCP/IP, SMTP (Simple Mail Transport Protocol), FTP (File Transfer Protocol), and so on.

XML

XML is used in the Java EE platform for deployment descriptors, metadata information, and so on. For web services, XML is also used as an integration technology that solves the problem of data independence and interoperability. It is used not only as the message format, but also as the way the services are defined (WSDL) or exchanged (SOAP). Associated with these XML documents, schemas are used to validate exchanged data.

Web Services Specification Overview

Persistence is mostly covered by one specification: JPA. For web services, the situation is more complex, as you have to deal with many specifications from different standards bodies. Moreover, because web services are used by other programming languages, these specifications are not all directly related to the Java Community Process (JCP).

A Brief History of Web Services

Web services are a standard way for businesses to communicate over a network, and there were precursors before them: Common Object Request Broker Architecture (CORBA), initially used by Unix systems, and Distributed Component Object Model (DCOM), its Microsoft competitor. On a lower level, there is Remote Procedure Call (RPC) and closer to our Java world, Remote Method Invocation (RMI).

Before the Web, it was difficult to get all major software vendors to agree on a transport protocol. When the HTTP protocol became a mature standard, it gradually became a universal business medium of communication. At about the same time, XML officially became a standard when the World Wide Web Consortium (W3C) announced that XML 1.0 was suitable for deployment in applications. By 1998, both ingredients, HTTP and XML, were ready to work together.

SOAP 1.0, started in 1998 by Microsoft, was finally shipped at the end of 1999, and modeled typed references and arrays in XML Schema. By 2000, IBM started working on SOAP 1.1, and WSDL was submitted to the W3C in 2001. UDDI was written in 2000 by the Organization for the Advancement of Structured Information Standards (OASIS) to allow businesses to publish and discover web services. With SOAP, WSDL, and UDDI in place, the de facto standards to create web services had arrived with the support of major IT companies.

Java introduced web services capabilities with the Java API for XML-based RPC 1.0 (JAX-RPC 1.0) in June 2002 and added JAX-RPC 1.1 to J2EE 1.4 in 2003. This specification was very verbose and not easy to use. With the arrival of Java EE 5, the brand-new Java API for XML-based Web Services 2.0 (JAX-WS 2.0) specification was introduced as the preferred web service model. Today Java EE 6 is shipped with JAX-WS 2.2.

Java EE Specifications

To master all web services standards, you would have to spend some time reading a bunch of specifications coming from the W3C, the JCP, and OASIS.

The W3C is a consortium that develops and maintains web technologies such as HTML, XHTML, RDF, CSS, and so forth and, more interestingly, for web services, XML, XML Schemas, SOAP, and WSDL.

OASIS hosts several web service–related standards such as UDDI, WS-Addressing, WS-Security, WS-Reliability, and many others.

Coming back to Java, the JCP has a set of specifications that are part of Java EE 6 and Java SE 6. They include JAX-WS (JSR 224), Web Services 1.2 (JSR 109), JAXB 2.2 (JSR 222), Web Services Metadata 2.0 (JSR 181), and JAXR 1.0 (JSR 93). Taken together, these specifications are usually referred to by the informal term Java Web Services (JWS).

At first, these lists of specifications may make you think that writing a web service in Java would be difficult, especially when it comes to getting your head around the APIs. However,

the beauty of it is that you don't need to worry about the underlying technologies (XML, WSDL, SOAP, HTTP, etc.), as just a few JWS standards will do the work for you.

JAX-WS 2.2

JAX-WS (JSR 224) is the new name of JAX-RPC. JAX-RPC has been pruned in Java EE 6, meaning that it is proposed to be removed from Java EE 7.

JAX-WS 2.2 defines a set of APIs and annotations that allow you to build and consume web services with Java. It provides the consumer and service facilities to send and receive web service requests via SOAP, masking the complexity of the protocol. Therefore, neither the consumer nor the service has to generate or parse SOAP messages, as JAX-WS deals with the low-level processing. The JAX-WS specification depends on other specifications such as Java Architecture for XML Binding (JAXB).

Web Services 1.2

JSR 109 ("Implementing Enterprise Web Services") defines the programming model and runtime behavior of web services in the Java EE container. It also defines packaging to ensure portability of web services across application server implementations.

JAXB 2.2

Web services send requests and responses exchanging XML messages. In Java, there are several low-level APIs to process XML documents and XML Schemas. The JAXB specification provides a set of APIs and annotations for representing XML documents as Java artifacts, allowing developers to work with Java objects representing XML documents. JAXB (JSR 222) facilitates unmarshalling XML documents into objects and marshalling objects back into XML documents. Even if JAXB can be used for any XML purpose, it is tightly integrated with JAX-WS.

WS-Metadata 2.0

Web Services Metadata (WS-Metadata, specification JSR 181) provides annotations that facilitate the definition and deployment of web services. The primary goal of JSR 181 is to simplify the development of web services. It provides mapping facilities between WSDL and Java interfaces, and vice versa, through annotations. These annotations can be used within simple Java classes or EJBs.

JAXR 1.0

The Java API for XML Registries (JAXR) specification defines a standard set of APIs that allow Java clients to access UDDI. Like JAX-RPC, JAXR is the second web service–related specification to be pruned, with its removal proposed in the next version of Java EE. If the pruning is accepted, JAXR will still keep on evolving, but outside Java EE.

Reference Implementation

Metro is not a Java EE specification, but rather the open source reference implementation of the Java web service specifications. It consists of JAX-WS and JAXB, and also supports the