Introducción

En el laboratorio de profundización de objectDB se mostró cómo la base de datos se podía acceder de manera remota desde el ordenador que ejecutaba la lógica del negocio (que es quien accede a la base de datos). Lo único que falta ahora para construir una aplicación física en 3 niveles es que desde el ordenador cliente con la capa de presentación se pueda acceder a la capa de lógica del negocio que se ejecuta en una máquina remota. Esto se va a conseguir mediante una tecnología concreta: los Servicios Web.

En este laboratorio desarrollaremos una aplicación distribuida utilizando JAX-WS (Java API for XML Web Services), que es la API de Java para la creación de Servicios Web, y JAXB (*Java Architecture for XML Binding*), que es necesaria porque en los servicios web se serializan los objetos enviados usando XML. Para esta aplicación se necesitará utilizar una versión de Java a partir de JDK 6, ya que JAX-WS y JAXB se basan en el mecanismo de anotaciones de Java. Con la versión JDK 8 se incluyen las APIs de JAX-WS y JAXB, y no se requiere nada más. Pero para trabajar con versiones de JDK superiores a la 11, hay que definir las dependencias Maven que incluyen las librerías necesarias de JAX-WS y JAXB que han sido eliminadas del JDK.

Objetivos

Los objetivos del laboratorio son los siguientes:

- Crear un servicio web y el punto de acceso (endpoint) al mismo usando JAX-WS
- Crear una clase cliente que consuma dicho servicio web.

El servicio web a desarrollar y el modelo del dominio

El servicio web a desarrollar, en el dominio de las casas rurales y propietarios, hará públicos cuatro métodos que forman la lógica del negocio, y que serán los siguientes:

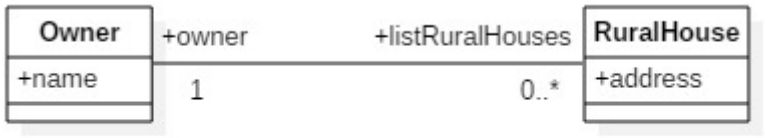
```
public List<Owner> getListOwners();
```
Método que devuelve la lista de todos los propietarios (todas las instancias de Owner)

```
public Owner getOwner(String name);
```
Método que devuelve la instancia de un propietario dado su nombre.

```
public List<RuralHouse> getListRuralHouses ();
```
Método que devuelve la lista de todas las casas rurales (instancias de RuralHouse)

```
public RuralHouse getRuralHouse(String address);
```
Método que devuelve la instancia de una casa rural dada su dirección.

El modelo del dominio utilizado es el siguiente:



donde cada propietario (Owner) tiene un nombre (name) y está asociado con la lista de casas rurales que posee, cada una de las cuales tiene su dirección (address).

Pasos a realizar

1. Crear el proyecto y las clases del dominio

Crear un proyecto Java (con el nombre **`SWruralHouses`**) y añadir las dos clases del dominio: Owner y RuralHouse. Nota: como son ya clases de implementación, en las mismas la asociación se ha implementado por medio de dos atributos: `listRuralHouses` en Owner y `owner` en RuralHouse, los cuales son redundantes entre sí pero permiten navegación en ambos sentidos.

Owner.java

```
package domain;

import java.io.Serializable;
import java.util.LinkedList;
import java.util.List;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlID;

@XmlAccessorType(XmlAccessType.FIELD)
public class Owner implements Serializable{
    @XmlID
    private String name;

private List<RuralHouse> listRuralHouses=new LinkedList<RuralHouse>();

public Owner() {  }
public Owner(String name) { this.name = name;   }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public List<RuralHouse> getListRuralHouses(){return listRuralHouses; }

public void setListRuralHouses(List<RuralHouse> listRH){
      listRuralHouses=listRH; }

public String toString(){return name+" "+listRuralHouses.toString(); }

public void addRuralHouse(RuralHouse rh){
       listRuralHouses.add (rh); } }
```

RuralHouse.java

```
package domain;

import java.io.Serializable;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlIDREF;
import javax.xml.bind.annotation.XmlID;

@XmlAccessorType(XmlAccessType.FIELD)

public class RuralHouse implements Serializable{
    @XmlID
    private String address;

    @XmlIDREF
    public Owner owner;

    public RuralHouse() { }
    public RuralHouse(String address,Owner o) {
       super();
       this.address = address;
       this.owner=o;
    }
    public String getAddress() {
       return address;
    }
    public void setAddress(String address) {
       this.address = address;
    }
    public String toString(){
       return address;
    }
    public Owner getOwner(){
       return owner;
    }
    public void setOwner(Owner o){
       owner=o;
    }
}
```

Si hay errores de compilación, ver la siguiente nota: "*incluir las librerías JAXB y JAX-WS*"

```
5  import javax.xml.bind.annotation.XmlAccessType;
6  import javax.xml.bind.annotation.XmlAccessorType;
7  import javax.xml.bind.annotation.XmlIDREF;
8  import javax.xml.bind.annotation.XmlID;
9
10 @XmlAccessorType(XmlAccessType.FIELD)
11
```

Explicación de las clases: Los objetos del dominio van a ser enviados a través de la red como parámetros en llamadas a métodos o como resultado de llamadas a métodos. Esto es, van a tener que ser serializados (transformados en texto con formato XML en este caso) antes de ser enviados por la red (mediante sockets). Los programadores no se van a tener que preocupar de implementar dichos mecanismos, pero sí deben tener en cuenta lo siguiente cuando implementen las clases del dominio:

1) Las clases deben implementar la interfaz `Serializable`

2) Las clases deben tener un constructor vacío (Ej: `public Owner(){..}`)

3) En las clases del dominio hay que poner la anotación Java `@XmlAccessorType(XmlAccessType.FIELD)` para indicar que se quiere serializar en XML todos los campos (no hace falta anotar los campos a serializar con `@XmlAttribute`)

4) Las clases deben tener métodos getter y setter para todos los campos (Ej: `public String getName(){..}` o `public void setName(String name){..}`)

5) Y además, cuando existan dependencias circulares entre las clases del dominio, esto es, cuando sea posible que un objeto contenga (o agregue) objetos que a su vez contengan otros objetos que pueden contener de manera recursiva a alguno de los anteriores, hay que evitar dichos ciclos.

   En nuestro caso esto ocurre ya que un objeto de la clase Owner contiene sus casas rurales (objetos de RuralHouse), que a su vez contienen objetos de la clase Owner.

   Para evitar los ciclos hay que anotar correctamente las clases Java. En el atributo que juega el rol de "clave" hay que poner la anotación `@XmlID`. Esto implica que el valor de dicho atributo identifica unívocamente a un objeto, y, por tanto, diferentes valores en dicho atributo corresponden a diferentes objetos.

```
@XmlID
private String name;
```

   Y en el atributo que juega el rol de "clave extranjera" hay que poner la anotación `@XmlIDREF`. El valor que tome para ese atributo identificará al objeto al que hace referencia, ya que tomará valores en el atributo "clave" de la clase a la que se refiere.

```
@XmlIDREF
public Owner owner;
```

IMPORTANTE: Lo que hace la anotación `@XmlIDREF` es indicar que se serialice la referencia del objeto pero no su contenido. En este caso la anotación `@XmlIDREF` de `owner` indica que se serialice el nombre del propietario (ya que `name` es el atributo etiquetado como `@XmlID` en `Owner`) y no todos los valores del objeto propietario. Así evitamos que se produzca un ciclo (ya que el contenido de propietario incluye la lista de casas rurales, que recursivamente volvería a incluir al mismo propietario).

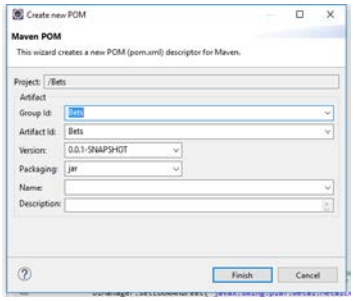
En este ejemplo se evita ese problema, se puede comprobar que al serializar el objeto de `Owner`, se serializarán con él sus casas rurales (puesto que no hemos puesto la anotación `@XmlIDREF` en el atributo `listRuralHouses` de Owner, pero sin embargo, cuando se serializan los objetos de RuralHouse, no se serializan sus Owner (debido a la anotación `@XmlIDREF` en el atributo `owner` de RuralHouse).

Más información en:
http://www.ibm.com/developerworks/rational/library/resolve-jaxb-cycle-errors/index.html

Nota: <u>incluir las librerías JAXB y JAX-WS</u>

A partir de la versión JDK11 de Java ya no se incluyen algunos paquetes que se consideran parte de la edición J2EE (Java 2 Enterprise Edition) de Java (ahora llamada Jakarta EE). Tan sólo se incluyen los paquetes que forman parte de la edición J2SE (Standard Edition). Lo que hay que hacer es incluir en el proyecto las clases apropiadas que implementan las APIs que forman parte del J2EE. La forma en la que se va a hacer es utilizando Apache Maven, que es una potente herramienta para la gestión y construcción de proyectos Java. En realidad sirve para compilar, ejecutar test automáticos (JUnit), generar los ficheros .jar, e instalar dichos .jar en repositorios y desplegarlos en servidores remotos. Para ello utiliza un fichero POM (Project Object Model) donde se describe el proyecto con sus dependencias con otros módulos y componentes y el orden de construcción de dichos elementos.

En nuestro caso, usaremos Eclipse para transformar nuestro proyecto Java a un proyecto Java Maven: Seleccionar el proyecto **`SWruralHouses`** en el Package Explorer => Click dcho. => Configure => Convert to Maven Project => Crear el fichero POM => Finish



En el fichero pom.xml generado se añaden las dependencias Maven necesarias para que en el proyecto se pueda trabajar con JAXB y JAX-WS. Para ello, añadir lo siguiente entre las etiquetas `</build>` y `</project>`. Nota: también se puede copiar el contenido del fichero pom.xml del proyecto Bets.zip en el fichero pom.xml

```
        <dependencies>
                <!-- API JAXB -->
                <dependency>
                        <groupId>jakarta.xml.bind</groupId>
                        <artifactId>jakarta.xml.bind-api</artifactId>
                        <version>2.3.2</version>
                </dependency>
                <!-- Runtime -->
                <dependency>
                        <groupId>org.glassfish.jaxb</groupId>
                        <artifactId>jaxb-runtime</artifactId>
                        <version>2.3.2</version>
                </dependency>
                <!-- API JAX-WS -->
                <dependency>
                        <groupId>jakarta.xml.ws</groupId>
                        <artifactId>jakarta.xml.ws-api</artifactId>
                        <version>2.3.2</version>
                </dependency>
                <!-- Runtime -->
                <dependency>
                        <groupId>com.sun.xml.ws</groupId>
                        <artifactId>jaxws-rt</artifactId>
                        <version>2.3.5</version>
                </dependency>
        </dependencies>
```

2. Definir la interfaz del Servicio Web

La siguiente interfaz Java define los métodos que forman el servicio web y que podrán ser invocados de manera remota. Como es una interfaz, todavía no se proporciona una implementación.

```
package service;

import java.util.List;

import javax.jws.WebMethod;
import javax.jws.WebService;

import domain.Owner;
import domain.RuralHouse;

@WebService
public interface WebServiceLogicInterface {
      @WebMethod
      public Owner getOwner(String name);
      @WebMethod
      public RuralHouse getRuralHouse(String address);
      @WebMethod
      public List<Owner> getListOwners();
      @WebMethod
      public List<RuralHouse> getListRuralHouses();
}
```

Obsérvese cómo se define que la interfaz Java es un servicio web mediante la anotación `@WebService` y cómo todos los métodos se deben anotar con `@WebMethod`. Cuando un método devuelve una lista, hay que utilizar el tipo de datos java.util.List.

3. Realizar la implementación del Servicio Web

La siguiente clase Java implementa la interfaz Java anterior, la cual describe el servicio web, esto es, implementa los métodos `getOwner`, `getListOwners`, `getRuralHouse`, y `getListRuralHouses`, que constituyen la lógica del negocio. Aunque para su implementación se debería acceder a una base de datos por medio de una clase de acceso a los datos, en este caso realizamos una implementación muy simple de la lógica del negocio que devuelve siempre los mismos datos de propietarios.

```
package service;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import javax.jws.WebMethod;
import javax.jws.WebService;
import domain.Owner;
import domain.RuralHouse;

@WebService(endpointInterface = "service.WebServiceLogicInterface")
public class WebServiceLogic {
      List<Owner> listOwners=new LinkedList<Owner>();
      List<RuralHouse> listRH=new LinkedList<RuralHouse>();
      public WebServiceLogic(){
            initialize(); }

      @WebMethod
      public Owner getOwner(String nameA){
              Iterator<Owner> it=listOwners.iterator();
              while (it.hasNext()) {
                    Owner o=it.next();
                    if (o.getName().equals(nameA))
                          return o; }
            return null; }

      @WebMethod
      public RuralHouse getRuralHouse(String address){
              Iterator<RuralHouse> it=listRH.iterator();
              while (it.hasNext()) {
                    RuralHouse rh=it.next();
                    if (rh.getAddress().equals(address))
                          return rh; }
            return null; }
      @WebMethod
      public List<Owner> getListOwners(){
            return listOwners;}

      @WebMethod
      public List<RuralHouse> getListRuralHouses(){
            return listRH;    }

      public void initialize(){
            listOwners=new LinkedList<Owner>();
            listRH=new LinkedList<RuralHouse>();
            Owner o1= new Owner("jon");
            Owner o2= new Owner("mikel");
            RuralHouse rh1 = new RuralHouse("jonTolosa",o1);
            RuralHouse rh2 = new RuralHouse("jonDonostia",o1);
            RuralHouse rh3 = new RuralHouse("mikelTolosa",o2);
            RuralHouse rh4 = new RuralHouse("mikelDonostia",o2);
            o1.addRuralHouse(rh1);
            o1.addRuralHouse(rh2);
            o2.addRuralHouse(rh3);
            o2.addRuralHouse(rh4);
            listOwners.add(o1); listOwners.add(o2);
            listRH.add(rh1);  listRH.add(rh2);
            listRH.add(rh3);  listRH.add(rh4);  } }
```

Obsérvese que en la implementación de esta clase:

1) Se implementan todos los métodos definidos en la interfaz `service.WebServiceLogicInterface`

2) `@WebService`(endpointInterface = "service.WebServiceLogicInterface") es una anotación que indica de manera explícita que se implementa un servicio web: el definido en service.WebServiceLogicInterface

3) Todos los métodos se anotan con `@WebMethod`

4. Crear una clase que publique el servicio web anterior

Por último, hay que dejar disponible el servicio web en ejecución para que los clientes puedan invocarlo. Para ello, definiremos una clase que al ejecutarse publicará el servicio web, esto es, creará una instancia de la clase que implementa el servicio web (la clase `WebServiceLogic`) y la dejará accesible en un puerto concreto (en este caso el `9999`) de la máquina local (se indica con el número de IP `0.0.0.0` que permite que sea accesible desde otras máquinas) y además dará un nombre lógico al servicio web (en este caso el "ws"). En realidad, lo que sucede es que se crea/lanza un servidor web que atenderá a las URLs correspondientes.

```
package service;

import javax.xml.ws.Endpoint;

public class Publisher {
  public static void main (String args[]){
    Endpoint.publish("http://0.0.0.0:9999/ws", new WebServiceLogic());
    System.out.println("Lanzado el servicio web");
  }
}
```

Nota: al hacer Run de la clase `Publisher` podría ser que diera un error de puerto ocupado en vuestra máquina (podríais cambiar el número de puerto `9999` por algún otro que no esté ocupado)

5. Comprobar que el servicio web está lanzado

En este momento, una vez lanzado el servicio web, se puede comprobar que está en marcha. El fichero que describe la interfaz del servicio web estará accesible en la URL siguiente: `http://localhost:9999/ws?wsdl`. Dicho fichero contiene la descripción WSDL (Web Services Definition Language) del servicio web, que no está expresada en Java, sino en XML. Lo cierto, es que se podrían crear clientes que invocaran a este servicio en otros lenguajes que no fueran Java: lo cual constituye una de las características más importantes de los servicios web. El contenido del fichero WSDL es similar al siguiente (en el que el servicio web tiene sólo los métodos getOwner y getListOwners):

```
- <definitions targetNamespace="http://service/" name="WebServiceLogicService">
  - <types>
    - <xsd:schema>
        <xsd:import namespace="http://service/" schemaLocation="http://localhost:9999/ws?xsd=1"/>
      </xsd:schema>
    </types>
  - <message name="getListOwners">
      <part name="parameters" element="tns:getListOwners"/>
    </message>
  - <message name="getListOwnersResponse">
      <part name="parameters" element="tns:getListOwnersResponse"/>
    </message>
  - <message name="getOwner">
      <part name="parameters" element="tns:getOwner"/>
    </message>
  - <message name="getOwnerResponse">
      <part name="parameters" element="tns:getOwnerResponse"/>
    </message>
  - <portType name="WebServiceLogicInterface">
    - <operation name="getListOwners">
        <input wsam:Action="http://service/WebServiceLogicInterface/getListOwnersRequest" message="tns:getListOwners"/>
        <output wsam:Action="http://service/WebServiceLogicInterface/getListOwnersResponse" message="tns:getListOwnersResponse"/>
      </operation>
    - <operation name="getOwner">
        <input wsam:Action="http://service/WebServiceLogicInterface/getOwnerRequest" message="tns:getOwner"/>
        <output wsam:Action="http://service/WebServiceLogicInterface/getOwnerResponse" message="tns:getOwnerResponse"/>
      </operation>
    </portType>
  - <binding name="WebServiceLogicPortBinding" type="tns:WebServiceLogicInterface">
      <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    - <operation name="getListOwners">
        <soap:operation soapAction=""/>
      - <input>
          <soap:body use="literal"/>
        </input>
      - <output>
          <soap:body use="literal"/>
        </output>
      </operation>
    - <operation name="getOwner">
        <soap:operation soapAction=""/>
      - <input>
          <soap:body use="literal"/>
        </input>
      - <output>
          <soap:body use="literal"/>
        </output>
      </operation>
    </binding>
  - <service name="WebServiceLogicService">
    - <port name="WebServiceLogicPort" binding="tns:WebServiceLogicPortBinding">
        <soap:address location="http://localhost:9999/ws"/>
      </port>
    </service>
</definitions>
```

Los tipos de datos utilizados en el servicio web se encuentran en la propiedad schemaLocation del documento WSDL anterior, esto es, en http://localhost:9999/ws?xsd=1

```
- <!--
     Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.1.6 in
  -->
- <xs:schema version="1.0" targetNamespace="http://service/">
    <xs:element name="getListOwners" type="tns:getListOwners"/>
    <xs:element name="getListOwnersResponse" type="tns:getListOwnersResponse"/>
    <xs:element name="getOwner" type="tns:getOwner"/>
    <xs:element name="getOwnerResponse" type="tns:getOwnerResponse"/>
  - <xs:complexType name="getListOwners">
      <xs:sequence/>
    </xs:complexType>
  - <xs:complexType name="getListOwnersResponse">
    - <xs:sequence>
        <xs:element name="return" type="tns:owner" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  - <xs:complexType name="owner">
    - <xs:sequence>
        <xs:element name="listRuralHouses" type="tns:ruralHouse" nillable="true" minOccurs="0"
        maxOccurs="unbounded"/>
        <xs:element name="nameA" type="xs:string" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  - <xs:complexType name="ruralHouse">
    - <xs:sequence>
        <xs:element name="adress" type="xs:string" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  - <xs:complexType name="getOwner">
    - <xs:sequence>
        <xs:element name="arg0" type="xs:string" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  - <xs:complexType name="getOwnerResponse">
    - <xs:sequence>
        <xs:element name="return" type="tns:owner" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:schema>
```

En este fichero se puede comprobar cómo se describen en XML las clases Owner y RuralHouse (ambas etiquetadas en XML como complexType, donde cada una de ellas tienen sus atributos. Así mismo, se puede comprobar la definición en XML de los dos métodos que forman el servicio web: getListOwners y getOwner, ambos también con sus parámetros y sus respuestas.

6. Creación de un cliente Java que consume el servicio web (cuando el código de la interfaz Java está disponible)

Cuando la interfaz Java que define el servicio web está disponible (el fichero `WebServiceLogicInterface`.class), entonces se puede definir una clase cliente que acceda al servicio web. En nuestro caso, este código tendría que ejecutarse en la capa de presentación, para que pueda acceder a la lógica del negocio.

```
package client;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Iterator;
import java.util.List;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

import service.WebServiceLogic;
import service.WebServiceLogicInterface;
import domain.Owner;
import domain.RuralHouse;

public class Main {
public static void main(String[] args) throws Exception{

  URL url = new URL("http://localhost:9999/ws?wsdl");
  QName qname = new QName("http://service/","WebServiceLogicService");
  Service service = Service.create(url, qname);
  WebServiceLogicInterface wsl=
                    service.getPort(WebServiceLogicInterface.class);

  System.out.println("BUSCANDO E IMPRIMIENDO OWNERS");
  List<Owner> listOwners=wsl.getListOwners();
  Iterator<Owner> it=listOwners.iterator();
  while (it.hasNext()) {
      Owner o=it.next();
      System.out.println("Owner: "+ o.getName()+
                         " y sus RH: "+o.getListRuralHouses());}

  System.out.println("BUSCANDO E IMPRIMIENDO RURAL HOUSES");
  List<RuralHouse> listRH=wsl.getListRuralHouses();
  Iterator<RuralHouse> itrh=listRH.iterator();
  while (itrh.hasNext()){
      RuralHouse rh=itrh.next();
      System.out.println("Su casa: "+rh.getAddress()+
                         " y su Owner: "+rh.getOwner());
  }
 }
}
```

Obsérvese que en la implementación de esta clase cliente:

1) Se crea un objeto de una clase URL que accede a la URL donde se encuentra publicada la interfaz WSDL del servicio web

```
URL url = new URL("http://localhost:9999/ws?wsdl");
```

2) Se crea un objeto de la clase QName con el nombre global que se le asigna al servicio web.

```
QName qname = new QName("http://service/",    "WebServiceLogicService");
```

En ese nombre se distinguen dos partes: el espacio de nombres o Namespace URI (que es `http://service/`) y el recurso local (que es `WebServiceLogicService`). ***Importante: no puede ser cualquier nombre***. Como espacio de nombres se pondrá después de `http:` el nombre del paquete de la clase que implementa el servicio web (en nuestro caso es "`service`" y como nombre de recurso local la concatenación entre el nombre de la clase (`WebServiceLogic`) y el string "`Service`".

```
package service;
...
public class WebServiceLogic
```

3) Se necesita crear previamente un objeto de la clase Service que permitirá obtener el objeto proxy o intermediario definido en el paso siguiente.

```
Service service = Service.create(url, qname);
```

4) Se crea un objeto proxy del servicio web. A este objeto ya se le podrá pedir directamente que ejecute los métodos del servicio web. Desde la aplicación cliente, ya se puede invocar al objeto remoto como si fuera un objeto local. Esto es, no debe encargarse de gestionar el paso de parámetros ni resultados a través de la red. Todo ello lo realizará el objeto proxy de tal manera que sea transparente para el cliente local.

```
WebServiceLogicInterface wsl=service.getPort(WebServiceLogicInterface.class);
```

La prueba de lo anterior es que a continuación se le pide al objeto proxy (objeto wsl) que ejecute el método `getListOwners` del servicio web.

```
List<Owner> listOwners=wsl.getListOwners();
```

En este punto podéis realizar lo siguiente:

1) Ejecutar el cliente (client.Main), que accede al servicio web lanzado tras ejecutar el servidor (service.Publisher). Este será el resultado de la ejecución:

```
BUSCANDO E IMPRIMIENDO OWNERS
Owner: jon y sus RH: [jonTolosa, jonDonostia]
Owner: mikel y sus RH: [mikelTolosa, mikelDonostia]
BUSCANDO E IMPRIMIENDO RURAL HOUSES
Su casa: jonTolosa y su Owner: null
Su casa: jonDonostia y su Owner: null
Su casa: mikelTolosa y su Owner: null
Su casa: mikelDonostia y su Owner: null
```

En esta ejecución se puede comprobar que las casas rurales correspondientes a los propietarios están disponibles (`Owner: jon y sus RH: [jonTolosa,`

`jonDonostia]`), pero no los propietarios de las casas (`Su casa: jonTolosa y su Owner: null`), tal y como se ha comentado al final del paso 1 de este laboratorio. En el último apartado de este laboratorio presentamos una solución que permite obtener los propietarios para las casas rurales.

2) Ejecutar el cliente, pero accediendo ahora al servicio web lanzado en otro ordenador.

Para ello hay que: a) terminar la ejecución del proceso service.Publisher en tu máquina y b) modificar la sentencia siguiente de tu cliente (client.Main) y que en vez de localhost aparezca la IP de la máquina de tu compañero/a

```
URL url = new URL("http://localhost:9999/ws?wsdl");
```

Obviamente, no podréis hacer esto todos a la vez. Hacedlo por parejas intercambiándoos el rol de cliente y servidor.

3) Identificar en el código de la aplicación RuralHouses si están bien definidos y en qué clases se encuentra todo lo explicado en este laboratorio, esto es, qué clases son las que definen el servicio web y qué clase es el cliente del servicio web.

7. Serialización de objetos anotados con @XmlIDREF

Al final del paso 1 hemos explicado que los objetos anotados con **@XmlIDREF** no se serializan en XML, sino que tan sólo se serializa la referencia del objeto. Por esa razón, en la ejecución del paso 6 hemos comprobado que al traer las casas rurales no venían con su correspondiente propietario.

```
BUSCANDO E IMPRIMIENDO RURAL HOUSES
Su casa: jonTolosa y su Owner: null
Su casa: jonDonostia y su Owner: null
Su casa: mikelTolosa y su Owner: null
Su casa: mikelDonostia y su Owner: null
```

Esto es debido a que en la clase `RuralHouse`, el atributo `owner` estaba anotado con `@XmlIDREF:`

```
public class RuralHouse implements Serializable{
    @XmlID
    private String address;

    @XmlIDREF
    public Owner owner;
```

Hay que recordar que es necesaria esa anotación para evitar el ciclo infinito que se produciría al serializar un objeto de Owner, el cual se serializa con sus objetos de RuralHouse, los cuales se serializarían con el objeto Owner anterior y así sucesivamente.

Si queremos que desde el cliente estén disponibles tanto el objeto de RuralHouse como el de su Owner, una posible solución consiste en definir una clase contenedora de RuralHouse, la cual permite almacenar la casa rural junto con su propietario.

```
package domain;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
@XmlAccessorType(XmlAccessType.FIELD)
public class RuralHouseContainer {
      private Owner o;
      private RuralHouse rh;
      public RuralHouseContainer(RuralHouse rh) {
          this.rh = rh;
          this.o=rh.getOwner(); }
      public RuralHouseContainer() {
          o = null;
          rh = null;  }
      public Owner getOwner(){
            return o; }
      public RuralHouse getRuralHouse(){
            return rh; }
      public String toString(){
            return o+"/"+rh; }
}
```

Cuando en la lógica del negocio se quiere devolver una casa rural `rh`, entonces se crea un nuevo objeto contenedor de dicha casa rural, `new RuralHouseContainer(rh)`. Dicho contenedor contiene tanto la casa rh como su propietario, y es lo que se devuelve al cliente. A dicho objeto se le puede pedir que nos dé tanto el propietario como la casa rural con los métodos `getOwner()`y `getRuralHouse()` Así se consigue tener a ambos en el cliente pese a tener definida la anotación `@XmlIDREF` en el atributo `owner` de la clase `RuralHouse`.

Para comprobarlo, hay que hacer lo siguiente:

1) Añadir el siguiente método en la interfaz `WebServiceLogicInterface`

```
      @WebMethod
      public List<RuralHouseContainer> getListRuralHouseContainers();
```

2) Implementar el método en la clase `WebServiceLogic`

```
      @WebMethod
      public List<RuralHouseContainer> getListRuralHouseContainers(){

            List<RuralHouseContainer> listRHC=
                        new LinkedList<RuralHouseContainer>();
              Iterator<RuralHouse> it=getListRuralHouses().iterator();
              while (it.hasNext()) {
                    RuralHouse rh=it.next();
                    listRHC.add(new RuralHouseContainer(rh));
              }
            return listRHC;
      }
```

3) Y añadir las siguientes instrucciones en el `main` de la clase cliente `Main`

```
System.out.println("BUSCANDO E IMPRIMIENDO RURAL HOUSE CONTAINERS");
 List<RuralHouseContainer> listRHC=wsl.getListRuralHouseContainers();
  Iterator<RuralHouseContainer> itc=listRHC.iterator();
  while (itc.hasNext()) {
        RuralHouseContainer rhc=itc.next();
        System.out.println("RH: "+ rhc.getRuralHouse().getAddress()+
                           " y su Owner: "+rhc.getOwner().getName());}
```

Al ejecutar el programa cliente Main se puede comprobar que ahora sí están disponibles los propietarios de las casas:

```
BUSCANDO E IMPRIMIENDO RURAL HOUSE CONTAINERS
RH: jonTolosa y su Owner: jon
RH: jonDonostia y su Owner: jon
RH: mikelTolosa y su Owner: mikel
RH: mikelDonostia y su Owner: mikel
```