Servicios Web

Sesión 2: Creación de servicios Web SOAP

Puntos a tratar

- Introducción
- Servicios web desde la vista del servidor
- Implementación del servicio JAX-WS
- Pasos para crear un WS con JAX-WS
- Implementación del servicio con jdk 1.6
- Implementación del servicio con Maven
- Modelo de despliegue de J2EE
- Implementación del servicio con Netbeans

Introducción

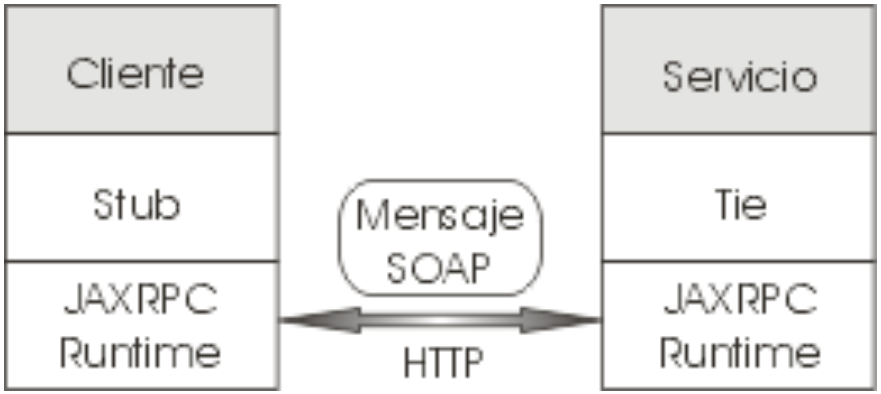
- Los Servicios Web que creemos deberán ofrecer una serie de operaciones que se invocarán mediante SOAP. Un servicio, por lo tanto:
  - Debe recibir y analizar el mensaje SOAP de petición
  - Ejecutará la operación y obtendrá un resultado
  - Deberá componer un mensaje SOAP de respuesta con este resultado y devolverlo al cliente del servicio
- Si tuviésemos que implementar todo esto nosotros
  - Desarrollar Servicios Web sería muy costoso
  - Se podría fácilmente cometer errores, no respetar al 100% los estándares y perder interoperabilidad

Librerías y herramientas

- Para facilitarnos la tarea contamos con:
  - Librerías (JAX-WS)
    - Nos permitirá leer y componer mensajes SOAP de forma sencilla
    - Estos mensajes respetarán el estándar
  - Herramientas
    - Generarán de forma automática el código para:
      1. Leer e interpretar el mensaje SOAP de entrada
      2. Invocar la operación correspondiente
      3. Componer la respuesta con el resultado obtenido
      4. Devolver la respuesta al cliente
- Sólo necesitamos implementar la lógica del servicio
  - La infraestructura necesaria para poderlo invocar mediante SOAP se creará automáticamente

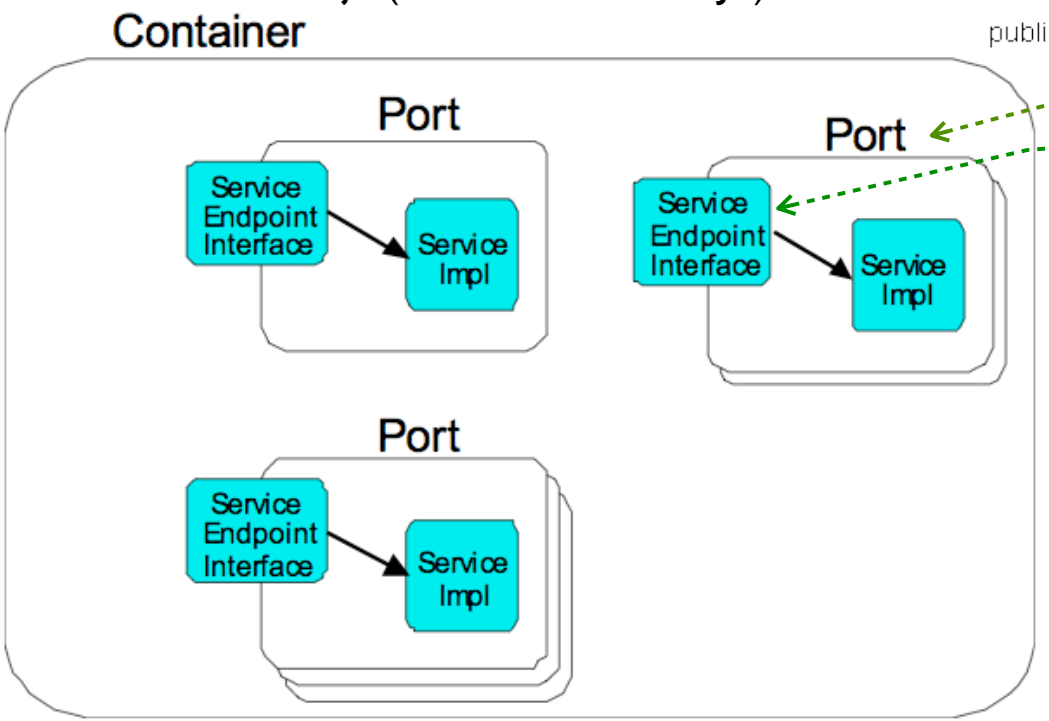
Capas del servicio

- Las capas *Stub* y *Tie*
  - Se encargan de componer e interpretar los mensajes SOAP que se intercambian
  - Utilizan la librería JAX-WS
  - Se generan automáticamente
- El cliente y el servicio
  - No necesitan utilizar JAX-WS, este trabajo lo hacen las capas anteriores
  - Los escribimos nosotros
  - Para ellos es transparente el método de invocación subyacente
  - El servicio es un componente que implementa la lógica (clase Java)
  - El cliente accede al servicio a través del *stub*, como si se tratase de un objeto Java local que tiene los métodos que ofrece el servicio

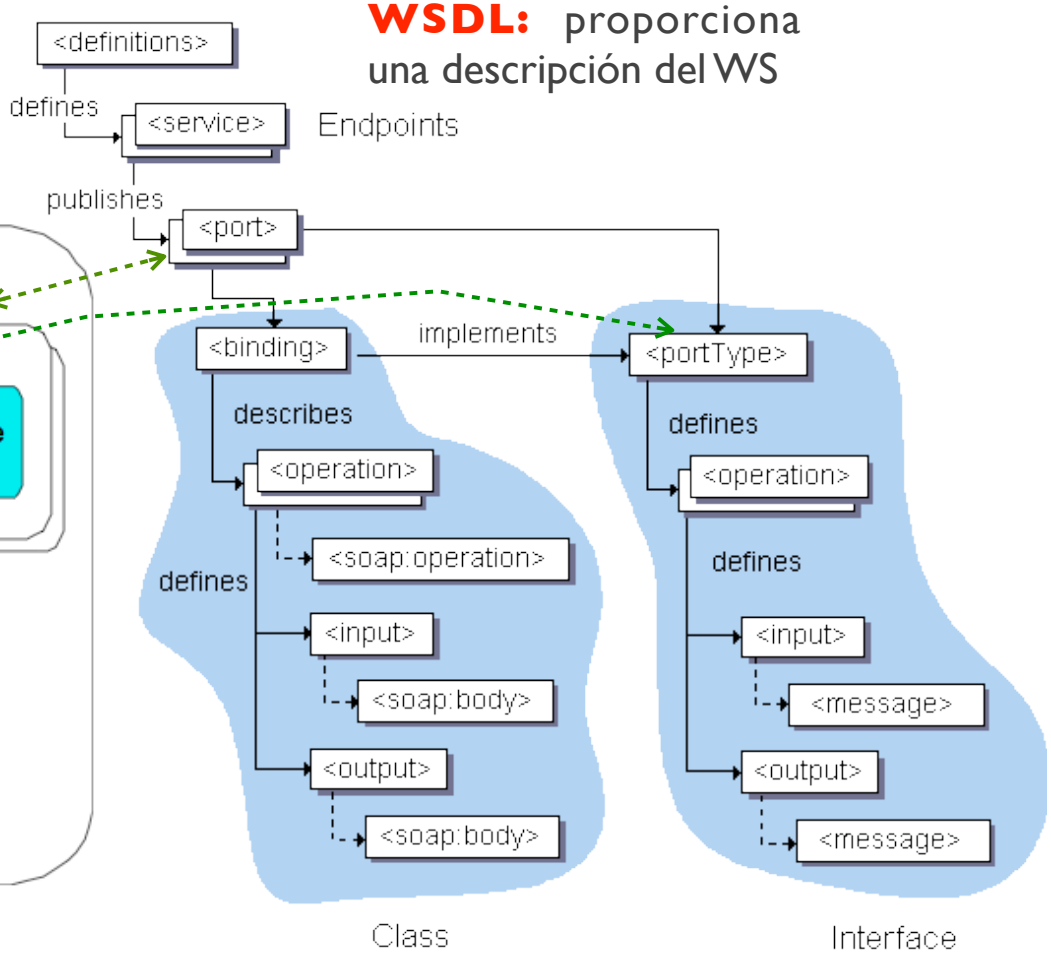


Vista del servidor de un servicio Web

Tipos de contenedores:
web (servlet JAX-WS)
ejb (stateless session EJB)



WSDL: proporciona una descripción del WS



SEI: Interfaz que define los métodos implementados por el *Service Implementation Bean*

Service Implementation Bean: Clase java que proporciona la lógica del WS

Cada **Port** tiene una dirección física particular asociada. Un Port asocia una dirección con la implementación del servicio

Implementación del servicio Web

- La implementación del servicio web (*Service Implementation Bean*) puede variar dependiendo del contenedor en el que despleguemos el componente Port, pero en general es una clase java anotada con javax.jws.WebService
  - Modelo de servlets (el componente Port reside en un contenedor Web)
  - Modelo EJB (el componente Port reside en un contenedor EJB)
- El componente Port asocia una dirección de puerto con la implementación del servicio

```
<service name="HelloService">
   <port name="HelloPort" binding="tns:HelloPortBinding">
    <soap:address location="http://localhost:8080/HolaServer/Hola"/>
   </port>
</service>
```

componente Port

Implementación del servicio

Dirección de puerto

El modelo de programación JAX-WS

- Podemos elegir dos puntos de partida:
  - Una clase Java que implementa el servicio Web
    - Tendremos la seguridad de que la clase que implementa el servicio tiene los tipos de datos adecuados
    - Tenemos menos control sobre el esquema XML generado
  - Un fichero WSDL
    - Tenemos un control total sobre el esquema que se está usando
    - Tenemos menos control sobre el *endpoint* del servicio generado y de las clases que utiliza

Pasos para crear un WS con JAX-WS

- El punto de partida para desarrolar un WS es una clase Java anotada con `javax.jws.WebService`.
- Dicha anotación define la clase como un *endpoint* del servicio web
- Un **SEI** (Service Endpoint Interface) es una interfaz Java que declara los métodos que el cliente puede invocar del servicio. No es necesario declarar dicha interfaz de forma explícita.
- Los pasos básicos para crear un WS JAX-WS (o *endpoint* JAX-WS) son:
  1. Codificar la clase que implementa el servicio
  2. Compilar la clase que implementa el servicio
  3. Empaquetar los ficheros en un *war*
  4. Desplegar el *war*. Los artefactos del WS necesarios para comunicarse con los clientes serán generados por Glassfish durante el despliegue

Anotaciones JAX-WS

- La clase que implementa el servicio debe hacer uso de las anotaciones de JAX-WS para servicios web
- *Anotaciones que pueden utilizarse:*

@WebService	Indica que la clase define un servicio web. Se pueden especificar como parámetros los nombres del servicio (serviceName), del componente Port (portName), del SEI del servicio (portName), de su espacio de nombres (targetNamespace), y de la ubicación del WSDL (wsdlLocation), que figurarán en el documento WSDL del servicio:
	
	`@WebService(name="ConversionPortType",`
	`serviceName="ConversionService",`
	`portName="ConversionPort",`
	`targetNamespace="http://jtech.ua.es",`
	`wsdlLocation="resources/wsdl/")`

@SOAPBinding	Permite especificar el estilo y la codificación de los mensajes SOAP utilizados para invocar el servicio. Por ejemplo:
	
	`@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,`
	`use=SOAPBinding.Use.LITERAL,`
	`parameterStyle=`
	`SOAPBinding.ParameterStyle.WRAPPED)`

Anotaciones JAX-WS

`@WebMethod`	Indica que un determinado método debe ser publicado como operación del servicio. Si no se indica para ningún método, se considerará que deben ser publicados todos los métodos públicos. Si no, sólo se publicarán los métodos indicados. Además, de forma opcional se puede indicar como parámetro el nombre con el que queramos que aparezca la operación en el documento WSDL:
	`@WebMethod(operationName="eurosAptas")`
	`public int euro2ptas(double euros) {`
	`  ...`
	`}`

`@Oneway`	Indica que la llamada a la operación no debe esperar ninguna respuesta. Esto sólo lo podremos hacer con métodos que devuelvan `void`. Por ejemplo:
	`@Oneway()`
	`@WebMethod()`
	`public void publicarMensaje(String mensaje)`
	`{`
	`  ...`
	`}`

Anotaciones JAX-WS

`@WebParam`	Permite indicar el nombre que recibirán los parámetros en el fichero WSDL:
	`@WebMethod(operationName="eurosAptas")`
	`public int euro2ptas(`
	`  @WebParam(name="CantidadEuros",`
	`targetNamespace="http://jtech.ua.es")`
	`  double euros) {`
	`  ...`
	`}`

`@WebResult`	Permite indicar el nombre que recibirá el mensaje de respuesta en el fichero WSDL:
	`@WebMethod(operationName="eurosAptas")`
	`@WebResult(name="ResultadoPtas",`
	`  targetNamespace="http://jtech.ua.es")`
	`public int euro2ptas(double euros) {`
	`  ...`
	`}`

Tipos de datos compatibles

- Los tipos de datos que podemos utilizar como tipo de los parámetros y valor de retorno de los métodos del servicio serán los tipos soportados por JAXB.
- Tipos de datos básicos y *wrappers* de estos tipos

`boolean`	`java.lang.Boolean`
`byte`	`java.lang.Byte`
`double`	`java.lang.Double`
`float`	`java.lang.Float`
`int`	`java.lang.Integer`
`long`	`java.lang.Long`
`short`	`java.lang.Short`
`char`	`java.lang.Character`

Otros tipos de datos y estructuras

- Otros tipos de datos

```
java.lang.String          java.util.Calendar
java.math.BigDecimal      java.util.Date
java.math.BigInteger      java.awt.Image
```

- Colecciones y genéricos

Listas: `List`	Mapas: `Map`	Conjuntos: `Set`
`ArrayList`	`HashMap`	`HashSet`
`LinkedList`	`Hashtable`	`TreeSet`
`Stack`	`Properties`	
`Vector`	`TreeMap`	

Requisitos para las clases que implementan el *endpoint*

- Podremos utilizar objetos de clases propias, siempre que estas clases cumplan
  - Deben tener un constructor `void` público
  - No deben implementar `javax.rmi.Remote`
  - Todos sus campos deben
    - Ser tipos de datos soportados por JAXB
    - Los campos públicos no deben ser ni `final` ni `transient`
    - Los campos no públicos deben tener sus correspondientes métodos `get`/`set`.
  - Si no cumplen esto deberemos construir serializadores
- También podemos utilizar *arrays* y colecciones de cualquiera de los tipos de datos anteriores

Ejemplo de *endpoint* JAX-WS

```
package es.ua.jtech.servcweb.conversion;

import javax.jws.WebService;

@WebService
public class ConversionSW {


  public ConversionSW() { }


  public int euro2ptas(double euro) {
    return (int) (euro * 166.386);
  }

  public double ptas2euro(int ptas) {
    return ((double) ptas) / 166.386;
  }
}
```

Ejemplos de anotaciones

```
package utils;
import javax.jws.*;

@WebService(name="MiServicioPortType", serviceName="MiServicio",
            targetNamespace="http://jtech.ua.es")
public class MiServicio {
  @Resource private WebServiceContext context;

  @WebMethod(operationName="eurosAptas")
  @WebResult(name="ResultadoPtas", targetNamespace="http://jtech.ua.es")

  public int euro2ptas(@WebParam(name="CantidadEuros",
                targetNamespace="http://jtech.ua.es")
    double euro) { ... }

  @Oneway()
  @WebMethod()
  public void publicarMensaje(String mensaje) { ... }
}
```

Modelo EJB

- Podemos utilizar un *Stateless Session Bean* para implementar el servicio. En este caso, el componente Port residirá en un contenedor EJB
- Se utilizan las mismas anotaciones y normas y requisitos que hemos mencionado
- Se utiliza la anotación *@Stateless*

```
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.ejb.Stateless;

@WebService
@Stateless()
public class Hello {
    public void Hello() {}
    @WebMethod
    public String sayHello(String name) {
        return “Hola, “+ message + name + ".";
    }
}
```

Empaquetado del servicio Web

- Dependiendo de si se utiliza el modelo de servlets o EJB, el servcio se puede empaquetar en un *war* o un *ejb-jar*
- El desarrollador es responsable de empaquetar, directamente o referenciando:
  - El fichero wsdl (opcional)
  - La clase SEI (opcional)
  - La clase que implementa el servicio y sus clases dependientes
  - Los artefactos portables generados por JAX-WS
  - Descriptor de despliegue (opcional si se usan anotaciones JAX-WS)

Despliegue del servicio Web (I)

- JAX-WS soporta dos modelos de despliegue:
  - El definido por JSR-109 (Web Services for Java EE), que utiliza el fichero *webservices.xml*
  - El modelo específico definido en JAX-WS, que utiliza los ficheros *web.xml* y *sun-jaxws.xml*
- Contenido del fichero *sun-jaxws.xml*

```
<?xml version="1.0" encoding="UTF-8"?>

<endpoints version="2.0"
           xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime">

    <endpoint implementation="ws.news.NewsService"
              name="NewsService"
              url-pattern="/NewsService"/>

</endpoints>
```

Cada endpoint representa un port WSDL

clase anotada con @WebService

Despliegue del servicio Web (II)

- Contenido del fichero *web.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" ... >
    <listener>
        <listener-class>
           com.sun.xml.ws.transport.http.servlet.WSServletContextListener
        </listener-class>
    </listener>
    <servlet>
        <servlet-name>NewsService</servlet-name>
        <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>NewsService</servlet-name>
        <url-pattern>/NewsService</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

listener que inicializa y configura el endpoint (componente port) del servicio web

servlet que sirve las peticiones realizadas al servicio

Responsabilidades del contenedor

- El contenedor en el que resida nuestro servicio web debe proporcionar el *runtime* de JAX-WS, para soportar peticiones de invocación sobre los componentes port desplegados en dicho contenedor. El *runtime* se encarga de convertir las llamadas SOAP entrantes en llamadas al API java y viceversa. El contenedor es responsable de:
  - "Escuchar" a la espera de peticiones SOA/HTTP
  - "Parsear" el mensaje de entrada según el tipo de *binding*
  - "Mapear" el mensaje a la clase y método correspondiente, según los descriptores de despliegue
  - Invocar al *Service Implementation Bean*
  - Capturar la respuesta de la invocación y mapearla al mensaje SOAP
  - Enviar el mensaje al cliente del servicio web

Generar el servicio con JDK 1.6

- Contamos con la herramienta `wsgen`
  - Genera los artefactos necesarios
  - Debemos compilar previamente el fichero del *endpoint*

```
wsgen -cp bin -s src -d bin
      es.ua.jtech.servcweb.conversion.ConversionSW
```

- También disponible como tarea de Ant

```
<wsgen classpath="${bin.home}"
       sei="${service.class.name}"
       sourcedestdir="${src.home}"
       destdir="${bin.home}" />
```

Publicar servicios con JDK 1.6

- Podemos publicar sin servidor de aplicaciones

```
public class Servicio {
  public static void main(String[] args) {
    Endpoint.publish(
        "http://localhost:8080/ServicioWeb/Conversion",
        new ConversionSW());
  }
}
```

Construcción, empaquetado y despliegue con Maven

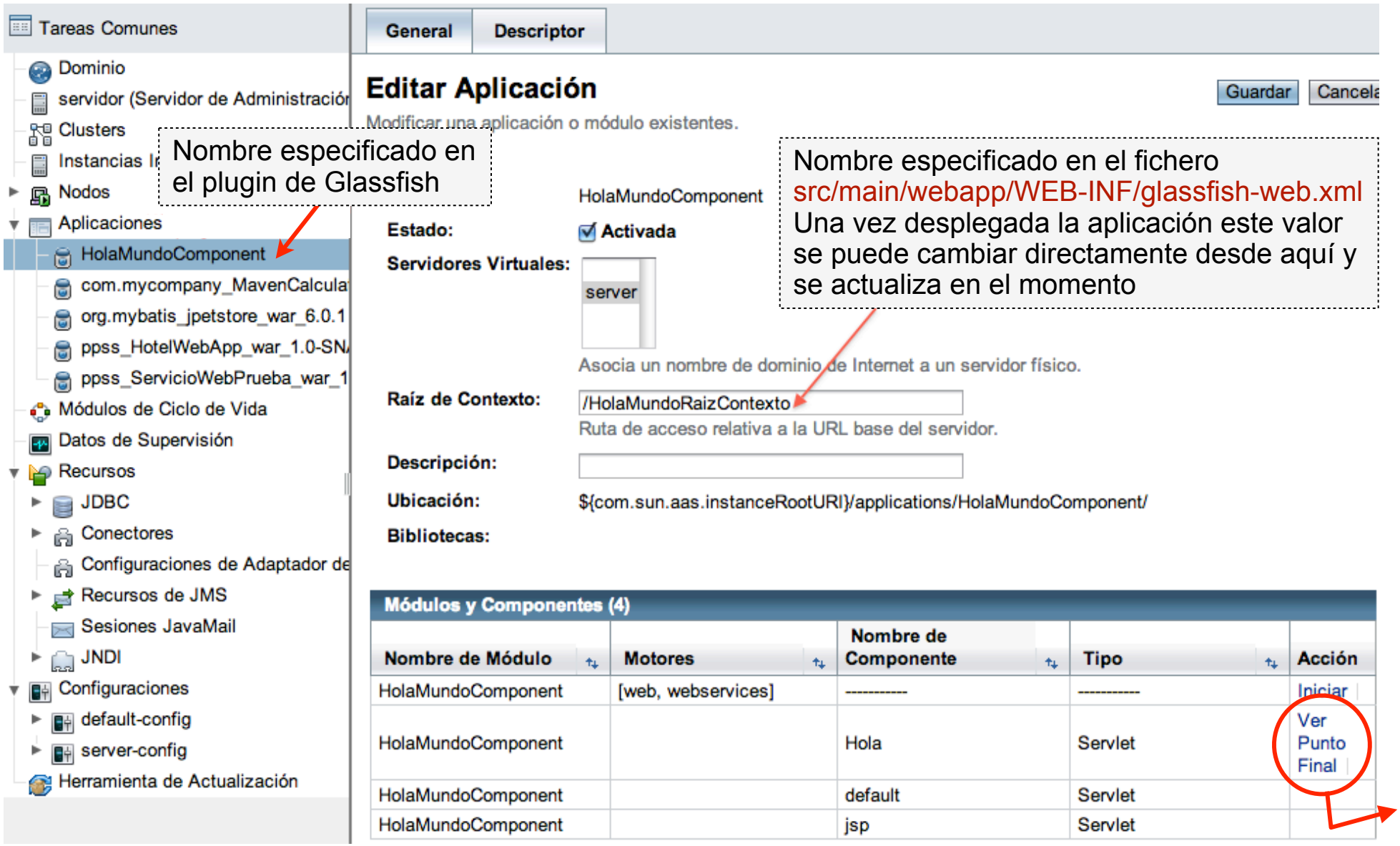
- Utilizaremos como ejemplo la clase `jtech.Hola` como clase que implementa nuestro servicio Web
- Comenzamos creando una aplicación Web con Maven
- La meta ***jaxws:wsgen***, "lee" la clase que implementa un *endpoint* y genera todos los artefactos necesarios para crear el servicio Web. Por defecto, los fuentes y clases generadas no se almacenan en el disco
- Para generar un servicio Web con Maven NO es necesario especificar la ejecución de *wsgen* en el pom. Se ejecuta automáticamente en la fase del ciclo de vida de Maven *process-classes*

Creación de la clase que implementa el WS

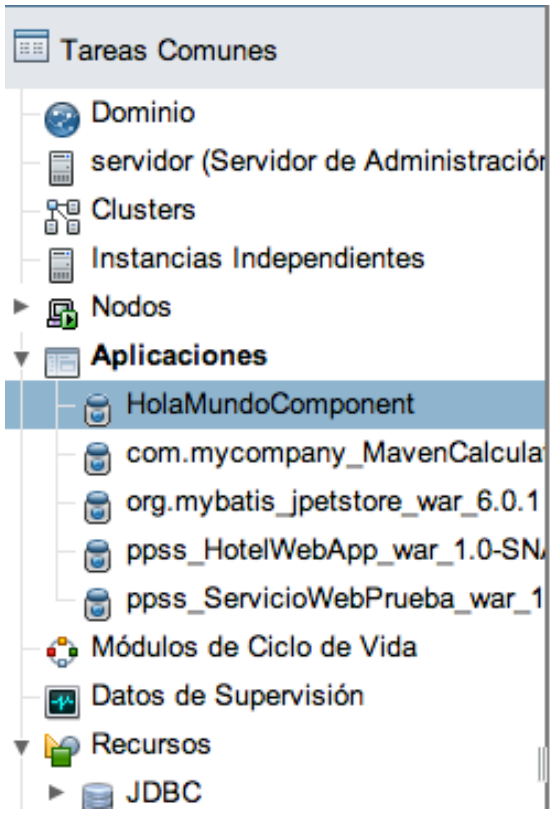
Por defecto, el nombre del servicio (atributo serviceName de WebService, y etiqueta <service> del wsdl) es el nombre de la clase con el sufijo Service

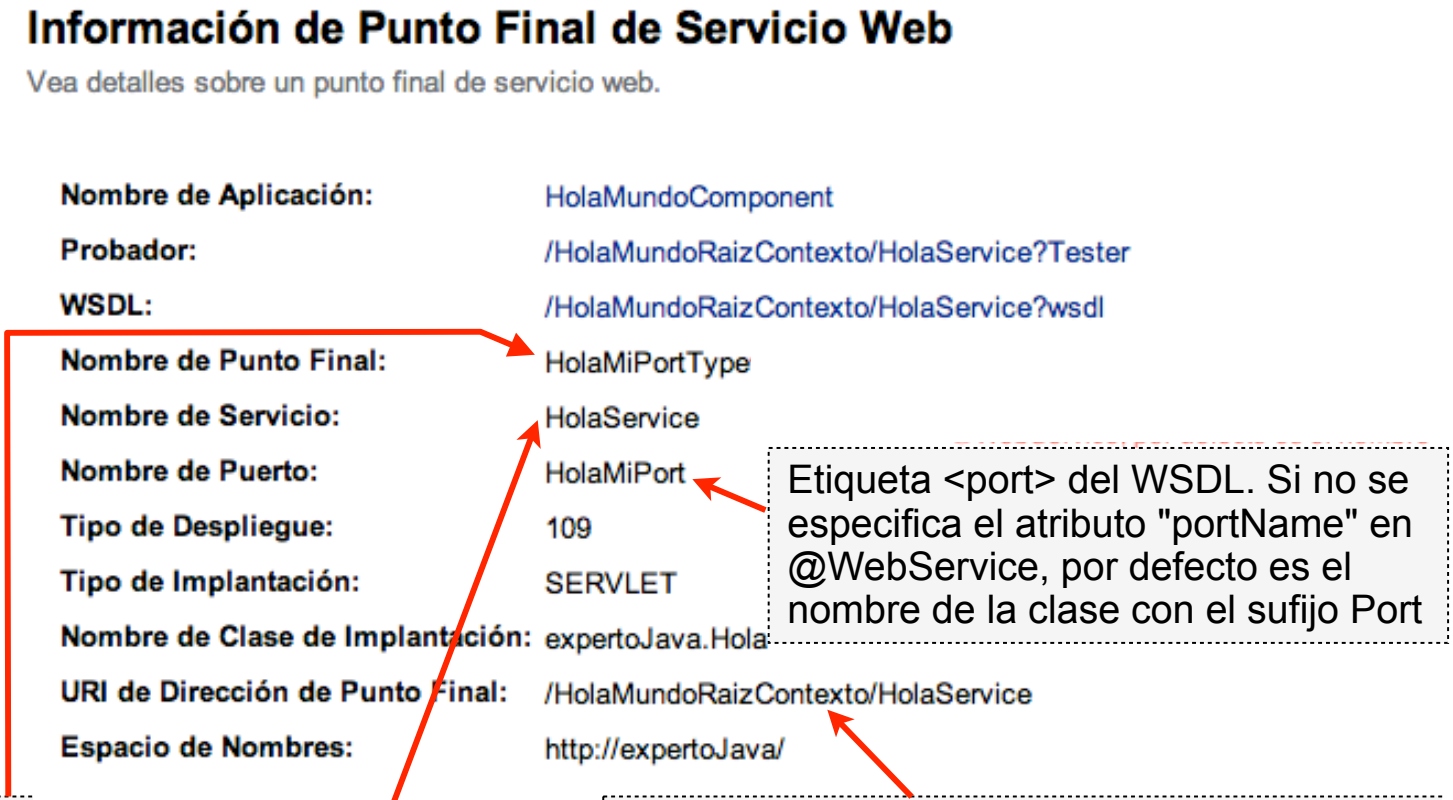
```
package expertoJava;

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

@WebService(name="HolaMiPortType", portName="HolaMiPort")
public class Hola {

    @WebMethod(operationName = "hello_operation_name")
    public String hello(@WebParam(name = "name") String txt) {
        return "Hola " + txt + " !";
    }
}
```

Componente desplegado en Glassfish (I)



Componente desplegado en Glassfish (II)





Etiqueta <port> del WSDL. Si no se especifica el atributo "portName" en @WebService, por defecto es el nombre de la clase con el sufijo Port

Etiqueta <portType> del WSDL. Si no se especifica el atributo "name" en @WebService, por defecto es el nombre de la clase

Etiqueta <service> del WSDL. Si no se especifica el atributo "serviceName" en @WebService, por defecto es el nombre de la clase con el sufijo Service

Especificada en el fichero src/main/webapp/WEB-INF/glassfish-web.xml Por defecto es el nombre del artefacto war generado por Maven en el directorio target

WSDL generado por wsimport (I)

/HolaMundoRaizContexto/HolaService?wsdl

```
<!--
 Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is Metro/2.2.0-1
(tags/2.2.0u1-7139; 2012-06-02T10:55:19+0000) JAXWS-RI/2.2.6-2 JAXWS/2.2 ... -->
<!--
 Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is Metro/2.2.0-1
(tags/2.2.0u1-7139; 2012-06-02T10:55:19+0000) JAXWS-RI/2.2.6-2 JAXWS/2.2 ...  -->

<definitions xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
                        oasis-200401-wss-wssecurity-utility-1.0.xsd"...
             targetNamespace="http://expertoJava/" name="HolaService">

<types>
   <xsd:schema>
      <xsd:import namespace="http://expertoJava/"
          schemaLocation=
            "http://pc-eli.dccia.ua.es:8080/HolaMundoRaizContexto/HolaService?xsd=1"/>
   </xsd:schema>
</types>

<message name="hello_operation_name">
   <part name="parameters" element="tns:hello_operation_name"/>
</message>

<message name="hello_operation_nameResponse">
   <part name="parameters" element="tns:hello_operation_nameResponse"/>
</message>

...
```

WSDL generado (II)

/HolaMundoRaizContexto/HolaService?wsdl

Especificado en el atributo "name" en @WebService, por defecto es el nombre de la clase con el sufijo Port

```
<portType name="HolaMiPortType">
   <operation name="hello_operation_name">
      <input wsam:Action=
             "http://expertoJava/HolaMiPortType/hello_operation_nameRequest"
             message="tns:hello_operation_name"/>
      <output wsam:Action=
              "http://expertoJava/HolaMiPortType/hello_operation_nameResponse"
             message="tns:hello_operation_nameResponse"/>
   </operation>
</portType>

<binding name="HolaMiPortBinding" type="tns:HolaMiPortType">
   <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
        style="document"/>
   <operation name="hello_operation_name">
      <soap:operation soapAction=""/>
      <input>
         <soap:body use="literal"/>
      </input>
      <output>
         <soap:body use="literal"/>
      </output>
   </operation>
</binding>
```

WSDL generado (III)

/HolaMundoRaizContexto/HolaService?wsdl

Atributo "serviceName" de @WebService

```
<service name="HolaService">
    <port name="HolaMiPort" binding="tns:HolaMiPortBinding">
        <soap:address location=
            "http://pc-eli.dccia.ua.es:8080/HolaMundoRaizContexto/
HolaService"/>
    </port>
</service>
</definitions>
```

Atributo "portName" de @WebService

URL del servicio

- El wsdl del servicio se genera AUTOMÁTICAMENTE durante el despliegue
- Podremos probarlo desde: /HolaMundoRaizContexto/HolaService?Tester

Creación de servicios con Netbeans

- En Netbeans tenemos asistentes para crear servicios web en un proyecto web o módulo EJB
  - Nos permiten exponer funcionalidades de la aplicación

- Se pueden crear:
  - En una nueva clase Java
    - *New > Web Service , Add operation , Test Web Service*
  - A partir de un EJB existente
    - *New > Web Service (create Web service from Session Bean)*
  - A partir del WSDL
    - *New > Web Service from WSDL*

Servicios Web a partir de EJBs existentes

```
package jtech;

import javax.ejb.EJB;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;

@WebService(serviceName = "ConversionSW")
public class ConversionSW {
    @EJB
    private jtech.ConversionEJBBeanLocal ejbRef;

    @WebMethod(operationName = "euro2ptas")
    public int euro2ptas(@WebParam(name = "euros") double euros) {
        return ejbRef.euro2ptas(euros);
    }

    @WebMethod(operationName = "ptas2euros")
    public double ptas2euros(@WebParam(name = "ptas") int ptas) {
        return ejbRef.ptas2euros(ptas);
    }
}
```

el EJB existente se inyecta en la clase que implementa el servicio web

el servicio web realiza llamadas al EJB

Paso de datos binarios

- Supongamos que queremos enviar datos binarios, por ejemplo una imagen en formato *jpg*
- Por defecto, si en un mensaje SOAP incluimos datos binarios, éstos se codifican con el tipo `base64Binary` y el cliente es el que tiene que saber cómo interpretar los datos
  - si enviamos un dato con el tipo `java.awt.Image`
  - el elemento asociado: `<element name="image" type="base64Binary"/>`
  - por defecto se mapeará en el cliente como el tipo `byte []`
- Para poder enviar un java.awt.Image y que el cliente lo reciba como tal, tenemos que:
  - Añadir el atributo `expectedContentTypes="mime_type"` en el fichero de esquema
  - Utilizar en el servicio web el wsdl con el esquema con la nueva configuración

Generamos y guardamos el wsdl (y xsd)

- Por defecto, el wsdl generado no se almacena en ningún sitio, vamos a guardar el fichero generado en target/jaxws/wsgen/wsdl

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>jaxws-maven-plugin</artifactId>
    <version>1.10</version>
     <executions>
       <execution>
           <goals> <goal>wsgen</goal> </goals>
           <configuration>
             <sei>jtech.floweralbumservice.FlowerService</sei>
             <genWsdl>true</genWsdl>
           </configuration>
       </execution>
     </executions>
     <dependencies>
        <dependency>
           <groupId>javax</groupId>
           <artifactId>javaee-web-api</artifactId>
           <version>6.0</version>
        </dependency>
     </dependencies>
</plugin>
```

el wsdl+ xsd generados se almacenarán por defecto en target/jaxws/wsgen

Cambios en el fichero de esquema

- Supongamos que un servicio web tiene definido el siguiente tipo en el fichero de esquema:

```
<xsd:complexType name="getFlowerResponse">
  <xsd:sequence>
    <xsd:element name="return" type="xs:base64Binary" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

- La modificación a realizar será:

```
<xsd:complexType name="getFlowerResponse">
  <xsd:sequence>
    <xsd:element name="return" type="xs:base64Binary" minOccurs="0"
      xmime:expectedContentTypes="image/jpeg"
      xmlns:xmime="http://www.w3.org/2005/05/xmlmime"/>
  </xsd:sequence>
</xsd:complexType>
```

Utilizamos el wsdl con el esquema modificado

- Copiaremos los ficheros generados en el directorio src/main/resources. Por defecto, los ficheros aquí incluidos se copiarán en el war generado, en el directorio WEB-INF/classes
- Indicamos de forma explícita, con el atributo wsdlLocation que queremos usar nuestra propia versión del fichero wsdl. (Si no lo hacemos así, el servidor de aplicaciones generará su propio fichero wsdl)

```
//fichero FlowerService.java
@WebService(serviceName="FlowerService",
        wsdlLocation = "WEB-INF/classes/FlowerService.wsdl")
@Stateless
public class FlowerService {
    ...
    @WebMethod(operationName="getFlower")
    public Image getFlower(@WebParam(name="name") String name) throws
IOException {
      ...
    }
    ...
```

Servicios con estado

- Mantienen información de estado de cada cliente
  - Por ejemplo un carrito de la compra
    - Cada llamada al servicio añade un producto al carrito
  - Disponible a partir de JAX-WS 2.1
- Cada cliente accede a una instancia del servicio
  - El estado se mantiene mediante variables de instancia
- Basado en *WS-Addressing*. Permite especificar:
  - Dirección del *endpoint*
  - Instancia concreta del servicio a la que acceder

Ejemplo de servicio *stateful*

```
@Stateful
@WebService
@Addressing
public class CuentaSW {
  private int id; private int saldo;

  public CuentaSW(int id) { this.id = id; this.saldo = 0; }

  public void ingresar(int cantidad) { saldo += cantidad; }
  public int saldo() { return saldo; }
  public void cerrar() { manager.unexport(this); }

  public static StatefulWebServiceManager<CuentaSW> manager;
}
```

el contenedor inyecta de forma automática un objeto de tipo StatefulWebServiceManager

Crear instancias

- Utilizamos un servicio adicional

devuelve una referencia a un endpoint (implementación del servicio), concretamente a la instancia concreta del servicio CuentaSW

```
@WebService
public class BancoSW {
  static Map<Integer, CuentaSW> cuentas = new HashMap();


  @WebMethod public synchronized W3CEndpointReference
                                              abrirCuenta(int id) {
    CuentaSW c = cuentas.get(id);
    if (c == null) {
      c = new CuentaSW(id);
      cuentas.put(id, c);
    }
    W3CEndpointReference endpoint = CuentaSW.manager.export(c);
    return endpoint;
  }
```

Cliente de servicios stateful

```
BancoSWService bService = new BancoSWService();
CuentaSWService cService = new CuentaSWService();
BancoSW bPort = bService.getBancoSWPort();


W3CEndpointReference endpoint = bPort.abrirCuenta(1);
CuentaSW c = cService.getPort(endpoint,CuentaSW.class);

c.ingresar(10);
c.ingresar(5);
out.println("Saldo: " + c.saldo());
c.ingresar(20);
out.println("Nuevo saldo: " + c.saldo());
c.cerrar();
```

versión alternativa de getPort sobre el Service CuentaSWService

¿Preguntas...?