



CIFP César Manrique

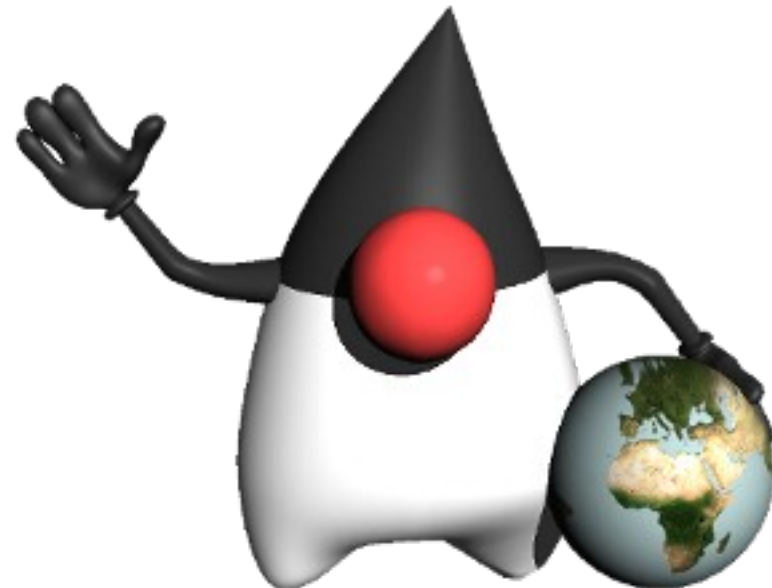
SOAP y REST

**CFGS DAW – Desarrollo web en entorno
servidor**



Web Services SOAP y REST

<https://docs.oracle.com/javaee/7/tutorial/webservices-intro001.htm>





CONTENIDOS

- Introducción a servicios SOAP y REST
- SOAP Web Services
- Creación de SOAP Web Services con JAX-WS
- RESTful services
- Creación de RESTful services con JAX-RS



¿Qué son los Web Service?

- Son aplicaciones que siguen una interacción tipo cliente y servidor, comunicándose mediante el protocolo **HTTP**
- Posibilitan la interacción entre aplicaciones heterogéneas (desarrolladas mediante diferentes tecnologías y que se ejecutan en diferentes entornos hardware-software), gracias al uso de estándares basados en **XML/SOAP** u otros formatos como **JSON**
- Los web service se caracterizan por su gran interoperabilidad y extensibilidad, así como la facilidad de ser descubiertos y analizados de forma automática, gracias al empleo de descriptores estándar
- Los servicios web pueden combinarse, logrando desarrollar operaciones muy complejas, con la ventaja de ser soluciones poco acopladas, gracias al uso de estándares y el empleo de XML



Tipos de web service

- Dos tipos principales: “big” web services (SOAP) y RESTful web services (REST)
- **Big web service /SOAP web service.** Son los web service tradicionales. Usan mensajes XML que siguen el estándar SOAP (Simple Object Access Protocol)

En java el API que utilizaremos será **JAX-WS**

- **RESTful web service.** Los conocidos como servicios REST (Representational State Transfer). Son de más reciente adopción y son más simples dado que no requieren de mensajes XML o definiciones del API WSDL. En el intercambio de mensajes suelen utilizar JSON

En java el API que utilizaremos será **JAX-RS**

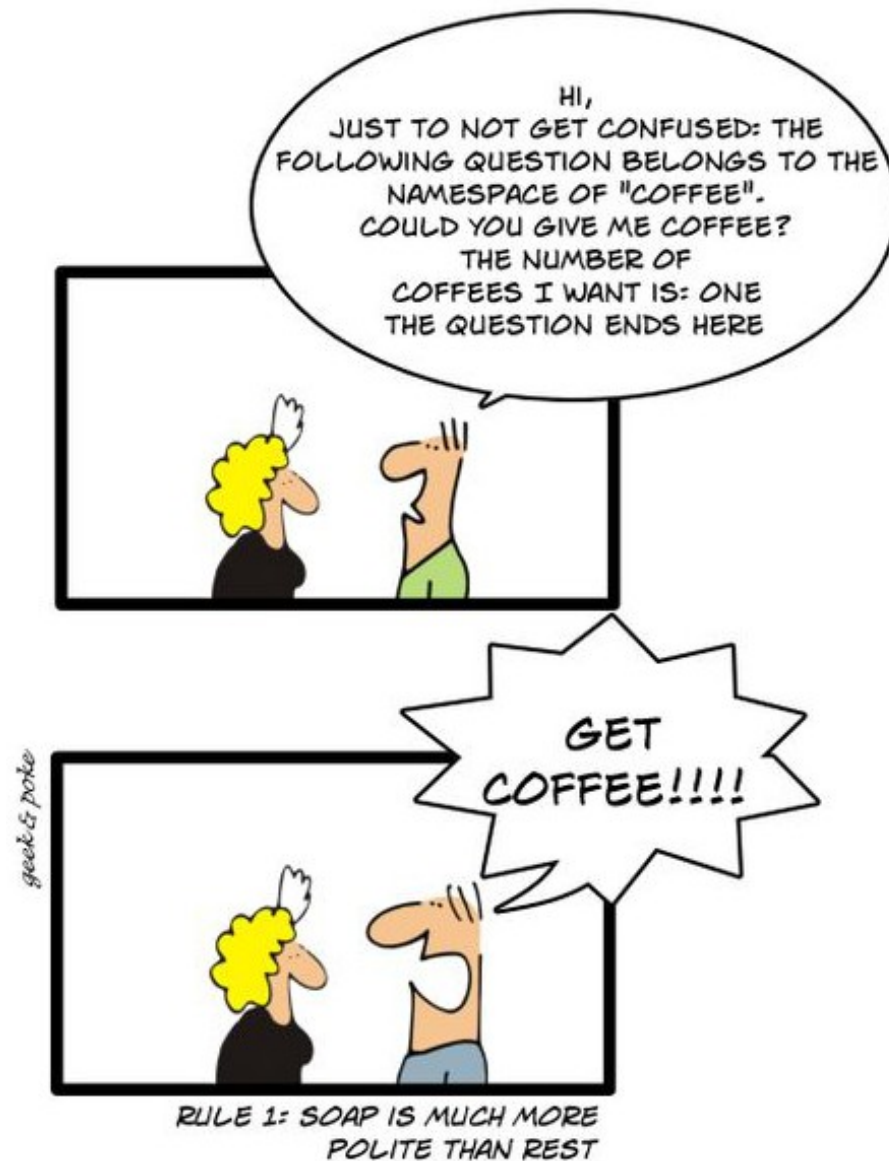


Web service XML/SOAP (JAX-WS) →

"Service Calling made easy"
by Geek and Poke

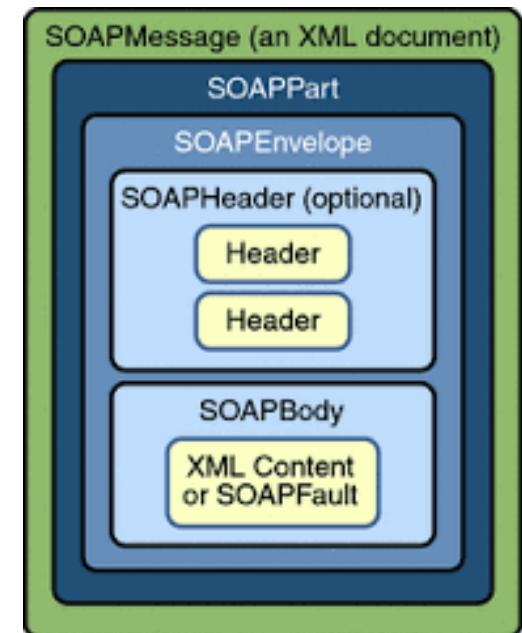
Web service REST (JAX-RS) →

SERVICE CALLING MADE EASY



Web Services o SOAP Services

- Estos servicios incluyen una descripción de las operaciones, escrita en lenguaje WSDL (Web Service Description Language)
- El intercambio de datos entre el servicio y el cliente del servicio se realiza mediante **mensajes** SOAP
- Los mensajes SOAP son documentos XML, que constan de:
 - Envelope: elemento raíz, contiene dos elementos:
 - header (opcional). Información para la app.
 - body (mandatorio). Contenido para destinatario
 - Puede contener elemento SOAPFault, para indicar errores





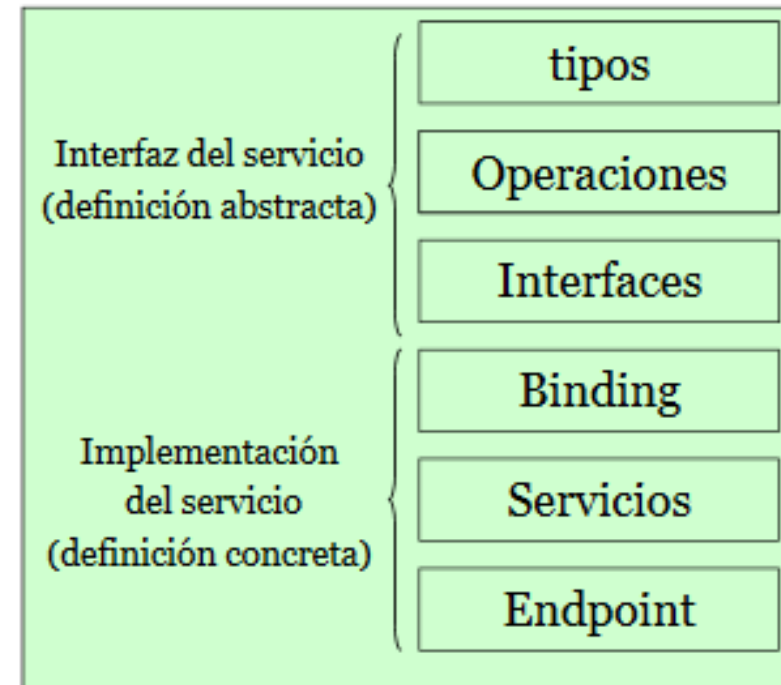
Ejemplo de mensaje SOAP

```
<?xml version='1.0' Encoding='UTF-8' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <m:reservation xmlns:m="http://travelcompany.example.org/reservation"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next">
      <m:reference>uuid:093a2da1-q345-739r-ba5d-pqff98fe8j7d</m:reference>
      <m:dateAndTime>2007-11-29T13:20:00.000-05:00</m:dateAndTime>
    </m:reservation>
    <n:passenger xmlns:n="http://mycompany.example.com/employees"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next">
      <n:name>Fred Bloggs</n:name>
    </n:passenger>
  </env:Header>
  <env:Body>
    <p:itinerary xmlns:p="http://travelcompany.example.org/reservation/travel">
      <p:departure>
        <p:departing>New York</p:departing>
        <p:arriving>Los Angeles</p:arriving>
        <p:departureDate>2007-12-14</p:departureDate>
        <p:departureTime>late afternoon</p:departureTime>
        <p:seatPreference>aisle</p:seatPreference>
      </p:departure>
      <p:return>
        <p:departing>Los Angeles</p:departing>
        <p:arriving>New York</p:arriving>
        <p:departureDate>2007-12-20</p:departureDate>
        <p:departureTime>mid-morning</p:departureTime>
        <p:seatPreference></p:seatPreference>
      </p:return>
    </p:itinerary>
  </env:Body>
</env:Envelope>
```




Web Service Description Language (WSDL) o güisdel ;-)

- Son documentos XML que describen el servicio Web
- En la interfaz del servicio se definen los tipos de datos usados en los mensajes, la signatura de los métodos (operaciones), la estructura de la interfaz o port-type del servicio
- En la implementación se define el *binding* (una característica de los servicios SOAP es que soportan diferentes medios de transporte, más allá de HTTP (ejemplo SMTP), se define el estilo DOCUMENT/RPC y los endpoint, indicando las direcciones de acceso al web service





Ejemplo de WSDL

- Operation
- Port type
- Binding
- End point

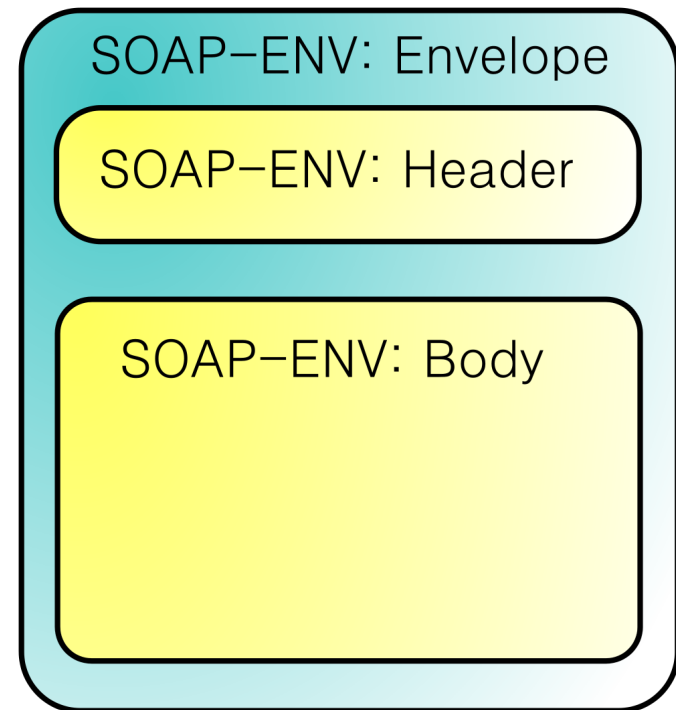
```

</xsd:schema>
</types>
<message name="DFHOXCMNOperationResponse">
  <part element="resns:DFHOXCMNOperationResponse" name="ResponsePart"/>
</message>
<message name="DFHOXCMNOperationRequest">
  <part element="reqns:DFHOXCMNOperation" name="RequestPart"/>
</message>
<porttype name="DFHOXCMNPort">
  <operation name="DFHOXCMNOperation">
    <input message="tns:DFHOXCMNOperationRequest" name="DFHOXCMNOperationRequest"/>
    <output message="tns:DFHOXCMNOperationResponse" name="DFHOXCMNOperationResponse"/>
  </operation>
</porttype>
<binding name="DFHOXCMNHTTPSoapBinding" type="tns:DFHOXCMNPort">
  <!-- This soap:binding indicates the use of SOAP 1.1 -->
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <!-- This soap:binding indicates the use of SOAP 1.2 -->
  <!-- <soap:binding style="document" transport="http://www.w3.org/2003/05/soap-http"/> -->
  <operation name="DFHOXCMNOperation">
    <soap:operation soapAction="" style="document"/>
    <input name="DFHOXCMNOperationRequest">
      <soap:body parts="RequestPart" use="literal"/>
    </input>
    <output name="DFHOXCMNOperationResponse">
      <soap:body parts="ResponsePart" use="literal"/>
    </output>
  </operation>
</binding>
<service name="DFHOXCMNService">
  <port binding="tns:DFHOXCMNHTTPSoapBinding" name="DFHOXCMNPort">
    <soap:address location="http://my-server:my-port/exampleApp/inquireSingles.log"/>
  </port>
</service>
/definitions>

```

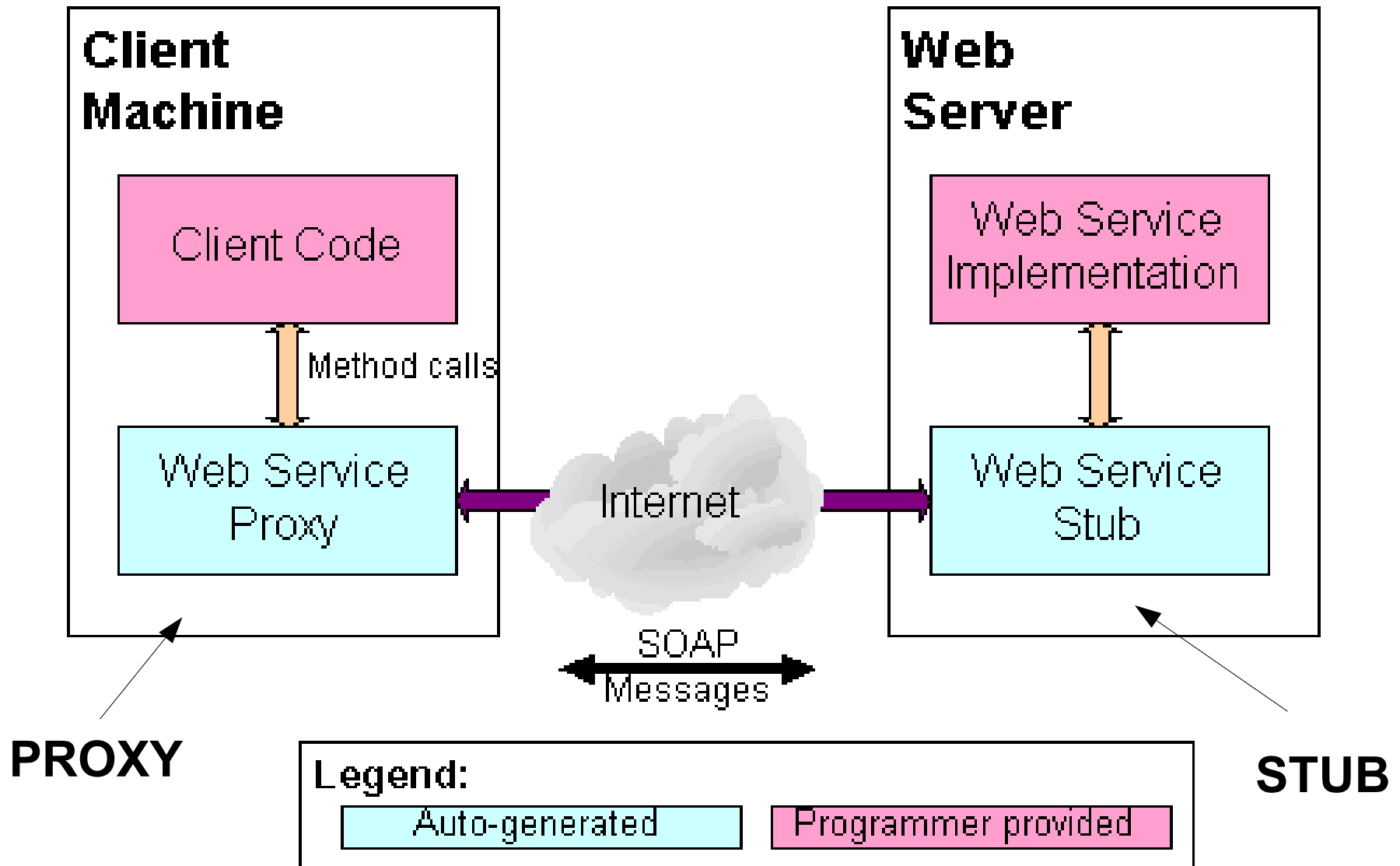


JAX-WS Web services (SOAP web services)





Creación de un SOAP web service con JAX-WS





Creación de un SOAP web service con JAX-WS

- Pasos

- Programar las clases de implementación del web service
- Compilar las clases de implementación
- Empaquetarlo todo en un módulo WAR
- Desplegar el WAR
 - Glassfish genera los artefactos (**stub**) necesarios para comunicar con los clientes durante el despliegue
- Programar las clases del código cliente
- Generar el **proxy** o clases necesarias para conectar e invocar métodos del webservice
- Compilar código cliente
- Ejecutar



JAX-WS: Requisitos de la clase de implementación del endpoint

- La clase de implementación debe anotarse como **@javax.jws.WebService** **@javax.jws.WebServiceProvider**. Debe tener un constructor público por defecto, no puede ser abstracta ni final, y no puede definir el método finalize
- La clase de implementación debería indicar la interfaz que implementa o SEI (Service endpoint interface) a través del elemento endpointInterface de @WebService , sino se define de forma implícita la interfaz (automáticamente)

```
@WebService(serviceName = "EchoService", endpointInterface =  
"com.ibm.was.wssample.echo.EchoServicePortType", portName="EchoServicePort")  
public class EchoServicePortTypeImpl implements EchoServicePortType {  
    ...  
}
```

- Los métodos de negocio (*business methods*) deben ser públicos, no pueden ser static o final y deben estar anotados con **@javax.jws.WebMethod**. Sus parámetros de entrada y tipos de retorno compatibles JAXB (Java Architecture for XML Binding)
- La clase de implementación puede usar las anotaciones @javax.annotation.PostConstruct o @javax.annotation.PreDestroy en sus métodos que implementen llamadas de ciclo de vida.
 - El método @PostConstruct method es invocado por el contenedor previo a peticiones de clientes
 - El método @PreDestroy method se llama antes de que el endpoint se retire



Creando la clase de implementación

- Creamos un proyecto: Maven > Web > Java EE 7 > HelloSoap
- Dentro creamos la clase de implementación **EchoService** (no creamos Interface, se crea de forma implícita)

```
package es.cifpcm.hellosoap.services;  
import javax.jws.WebService;  
import javax.jws.WebMethod;  
import javax.jws.WebParam;
```

```
@WebService(serviceName = "EchoService")
```



Web service endpoint

```
public class EchoService {
```

```
    @WebMethod(operationName = "reply")
```



Operación del web service

```
    public String reply(@WebParam(name = "message") String message) {
```

```
        return "Echo... " + message;
```

```
    }
```



Parámetro

```
}
```



Prueba del web service

- Ejecutamos el proyecto HelloSoap y probamos el web service:

<http://localhost:8080/HelloSoap/EchoService?tester>

- Podemos ver el WSDL generado

<http://localhost:8080/HelloSoap/EchoService?wsc>

```
<!--
  Published by JAX-WS RI (http://jax-ws.java.net). RI's version is Metro/2.3.2-b608 (trunk-7979; 2011-08-16T14:22:29Z)
-->
<!--
  Generated by JAX-WS RI (http://jax-ws.java.net). RI's version is Metro/2.3.2-b608 (trunk-7979; 2011-08-16T14:22:29Z)
-->
<definitions targetNamespace="http://services.hellosoap.cifpcm.es/" name="EchoService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://services.hellosoap.cifpcm.es/" schemaLocation="http://localhost:8080/HelloSoap/EchoService?xsd=1"/>
    </xsd:schema>
  </types>
</definitions>
```




Creando el cliente web JAX-WS

- Creamos un proyecto: Maven > Web > Java EE 7 > HelloClient
- Añadimos un Web Service Client
 - El proxy podemos crearlo a partir de la URL del WSDL

<http://localhost:8080/HelloSoap/EchoService?wsdl>

O seleccionando proyecto, que ejecuta automáticamente el JAX-WS:ws:import

WSDL and Client Location

Specify the WSDL file of the Web Service.

☒ Project:

☐ Local File:

☐ WSDL URL:

☐ IDE Registered:

Specify a package name where the client java artifacts will be generated:

Project: HelloClient

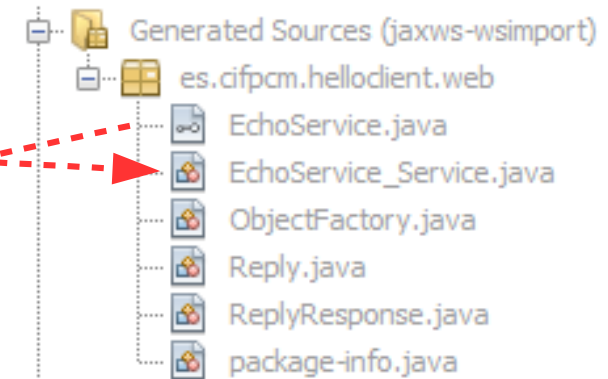
Package:

☐ Generate Dispatch code

Utilizando el proxy para acceder al servicio

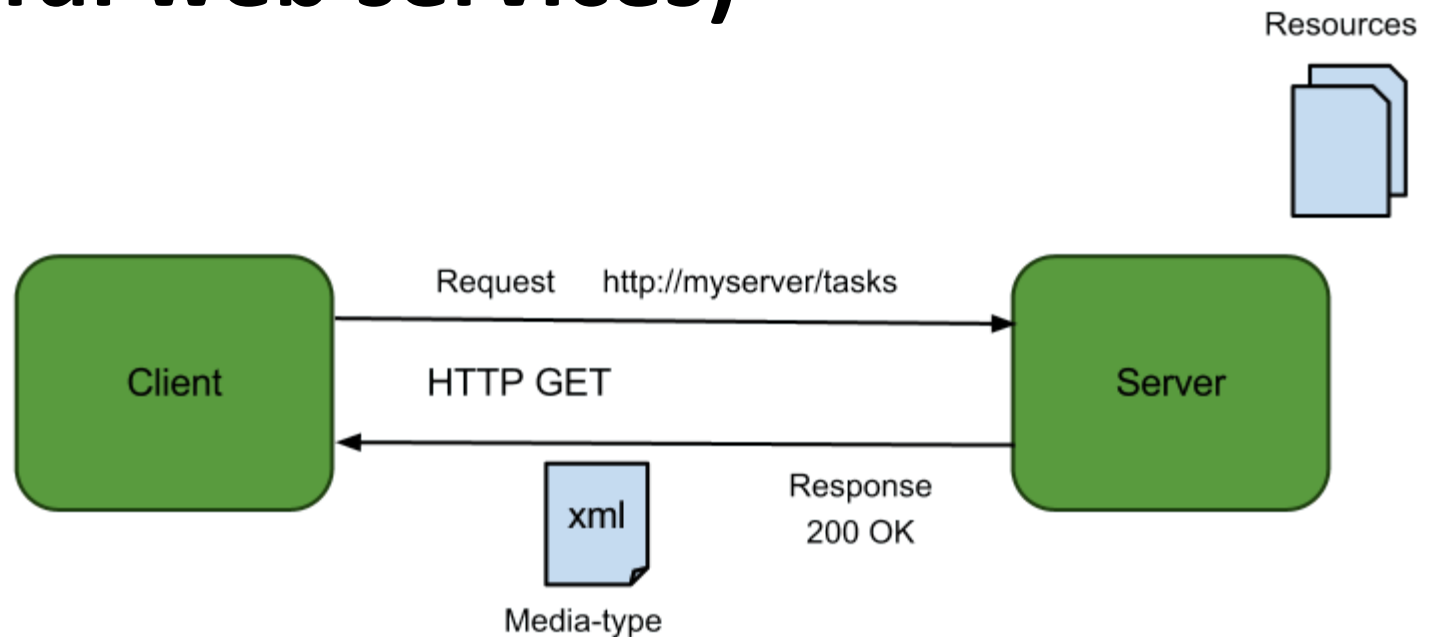
- El target Maven ws:import, invocado con el asistente anterior, ha generado un conjunto de clases Java, donde encontramos el proxy; la clase que vamos a utilizar para llamar al servicio
- Creamos un servlet cliente y en él vamos a utilizar el proxy para acceder al webservice

```
EchoService_Service echoSrvProxy = new  
    EchoService_Service();  
  
EchoService echoService =  
    echoSrvProxy.getEchoServicePort();  
  
String reply = echoService.reply("Hola mundo");
```





JAX-RS Web services (RESTful web services)





REST

- REST (REpresentational State Transfer) es un estilo de arquitectura propuesto por Roy Fielding para el diseño de servicios web
- Ha ganado mucha popularidad como alternativa ligera al uso de los tradicionales web service (SOAP)
- Los servicios REST usan HTTP como medio de comunicación
- La filosofía de REST se basa en manipular recursos, accesibles mediante una URI, valiéndose de los métodos HTTP para definir las operaciones sobre esos recursos
- Podemos ver una analogía entre las operaciones CRUD (create, read, update, delete) y los verbos HTTP

<https://geekytheory.com/json-i-que-es-y-para-que-sirve-json/>

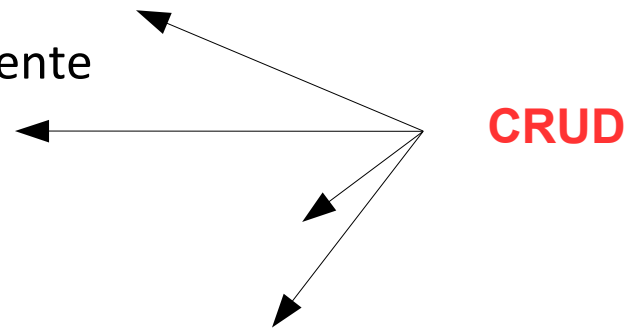


REST

- En REST utilizamos sustantivos para definir los recursos. Por ejemplo para manipular información de empleados podríamos utilizar el recurso

<http://EmployeeService/Employee>

- Nos valemos de la semántica de las operaciones HTTP (verbos HTTP) para definir operaciones sobre ese recurso. Por ejemplo:
 - **POST** → **C**reate. Crear un nuevo recurso
 - **GET** → **R**ead. Obtener el recurso
 - **PUT** → **U**ppdate. Modificar un recurso existente
 - **DELETE** → **D**elete. Borrarr un recurso





Representación

- REST admite múltiples formatos a la hora de exponer los recursos a los clientes
- La información puede serializarse como XML, JSON, o cualquier otro formato
- En línea con la idea de ligereza y simplicidad, el formato de representación más usado es JSON
- Ejemplo de representación de objetos JSON

```
{ "NombreFruta":"Manzana" , "Cantidad":20 }
```

```
{  
  "Frutas": [  
    { "NombreFruta":"Manzana" , "cantidad":10 },  
    { "NombreFruta":"Pera" , "cantidad":20 },  
    { "NombreFruta":"Naranja" , "cantidad":30 }  
  ]  
}
```



Características de REST

- Protocolo cliente-servidor sin estado
- Basándose en el protocolo HTTP se pueden implementar sistemas de caché de peticiones
- Los objetos REST siempre se manipulan a través de la URI
- Interfaz uniforme: para la transferencia de datos en un sistema REST, este aplica acciones concretas (POST, GET, PUT y DELETE) sobre los recursos
- Arquitectura en capas
- Dado que se basa en un protocolo sin estado, la forma de conseguir una interacción stateful pasa por “pasarse” la información de estado en las peticiones. Veremos un ejemplo



Interacción REST. Principio HATEOAS

- Para cualquier API REST es obligatorio disponer del principio HATEOAS (Hypermedia As The Engine Of Application State - Hipermedia Como Motor del Estado de la Aplicación) para ser una verdadera API REST. Este principio es el que define que cada vez que se hace una petición al servidor y éste devuelve una respuesta, parte de la información que contendrá serán los hipervínculos de navegación asociada a otros recursos del cliente
- Ejemplo práctico de una petición a una base de datos de automóviles):

```
{
  "id": 78,
  "nombre": "Juan",
  "apellido": "García",
  "coches": [
    {
      "coche":
        "http://miservidor/concesionario/api/v1/clientes/78/coches/1033"
    },
    {
      "coche":
        "http://miservidor/concesionario/api/v1/clientes/78/coches/3889"
    }
  ]
}
```




JAX-RS

- Java API for RESTful Web Services (JAX-RS, definido en JSR 339)
- Los recursos REST o Root resource classes se basan en clases POJO "plain old Java objects"
- Estas clases o bien tienen la anotación `@Path` o al menos uno de sus métodos lleva la anotación `@Path` o un *request method designator*: `@GET`, `@PUT`, `@POST`, or `@DELETE`.
- Los métodos del recurso, son métodos de estas clases anotados con un *request method designator*



Ejemplo de clase de recurso REST

```
/**  
 * Root resource (exposed at "helloworld" path)  
 */  
@Path("helloworld")  
public class HelloWorld {  
    @Context  
    private UriInfo context;  
  
    public HelloWorld() {  
    }  
  
    @GET  
    @Produces("text/html")  
    public String getHtml() {  
        return "<html lang=\"en\"><body><h1>Hello, World!!</h1></body></html>";  
    }  
}
```

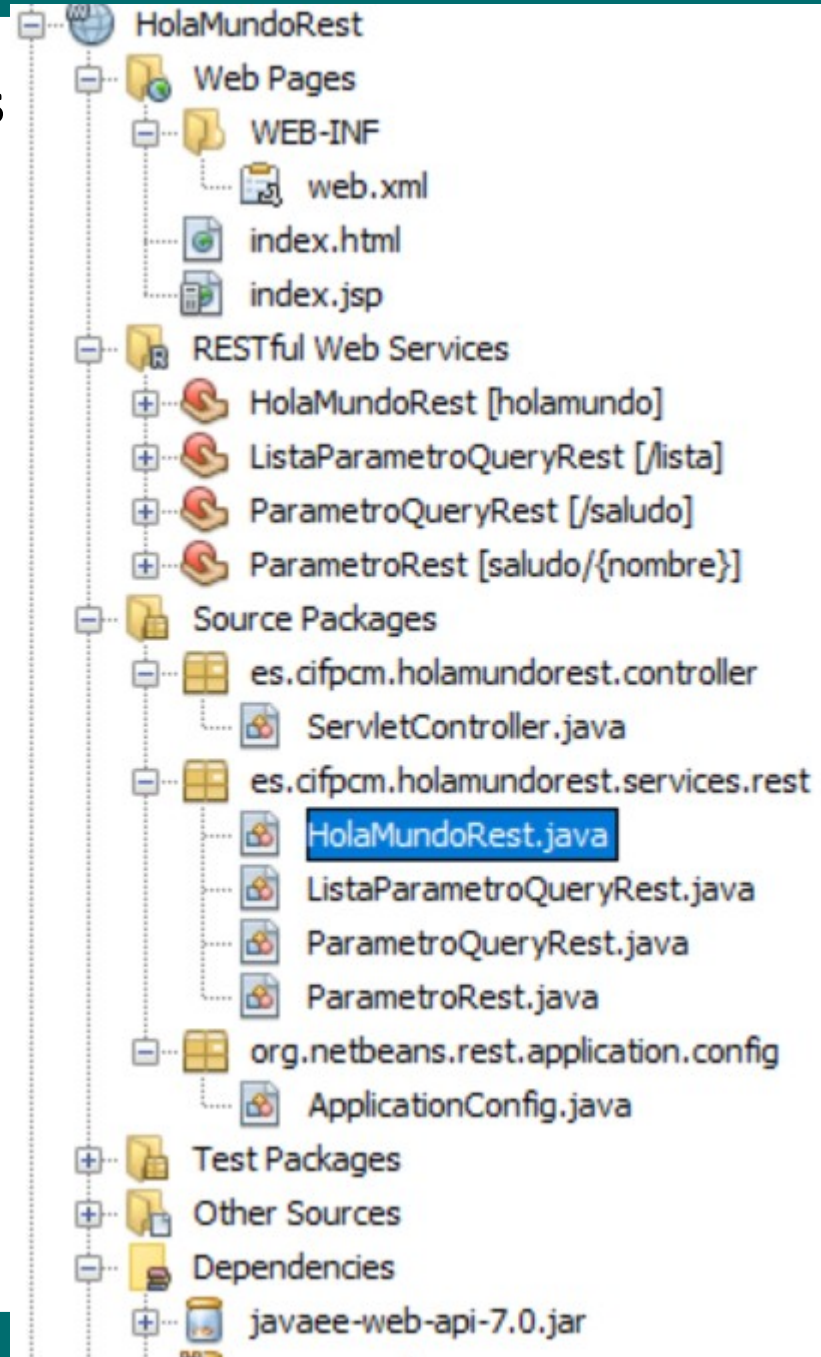


Ejemplo “Hola Mundo” Cruzamos

Usaremos la librería JAX-RS para usar un servicio web RESTful básico de forma sencilla.

- El servicio REST está representado por una clase donde añadiremos las anotaciones. HolaMundoRest

-





HolaMundoRest.java

```

/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package es.cifpcm.holamundorest.services.rest;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

/**
 *
 * @author inmav
 */
@Path("holamundo")
public class HolaMundoRest {
    public HolaMundoRest(){}

    @GET
    @Produces("text/html")
    public String getHtml(){
        return "<html lang=\"en\"><body><h1>HolaMundo!!</h1></body></html>";
    }
}

```




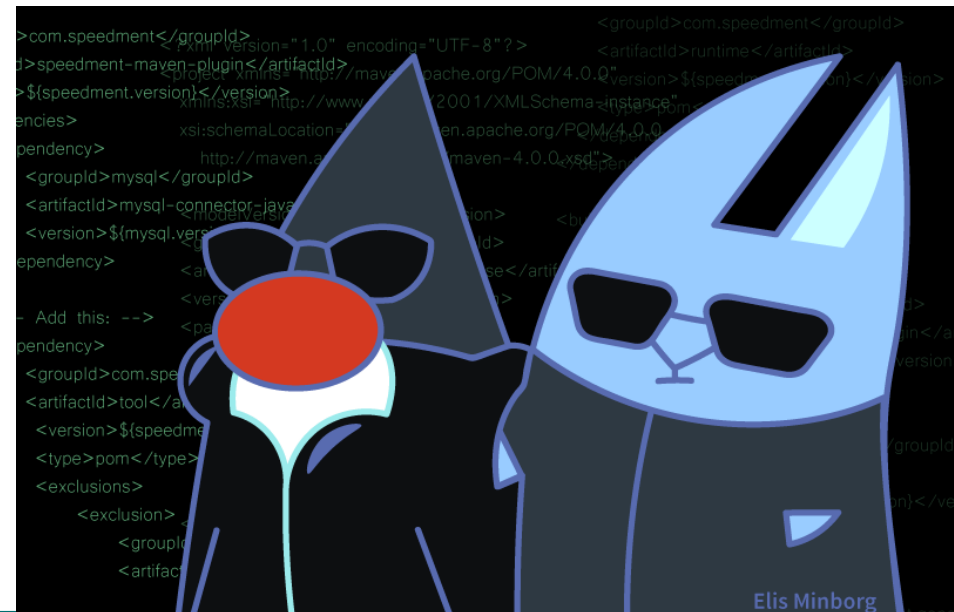
Index.html

```
<!DOCTYPE html>
```

-
- <html>
- <head>
- <title>Ejemplos de Línea de Código sobre Java EE</title>
- </head>
- <body>
- <h1>Ejemplos de Línea de Código sobre Java EE</h1>
-
- <h2>JAX-RS: Servicios RESTful</h2>
-
-
- <p>Servicio REST Hola Mundo
- <p>Servicio REST con parámetro
- <p>Servicio REST con parámetro por query (valor Luis)
- <p>Servicio REST con parámetro
-
-
-
-
- </body>
- </html>
-

Ejemplo en Netbeans

- <https://netbeans.org/kb/docs/websvc/rest.html?print=yes>
- Problema con Glassfish 4.1.1 (¿a quién hay que...  ?)
 - <http://entjavastuff.blogspot.com.es/2015/10/maven-javaee7-app-in-netbeans-8.html>
 - <https://java.net/jira/browse/GLASSFISH-21438>





Ejemplo con BD Sakila

- Cread una clase POJO para almacenar los datos del actor
En mi caso Actor o ActorVo (**vo** de value object-). Que sea JavaBean!!!
- Crear una clase POJO para el recurso REST
Ejemplo: ActorRest, en un packageservices.rest
- Anotar la clase con @Path (“actors”) o similar
(os pedirá que configuréis REST según Java EE 6)
- Cread un método **read** o similar, que tenga un parámetro de entrada **Short actorId** y devuelva un objeto Actor o ActorVo
Para la primera prueba devolved un objeto fijo
Ej: return new ActorVo(“Paco”, “Martínez”)
- Anotad el método para GET, que coja el parámetro y que produzca JSON
- Probad con petición GET a la URL y luego probad con JQuery AJAX (montón de ejemplos en StackOverflow)


<http://stackoverflow.com/questions/29599968/calling-rest-webservice-using-jquery-ajax>

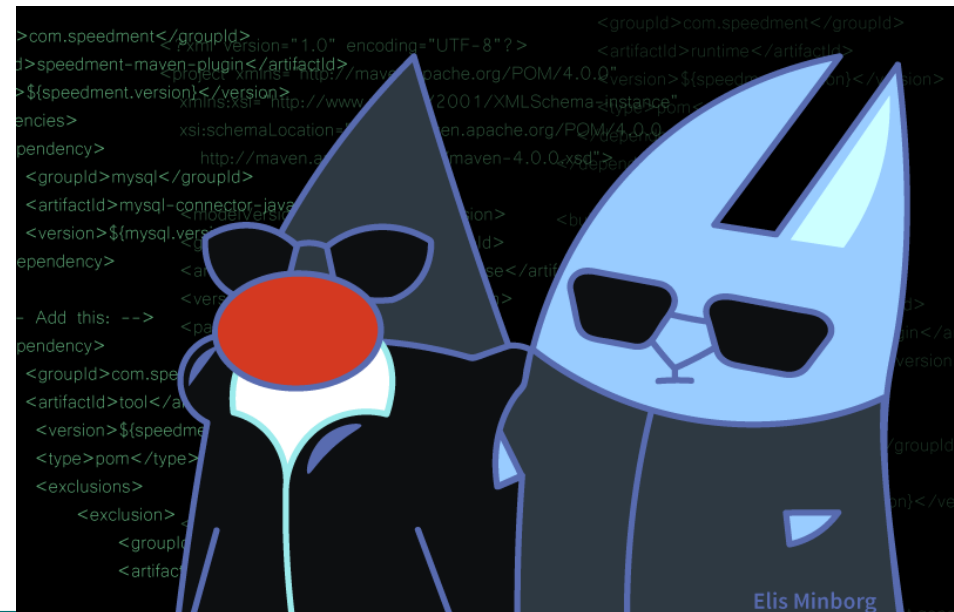


PREGUNTAS



Ejemplo en Netbeans

- <https://netbeans.org/kb/docs/websvc/rest.html?print=yes>
- Problema con Glassfish 4.1.1 (¿a quién hay que...  ?)
 - <http://entjavastuff.blogspot.com.es/2015/10/maven-javaee7-app-in-netbeans-8.html>
 - <https://java.net/jira/browse/GLASSFISH-21438>





Ejemplo con BD Sakila

- Cread una clase POJO para almacenar los datos del actor
En mi caso Actor o ActorVo (**vo** de value object-). Que sea JavaBean!!!
- Crear una clase POJO para el recurso REST
Ejemplo: ActorRest, en un packageservices.rest
- Anotar la clase con @Path (“actors”) o similar
(os pedirá que configuréis REST según Java EE 6)
- Cread un método **read** o similar, que tenga un parámetro de entrada **Short actorId** y devuelva un objeto Actor o ActorVo
Para la primera prueba devolved un objeto fijo
Ej: return new ActorVo(“Paco”, “Martínez”)
- Anotad el método para GET, que coja el parámetro y que produzca JSON
- Probad con petición GET a la URL y luego probad con JQuery AJAX (montón de ejemplos en StackOverflow)

<http://stackoverflow.com/questions/29599968/calling-rest-webservice-using-jquery-ajax>



PREGUNTAS

