Escuela Politécnica de Ingeniería
Grado de Ingeniería Informática en Tecnologías de la Información

Tecnologías Web

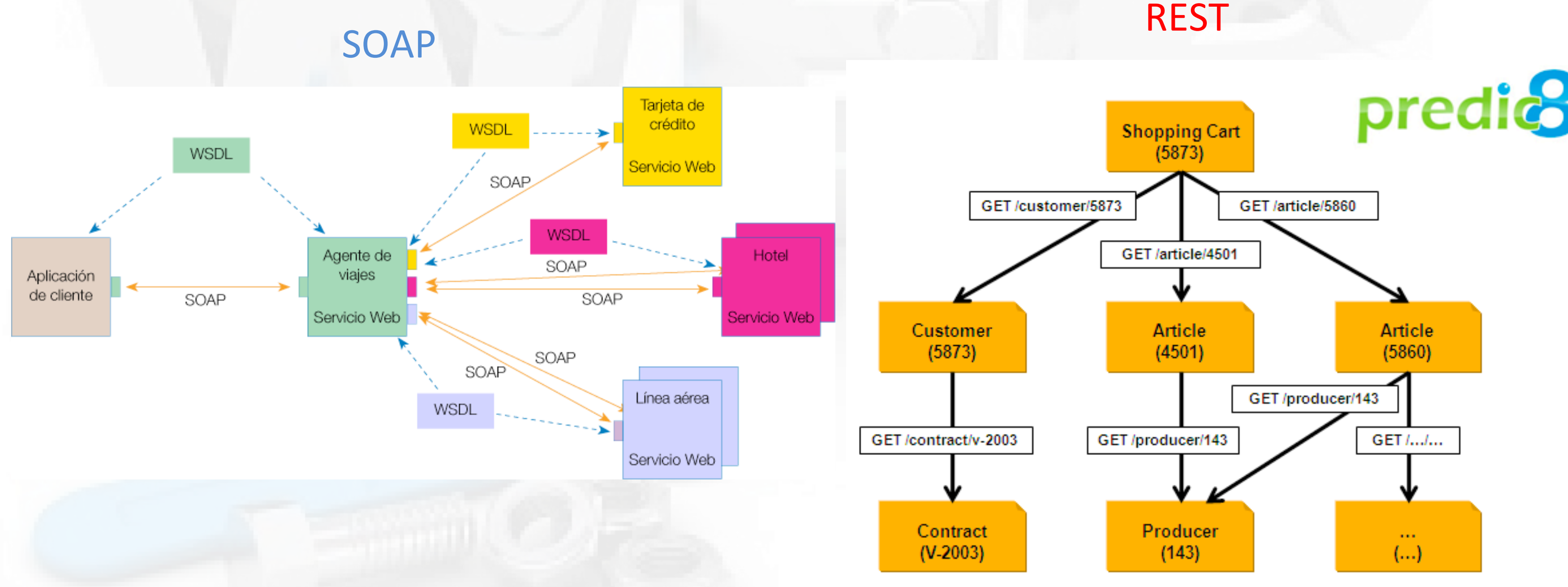
Tema 5
Servicios Web

Índice

– Introducción
– Protocolos y formatos
– APIs Java Servicios Web
– SOAP
– REST
– REST vs SOAP
– Diseño de un servicio REST con RestEasy

Introducción

- Definición de Servicio Web (W3C): "sistema de software diseñado para permitir la interacción interoperable entre máquinas en una red"



Introducción II

- Características de los servicios Web:
  - Independencia de plataforma (Hw + SO + Leng)
  - Uso de tecnologías Internet: XML, HTTP, etc
  - Interoperabilidad de programas: clientes y servidores escritos en cualquier lenguaje
- Invocación a servicios ofrecidos como procedimientos (SOAP) o como URIs (REST)

Servicios web como componentes remotos

- Mismo paradigma que otras soluciones anteriores:
  - CORBA: *Common Object Request Broker Architecture*
  - RMI: *Remote Method Invocation*
  - RPC: *Remote Procedure Call*
  - DCOM: *Distributed Component Object Model*
- Diferencia técnica: Se emplean tecnologías estándares abiertas Internet
  - Protocolo de transporte: TCP
  - Protocolo de Aplicación: HTTP
  - Protocolo de servicio web: REST y SOAP
  - Formato de datos: XML y JSON

Requisitos de interoperabilidad

- Intercambio de mensajes HTTP y XML
- Se necesita cerrar más las posibilidades
  - SOAP (*Simple Object Access Protocol*) → especificación XML para intercambio de invocaciones y respuestas basadas en RPC
  - **REST** (*Representational State Transfer*) -> especificación para intercambio de invocaciones y respuestas basado en métodos HTTP (GET, POST, PUT, DELETE y HEAD)
  - WSDL (*Web Services Description Language*) → descripción "semántica" de los servicios ofrecidos vía SOAP (el interfaz)
  - WADL (*Web Application Description Language*) → descripción "semántica" de los servicios ofrecidos vía REST (el interfaz)
  - **HTTP** (*Hypertext Transfer Protocol*) → protocolo de transporte para SOAP y REST (métodos y cabeceras estandarizados)
  - UDDI (*Universal Description, Discovery and Integration*) → norma de publicación y búsqueda de servicios ofrecidos por proveedor:
    - Páginas blancas - dirección, contacto y otros identificadores conocidos.
    - Páginas amarillas - categorización industrial basada en taxonomías.
    - Páginas verdes - información técnica sobre los servicios que aportan las propias empresas.

APIs java para SOAP/REST

- JDOM
- JAXP → Java API for XML procesing
- JAXB → Java API for XML binding
- JAX-RPC → Java API for RPC
- JAXR → Java API for UDDI registry
- SAAJ → SOAP API with Attachements
- JAX-WS → Java API for WebServices
  - API de más alto nivel
  - Usa todas las demás por debajo
  - Permite manejo de WS sin conocimientos de XML, SOAP, etc
- JAX-RS → Java API for RESTful Web Services

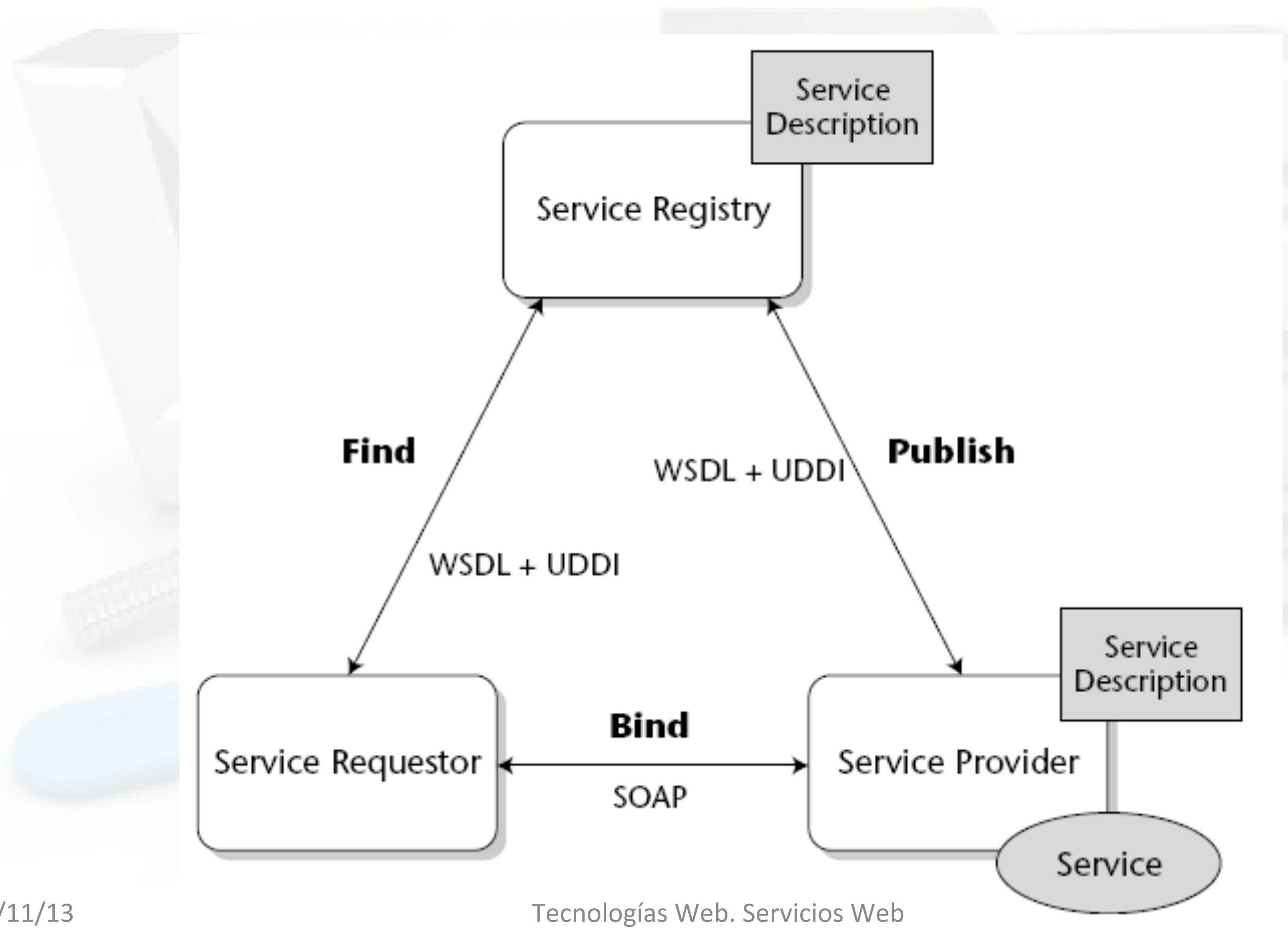
-

+

Complejidad

SOAP

Arquitectura de WS usando SOAP



Ejemplo mensajes SOAP

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:ns1="http://sample/">
    <soapenv:Body>
        <ns1:getGreeting>
            <arg0>Duke</arg0>
        </ns1:getGreeting>
    </soapenv:Body>
</soapenv:Envelope>
```

Web Service

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:ns1="http://sample/">
    <soapenv:Body>
        <ns1:getGreetingResponse>
            <return>Hola Duke</return>
        </ns1:getGreetingResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

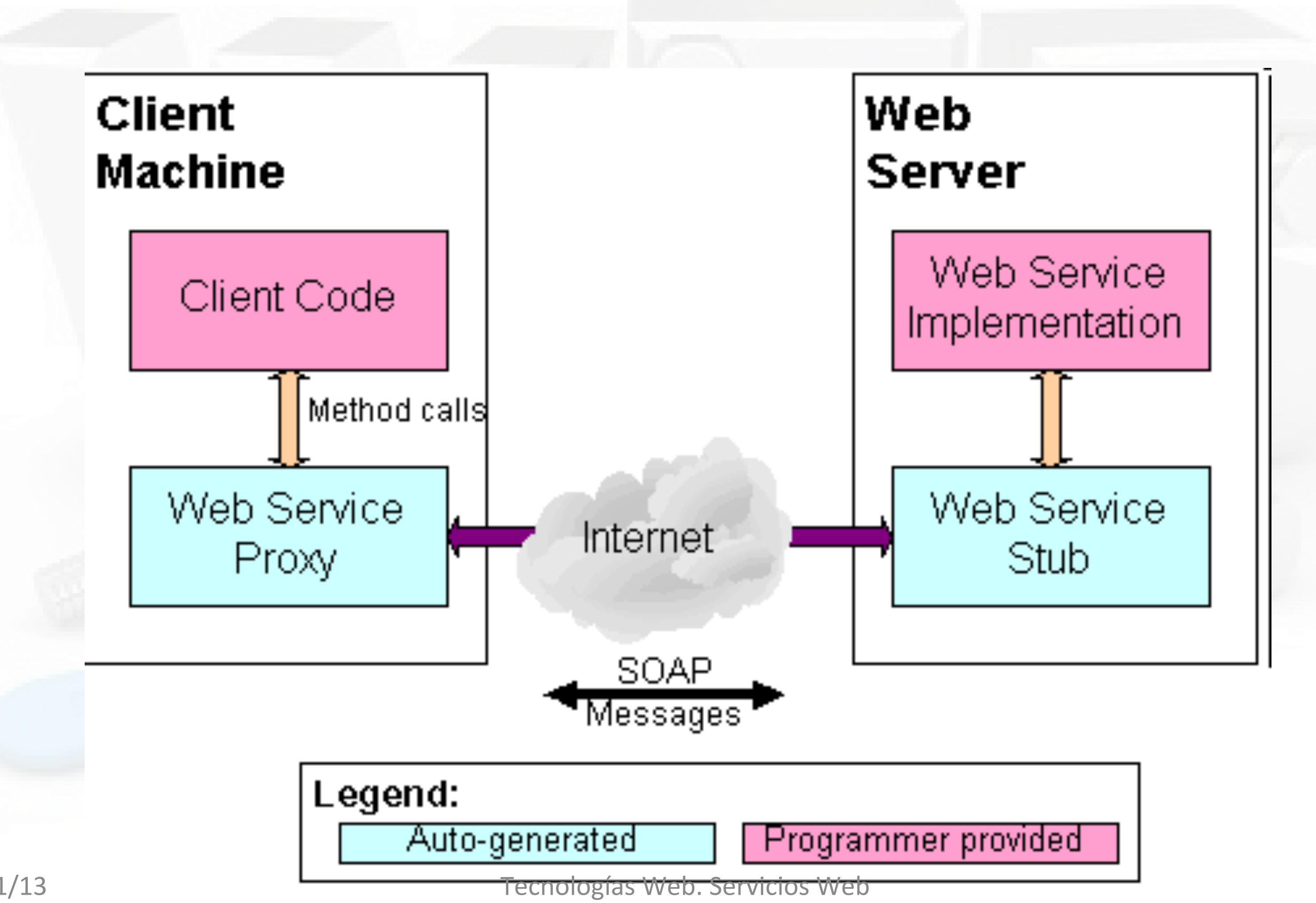
Ejemplo WSDL “hola mundo”

```
<definitions>
    <types>
        los tipos de datos...
    </types>
    <message>
        las definiciones del mensaje...
    </message>
    <portType>
        las definiciones de operación ...
    </portType>
    <binding>
        las definiciones de protocolo...
    </binding>
</definitions>
```

Estructura general de documento WSDL

```
<message name="hello">
    <part name="param" type="xs:string"/>
</message>
<message name="helloResponde">
    <part name="valor" type="xs:string"/>
</message>
<portType name="HelloBean">
    <operation name="sayHello">
        <input message="hello"/>
        <output message="helloResponse"/>
    </operation>
</portType>
<binding type="GreeterPortBinding" name="helleBean">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http" />
    <operation name="sayHello">
        <soap:operation soapAction="http://uoc.edu/obtTermino"/>
        <input>
            <soap:body use="literal"/>
        </input>
        <output>
            <soap:body use="literal"/>
        </output>
    </operation>
</binding>
```

Invocación de WS a través de proxy



Web Services for the Real World
RESTful
Web Services
O'REILLY

Arquitectura REST (Representational State Transfer)

- Def. técnica de arquitectura software para sistemas hipermedia distribuidos como la World Wide Web.
- Tesis de Roy Fielding en el 2000 (autor de HTTP 1.x)
- Principios de REST:
  - Un **protocolo** cliente/servidor **sin estado** (REST No admiten Reescritura de URLs)**.**
  - Un **conjunto de operaciones** bien definidas. HTTP admite un buen conjunto de operaciones: GET, POST, PUT y DELETE (CRUD)
  - Una **sintaxis universal** para identificar los recursos (URI)
  - El **uso de hipermedios**. No es necesario el uso de registros para navegar de un recurso REST a otro.
- **HATEOAS**: Principio consistente en que la interacción entre un cliente RES y los servicios se realiza de forma dinámica sin necesidad de IDL

Arquitectura REST 2

- REST amplia su definición a cualquier interfaz de servicios web que usa XML y HTTP sin las abstracciones RPC de SOAP
- RPC no es REST
- Los sistemas que siguen los principios REST se denominan con frecuencia *RESTful*

Un WS-RESTful en Java

- Librería: Especificación JAX-RS 2.0
- Usa anotaciones para desarrolar WSCLientes y endpoints
- Desde la versión 1.1 forma parte de JEE 6
- Anotaciones JAX-RS
  - @Path especifica el path relativo para un clase o método de recurso
  - @GET, @PUT, @POST, @DELETE and @HEAD especifica el tipo de solicitud HTTP
  - @Produces especifica los tipos MIME que produce un método REST.
  - @Consumes especifica los tipos MIME que puede recibir un método REST
  - Anotaciones adicionales:
    - @PathParam , @QueryParam, @MatrixParam, ...
- Implementaciones:
  - Jersey es la RI de Sun/Oracle
  - **RESTeasy es la implementación de Jboss.**

REST y HTTP Methods

- HTTP Methods “metáfora CRUD”.
- Recomendaciones:

CRUD	SQL	HTTP
Create	Insert	PUT
Read	Select	GET
Update	Update	POST
Delete	Delete	DELETE

```
@Path("/helloworld")
public interface HelloWorldResource {
@GET
@Path("/saluda")
public String getSaluda();
@GET
@Path("/saluda/{nombre}")
public String getSaludaA(@PathParam("nombre")
String nombre); }
```

```
public class HelloWorldResourceImpl implements
HelloWorldResource {

@Override
public String getSaluda() {
  return "¡Hola mundo!";
}
@Override
public String getSaludaA(String nombre) {
 return MessageFormat.format("¡Hola {0}!",
nombre);}
```

REST vs. SOAP (RPC)

- El enfoque de servicios en SOAP es procedimental basado en muchos servicios/verbos:
  - getUser()
  - addUser()
  - removeUser()
  - updateUser()
  - getLocation()
  - addLocation()
  - removeLocation()
  - updateLocation(

- En cambio REST sigue un esquema basado recursos o nombres, p.e.:
  - Usuario {}
  - Localización {}
  - Cada recurso tiene su propio URI:
    - http://www.example.org/locations/us/ny/new_york_city

REST vs. SOAP 2

- **REST**: utiliza HTTP como trasporte y XML o JSON para intercambio. Cada URL representa un objeto sobre el que se pueden utilizar los métodos POST, GET, PUT y DELETE. Utiliza el idioma de la web…
- **SOAP**: es toda una infraestructura basada en XML, cada objeto puede tener métodos definidos por el programador con los parámetros que sean necesarios.
- Conclusión:
  - REST: Más ligero, poca configuración, legible (URLs)
  - SOAP: Más ambicioso, tipado fuerte (WSDL). Está cayendo hacia el lado oscuro (Empresa).
- Ejemplo:

```
En SOAP:
bank = new SOAPProxy("http://…..")
bank.addMoneyToAccount(account 123456789, 50 euro)
bank = new SOAPProxy("http://…..")
bank.getAccountBalance(account 123456789)
En REST:
http://…/bank/addMoneyToAccount?account=123456789&money=50
http://…/bank/getAccountBalance?account=123456789
```

Cómo diseñar un servicio REST con RESTeasy

1. Definición de la fachada de recursos (interface HelloWorldResource)
2. Anotar los tipos estructurados (clase Mensaje)
3. Implementación de los recursos (Clase HelloWorldResourceImpl)
4. Registro estático de las clases de recursos (Clase Application)
5. Descriptor de despliegue (web.xml)
6. Diseño del cliente

Paso 1: Fachada de recursos

- @Path: indica parte de la URL en la que el web service responderá,
- @GET: con la segunda indicamos que el método HTTP que llame a esa URL deberá ser get y
- @PathParam. Permite recoger un parámetro indicado en la URL

```java
package ejemplo.resteasy;

import javax.ws.rs.*GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/helloworld")
public interface HelloWorldResource

    @GET
    @Path("/saluda")
    public String getSaluda();

    @GET
    @Path("/saluda/{nombre}")
    public String getSaludaA(@PathParam("nombre") String nombre);

    @GET
    @Path("/mensaje/{nombre}")
    @Produces(MediaType.APPLICATION_JSON)
    public Mensaje getMensajeJSON(@PathParam("nombre") String nombre);
```

```java
package ejemplo.resteasy;

import java.text.MessageFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
public class HelloWorldResourceImpl implements HelloWorldResource {
    @Override
    public String getSaluda() {
        return "¡Hola mundo!";
    }
    @Override
    public String getSaludaA(String nombre) {
        return MessageFormat.format("¡Hola {0}!", nombre);
    }

    @Override
    public Mensaje getMensajeJSON(String nombre) {
        return buildMensaje(nombre);
    }

    @Override
    public Mensaje getMensajeXML(String nombre) {
        return buildMensaje(nombre);
    }
```

XML vs JSON (JavaScript Object Notation)

- Ejemplo JSON

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
 }
}
```

- XML mayor soporte y herramientas (cliente y servidor).
- JSON, en Javascript se parsea con eval()
- JSON en general más compacto y eficiente que XML
- XML mejor en datos superestructuras
- YAML un superconjunto de JSON más complejo
- En Ruby se usa YAML para serialización

Fachada JSON/XML

- Anotar los métodos con @Produces indicando el formato de los datos que se devuelve
  - en la clase [MediaType hay constantes para muchos formatos.]()
  - Tipo específico con @Produces("text/html"):

```
package ejemplo.resteasy;

import javax.ws.rs.*GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/helloworld")
public interface HelloWorldResource {

    @GET
    @Path("/saluda")
    public String getSaluda();

    @GET
    @Path("/saluda/{nombre}")
    public String getSaludaA(@PathParam("nombre") String nombre);

    @GET
    @Path("/mensaje/{nombre}")
    @Produces(MediaType.APPLICATION_JSON)
    public Mensaje getMensajeJSON(@PathParam("nombre") String nombre);

    @GET
    @Path("/mensaje/{nombre}")
    @Produces(MediaType.APPLICATION_XML)
    public Mensaje getMensajeXML(@PathParam("nombre") String nombre);
}
```

Paso 2: Anotar los tipos estructuados

```
package ejemplo.resteasy;

import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "mensaje")
public class Mensaje {
    private String nombre;  private String mensaje; private String fecha;

    public Mensaje() {
    }
    public Mensaje(String nombre, String mensaje, String fecha) {
        this.nombre = nombre;
        this.mensaje = mensaje;
        this.fecha = fecha;
    }
    @XmlElement
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    @XmlElement
    public String getMensaje() {
        return mensaje;
    }
    public void setMensaje(String mensaje) {
        this.mensaje = mensaje;
    }
    @XmlAttribute
    public String getFecha() {
        return fecha;
    }
    public void setFecha(String fecha) {
        this.fecha = fecha;
    }
}
```

Tipos
- básicos: No es necesario anotarlos
- estructurados: Si es necesario anotarlos

Paso 3: Implementación del Servicio(End Point) con JSON/XML

```
1  package ejempld.resteasy;
2
3  import java.text.MessageFormat;
4  import java.text.SimpleDateFormat;
5  import java.util.Date;
6  public class HelloWorldResourceImpl implements HelloWorldResource {
7      @Override
8      public String getSaluda() {
9          return "¡Hola mundo!";
10     }
11     @Override
12     public String getSaludaA(String nombre) {
13         return MessageFormat.format("¡Hola {0}!", nombre);
14     }
15     @Override
16     public Mensaje getMensajeJSON(String nombre) {
17         return buildMensaje(nombre);
18     }
19     @Override
20     public Mensaje getMensajeXML(String nombre) {
21         return buildMensaje(nombre);
22     }
23     private Mensaje buildMensaje(String nombre) {
24         return new Mensaje(nombre, "¡Hola mundo!",
25         new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new Date()));
26     }
27 }
```

Paso 4: Clase Application

```java
package ejemplo.resteasy;

import java.util.Collections;
import java.util.HashSet;
import java.util.Set;

@SuppressWarnings("unchecked")
public class Application extends javax.ws.rs.core.Application {

    private Set<Class<?>> classes = new HashSet<Class<?>>();

    public Application() {
        classes.add(HelloWorldResourceImpl.class);
    }

    @Override
    public Set<Class<?>> getClasses() {
        return classes;
    }

    @Override
    public Set<Object> getSingletons() {
        return Collections.EMPTY_SET;
    }
}
```

No es necesario uso de WDSL/WADSL (contra HATEOAS)

Paso 5: Descriptor de despliegue

- Añadir es el servlet de RESTEasy y el correspondiente servlet-mapping.

```
<servlet>
    <servlet-name>resteasy</servlet-name>
    <servlet-class>org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-class>
    <init-param>
        <param-name>javax.ws.rs.Application</param-name>
        <param-value>ejemplo.resteasy.Application</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

```
<servlet-mapping>
    <servlet-name>resteasy</servlet-name>
    <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

- Indicar el parámetro de contexto “resteasy.servlet.mapping.prefix” con el

```
<context-param>
    <param-name>resteasy.servlet.mapping.prefix</param-name>
    <param-value>/rest</param-value>
</context-param>
```

- Clase que extiende de [javax.ws.rs.core.Application y utilizando el parámetro de inicialización javax.ws.rs.Application]()

Paso 6: Diseño de clientes (Cliente Java)

```java
package ejemplo.resteasy;
import org.jboss.resteasy.client.ProxyFactory;
import org.jboss.resteasy.plugins.providers.RegisterBuiltin;
import org.jboss.resteasy.spi.ResteasyProviderFactory;

public class HelloWorldResourceClient implements HelloWorldResource {
    private HelloWorldResource client;
    public HelloWorldResourceClient() {
        // Obtener el cliente a partir de la interfaz y de donde está localizado
        client = ProxyFactory.create(HelloWorldResource.class, "http://localhost:8080/helloworld-resteasy/rest");
    }
    @Override
    public String getSaluda() {
        return client.getSaluda();
    }
    @Override
    public String getSaludaA(String nombre) {
        return client.getSaludaA(nombre);
    }
    @Override
    public Mensaje getMensajeJSON(String nombre) {
        return client.getMensajeJSON(nombre);
    }
    @Override
    public Mensaje getMensajeXML(String nombre) {
        return client.getMensajeXML(nombre);
    }
    public static void main(String[] args) {
        // Inicializacion a realizar una vez por máquina virtual
        RegisterBuiltin.register(ResteasyProviderFactory.getInstance());

        HelloWorldResourceClient client = new HelloWorldResourceClient();
        System.out.println(client.getSaluda());
        System.out.println(client.getSaludaA("Soy Enrique"));
        System.out.println(client.getMensajeJSON("Soy Enrique"));
        System.out.println(client.getMensajeXML("Soy Enrique"));
    }
}
```

- Clases clave RegisterBuiltin y ProxyFactory

Cliente Javascript

- RESTEasy proporciona un servlet que genera un archivo js con los clientes javascript de nuestros servicios web REST, lo único que deberemos hacer es incluir ese servlet en el web.xml de nuestra aplicación web y usarlo en la página html en la que queramos hacer uso de él.
- Debe cargarse después del servlet resteasy (load-on-startup 2)

```
<servlet>
    <servlet-name>resteasy-jsapi</servlet-name>
    <servlet-class>org.jboss.resteasy.jsapi.JSAPIServlet</servlet-class>
    <load-on-startup>2</load-on-startup>
</servlet>
```

```
<servlet-mapping>
    <servlet-name>resteasy-jsapi</servlet-name>
    <url-pattern>/rest-jsapi</url-pattern>
</servlet-mapping>
```

CLIENTE Javascript

```
1  <html>
2  <head>
3      <title>Ejemplo sencillo de web service con RESTEasy</title>
4      <script type="text/javascript" src="http://localhost:8080/helloworld-resteasy/rest-jsapi"></script>
5  </head>
6  <body>
7      <script type="text/javascript">
8          alert(HelloWorldResource.getSaluda());
9          alert(HelloWorldResource.getSaludaA({nombre:'Soy Enrique'}));
10         alert(HelloWorldResource.getMensajeJSON({nombre:'Soy Enrique'}));
11         alert(HelloWorldResource.getMensajeXML({nombre:'Soy Enrique'}));
12     </script>
13 </body>
14 </html>
```