

# 6. Arquitecturas Orientadas a Servicios y Servicios Web

Copyright (c) 2025 Adrián Quiroga Linares Lectura y referencia permitidas; reutilización y plagio prohibidos

## 6.1 El Problema: Comunicación entre Programas Distantes

Imagina que tienes una clase Java `SaySomething` en tu portátil y quieres usar un método `helloBuddy` de una clase `Hello` que está en un servidor en la otra punta del mundo .

### 6.1.1 La Evolución Fallida

Se intentaron varias soluciones (RPC, CORBA, DCOM), pero fracasaron en la web abierta por dos razones:

1. **Bloqueo:** Los **Firewalls** bloqueaban sus comunicaciones complejas.
2. **Incompatibilidad:** Requerían que ambas máquinas usaran la misma tecnología (ej. Java con Java).

#### Info

El **firewall** es el guardia de seguridad de la red de una empresa o un ordenador.

- **Funcionamiento:** Controla las "puertas" (puertos) de entrada y salida. Imagina un edificio con 65.000 puertas. El Firewall las cierra todas por seguridad, excepto las imprescindibles.
- **El problema histórico:** Antiguamente, las tecnologías distribuidas usaban puertos raros y aleatorios. Los Firewalls las bloqueaban sistemáticamente, impidiendo la comunicación entre empresas.
- **La solución Web:** Los Firewalls casi siempre dejan abierta la **puerta 80 (HTTP/Web)**, porque todo el mundo necesita navegar por internet. SOA aprovecha esto para "colar" sus datos por esa puerta abierta.

### 6.1.2 El Paso Intermedio: Java Servlets

Antes de llegar a SOA, Java intentó solucionar la comunicación web con los **Servlets**. Es una clase Java que reside en el servidor y es capaz de interceptar peticiones **HTTP** (puerto 80) y devolver una respuesta.

- **La Ventaja frente a RMI:** Al usar HTTP, **atraviesa los Firewalls**. Ya no hay bloqueos de puertos raros.
- **El Problema que tienen:** Los Servlets están diseñados originalmente para devolver **HTML** (páginas web para humanos), no datos estructurados para otros programas.

**ⓘ El problema del "Espaguetti" en Servlets** Si usas un Servlet "a pelo" para conectar dos aplicaciones, tienes que procesar el texto manualmente.

1. **Cliente:** Envía `POST /sumar?a=5&b=5`.
2. **Servlet:** Recibe el texto, lo parsea, suma y devuelve `10`.

**¿Por qué esto no es SOA?** Porque no hay un contrato estándar (WSDL). Si cambias el código del Servlet, rompes al cliente. No hay validación de tipos automática. Es una comunicación "artesanal".

**SOA y los Servicios Web** nacen para poner orden aquí: usan la tecnología base de los Servlets (HTTP) pero añaden reglas estrictas (XML/SOAP) para que las máquinas se entiendan sin errores.

### 6.1.3 La Solución: Servicios Web

La idea genial fue: "*¿Y si empaquetamos nuestros datos en XML (que todos entienden) y los enviamos usando HTTP (que atraviesa todos los Firewalls)?*".

De aquí nace la **Arquitectura Orientada a Servicios (SOA)**.

**ⓘ Info**

**XML:** Imagina que escribes una carta a un amigo, pero quieres que un ordenador entienda perfectamente qué parte es la fecha, cuál es el saludo y cuál el contenido.

- **Definición:** Es un estándar para estructurar datos en formato de texto. A diferencia de HTML (que le dice al navegador *cómo mostrar* algo, ej: "pon esto en negrita"), **XML le dice al ordenador qué es ese dato**.
- **Funcionamiento:** Usa "etiquetas" (tags) que abren y cierran.
  - Ejemplo: `<precio>50</precio>`. Aquí el ordenador sabe que "50" es un precio.
- **Por qué es vital en SOA:** Porque es **independiente del lenguaje**. Java, Python o C++ pueden leer texto plano. Si enviamos datos en formato XML, todos se entienden .

**HTTP** es el protocolo (el idioma) de la World Wide Web.

- **Funcionamiento:** Funciona con un modelo de **Petición-Respuesta**.

1. Tu navegador (Cliente) envía una petición (Request) a un servidor: "*Dame la página <https://www.google.com/search?q=google.com>*".
  2. El servidor responde (Response): "*Aquí tienes el código de la página*".
- **Características:** Es un protocolo de texto y **sin estado** (stateless), lo que significa que cada petición es independiente de la anterior.
  - **Verbos:** Usa acciones como **GET** (pedir info) o **POST** (enviar info). En Servicios Web, usaremos mucho **POST** para enviar mensajes de datos al servidor.

## 6.2 Arquitectura Orientada a Servicios (SOA)

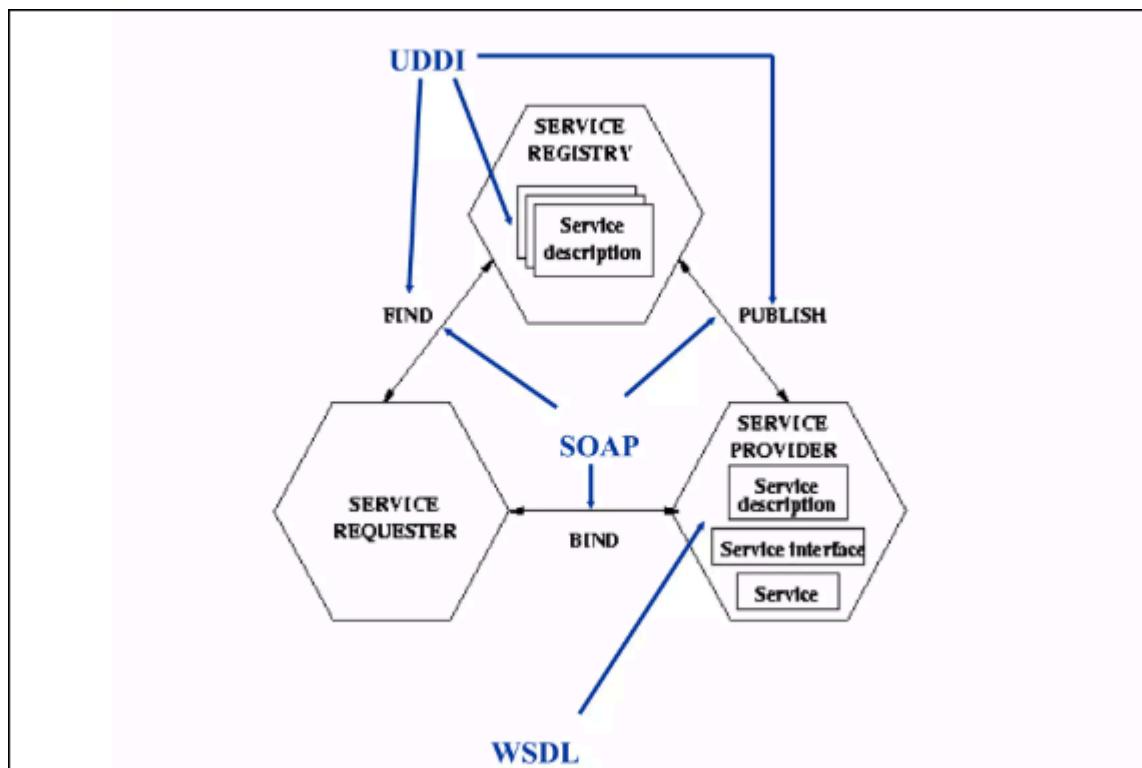
SOA no es un software, es una forma de organizar tus sistemas. Se basa en construir aplicaciones conectando piezas independientes llamadas **Servicios**.

Para que esto funcione, necesitamos tres actores :

1. **Proveedor (Service Provider):** Crea el servicio y lo mantiene.
2. **Registro (Service Registry):** Un directorio (como unas Páginas Amarillas) donde se listan los servicios disponibles.
3. **Consumidor (Service Requester):** El cliente que necesita usar el servicio.

### El Flujo de Trabajo:

1. **PUBLISH (Publicar):** El Proveedor sube la descripción de su servicio al Registro.
2. **FIND (Buscar):** El Consumidor busca en el Registro: "¿Quién tiene un servicio de calculadora?".
3. **BIND (Ligar/Conectar):** El Consumidor obtiene la dirección y conecta directamente con el Proveedor para usarlo.



## 6.3 Implementación de SOA mediante Servicios Web

Aunque SOA es el concepto, los **Servicios Web** son la implementación técnica más común. Utilizan estándares basados en **XML** para garantizar que cualquier lenguaje (Java, Python, C#) pueda hablar con cualquier otro .

El "Stack" tecnológico estándar del W3C incluye :

1. **Transporte:** HTTP (generalmente).
2. **Mensajería (Formato):** **SOAP** (Simple Object Access Protocol).
3. **Descripción (Contrato):** **WSDL** (Web Service Description Language).
4. **Descubrimiento:** **UDDI** (Universal Description, Discovery and Integration) — *Nota: Este último ha caído en desuso, pero es parte de la teoría clásica.*

## 6.4 El Manual de Instrucciones: WSDL

Si vas a conectar tu código con un servicio remoto, necesitas saber exactamente cómo hablarle. Para eso existe el **WSDL** (Web Service Description Language).

Es un documento **XML** que actúa como un "contrato". Describe dos cosas fundamentales :

### 6.4.1 Descripción Abstracta (El "Qué")

Define la funcionalidad sin importar el lenguaje de programación.

- **Types (Tipos):** Define los datos usando **XML Schema**. Aquí definimos qué es un "Usuario" o una "Factura".
  - *Traducción:* Si en Java tienes un **int**, en WSDL se define como **xsd:int**.
- **Messages (Mensajes):** Define qué datos entran y cuáles salen.
- **Port Type (Interfaz):** Agrupa las operaciones (funciones) disponibles .

### 6.4.2 Descripción Concreta (El "Cómo")

- **Binding:** Dice qué protocolo usar. Casi siempre dirá: "Usa **SOAP** sobre **HTTP**".
- **Service:** Dice la dirección web (URL) donde está el servidor escuchando.

**Proceso Mágico:** Gracias a que el WSDL es XML estandarizado, herramientas automáticas (como JAX-WS) pueden leerlo y generar código Java (clases) automáticamente por ti .

### 6.4.3 Ejemplo

Ponte que queremos exponer la siguiente función

JAVA

```
// Queremos convertir esto a WSDL
public int sumar(int numeroA, int numeroB);
```

Primero definiríamos los **types** (el diccionario de datos):

XML

```
<types>
  <xsd:schema>
    <xsd:element name="SumarRequest">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="numeroA" type="xsd:int"/>
          <xsd:element name="numeroB" type="xsd:int"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="SumarResponse">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="resultado" type="xsd:int"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</types>
```

Posteriormente los **message**, donde definimos qué es una petición y qué es una respuesta

XML

```
<message name="PeticionSuma">
  <part name="parameters" element="tns:SumarRequest"/>
</message>

<message name="RespuestaSuma">
  <part name="parameters" element="tns:SumarResponse"/>
</message>
```

Seguimos con el **portType**, define las **operaciones** (métodos). Conecta los mensajes de entrada y salida con un nombre de función.

XML

```
<portType name="CalculadoraInterface">
  <operation name="sumar">
    <input message="tns:PeticionSuma"/>
    <output message="tns:RespuestaSuma"/>
  </operation>
</portType>
```

En el **binding** se define que esos datos van a viajar usando el protocolo **SOAP** y sobre **HTTP**.

XML

```
<binding name="CalculadoraSoapBinding" type="tns:CalculadoraInterface">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>

  <operation name="sumar">
    <soap:operation soapAction="sumar"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
```

Y por último en **service**. El final del documento. Te dice la URL exacta donde vive el servidor.

XML

```
<service name="CalculadoraService">
  <port name="CalculadoraPort" binding="tns:CalculadoraSoapBinding">
    <soap:address location="http://localhost:8080/mi-app/calculadora"/>
  </port>
</service>
```

Obviamente esto último no lo hay que saber ni de coña, es solo para entender un poco de qué va, porque a la hora de programar en java este código se genera automáticamente.

### Info

**WSDL** Es el **Menú y el Formulario de Pedido**. No es la comida, ni es el camarero. Es un documento (un papel) que te dice:

- **Qué ofrecen:** "Tenemos Hamburguesas y Ensaladas".
- **Cómo pedirlo:** "Para la hamburguesa DEBES decirnos: tipo de carne (texto) y si quieres queso (booleano)".
- **Dónde:** "Estamos en la Calle Falsa 123".

**En resumen:** El WSDL es el **contrato**. Antes de escribir una sola línea de código, el cliente lee el WSDL para saber qué métodos existen y qué parámetros necesitan.

## 6.5 El Mensaje: SOAP (Simple Object Access Protocol)

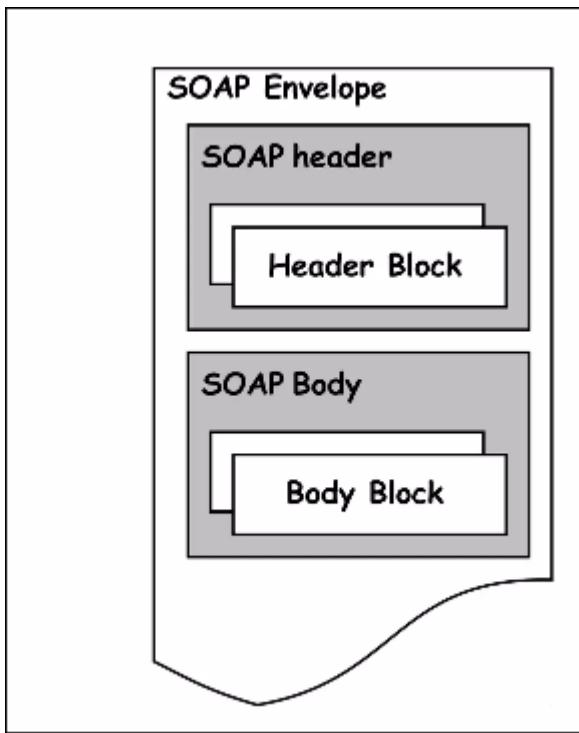
Ya tenemos el manual (WSDL) y el canal (HTTP). Ahora necesitamos el sobre para enviar la carta. Eso es SOAP.

SOAP es un protocolo basado estrictamente en XML para intercambiar información estructurada.

### 6.5.1 Estructura de un mensaje SOAP

Visualiza una carta real. Un mensaje SOAP tiene 3 partes jerárquicas:

1. **Envelope (Sobre):** Es la etiqueta raíz. Envuelve todo el mensaje. Sin esta etiqueta, el XML no se considera un mensaje SOAP válido.
2. **Header (Cabezera) - Opcional:** Es la parte **logística**. Contiene metadatos para la infraestructura, no para la aplicación final.
  - Aquí viajan cosas como: Tokens de seguridad, IDs de transacción o firmas digitales.
  - **Importante:** Puede ser procesado por **intermediarios** (nodos de red, firewalls, balanceadores) antes de llegar al destino final.
3. **Body (Cuerpo) - Obligatorio:** Es la parte del **negocio**. Contiene la información real que el usuario quiere enviar o recibir (ej: los números a sumar).



## 6.5.2 Ejemplo Visual

### Nota

Este código XML es solo para visualizar la estructura "Sobre dentro de Sobre". No es necesario memorizar las etiquetas exactas.

### Mensaje de Petición (Request):

```

<soapenv:Envelope xmlns:soapenv="...">
    <soapenv:Header>
        <miweb:AuthToken>SECRET_12345</miweb:AuthToken>
    </soapenv:Header>

    <soapenv:Body>
        <miweb:sumar>
            <numeroA>10</numeroA>
            <numeroB>5</numeroB>
        </miweb:sumar>
    </soapenv:Body>
</soapenv:Envelope>

```

XML

### Mensaje de Respuesta (Response):

```

<soapenv:Envelope xmlns:soapenv="...">
  <soapenv:Header/> <soapenv:Body>
    <miweb:sumarResponse>
      <resultado>15</resultado>
    </miweb:sumarResponse>
  </soapenv:Body>
</soapenv:Envelope>

```

## 6.5.3 Conceptos Clave del Header (**role** y **mustUnderstand**)

El mensaje SOAP no siempre viaja directo del Cliente al Servidor. A veces pasa por **Intermediarios** (nodos entre medias). Para gestionar esto, el Header tiene dos atributos vitales:

### A. El atributo **role** (¿Para quién es la nota?)

Define **quién** es el destinatario de ese bloque específico de la cabecera.

- Imagina un paquete que pasa por un Guardia de Seguridad antes de llegar al Director General.
- Podemos poner una nota en el Header que diga: "*role=GuardiaSeguridad*".
- El Guardia leerá esa nota, pero el Director General la ignorará porque no va dirigida a él.
- **Valor por defecto:** Si no se pone nada, se asume que es para el destinatario final (*role="ultimateReceiver"*).

### B. El atributo **mustUnderstand** (¿Es obligatorio leerlo?)

Es un interruptor de seguridad (**true** o **false**). Define si el receptor puede ignorar la cabecera si no sabe qué significa.

- **mustUnderstand="false" (Por defecto):** "Información opcional".
  - **Ejemplo:** Una etiqueta de "Prioridad Baja". Si el servidor receptor entiende de prioridades, genial. Si es un servidor viejo que no sabe qué es eso, simplemente ignora la etiqueta y procesa el mensaje igual.
- **mustUnderstand="true" (Obligatorio):** "Información Crítica".
  - **Ejemplo:** Un "Token de Seguridad". Le dices al servidor: "*O procesas esta seguridad o NO toques el mensaje*".
  - **Regla de Oro:** Si un servidor recibe un header con **mustUnderstand="true"** y **no entiende** esa etiqueta XML, **debe rechazar** el mensaje inmediatamente y devolver un Fallo (SOAP Fault). No puede arriesgarse a procesarlo mal.

#### Analogía de la Maleta

Imagina que envías una maleta (SOAP Envelope) en un avión.

1. **La Ropa (Body):** Es lo que te importa, tus datos.
2. **La Etiqueta de Facturación (Header):** Es logística.
  - **role:** Hay una pegatina pequeña para el operario de rayos X (*Intermediario*). Esa pegatina no es para ti (Destino final), es para él.
  - **mustUnderstand:**
    - Si la etiqueta dice "Fragile" (**mustUnderstand=false**) y el operario no sabe inglés, quizás la ignora y tira la maleta. La maleta llega, pero quizás golpeada. (El proceso continúa).
    - Si la etiqueta dice "PELIGRO BIOLÓGICO" (**mustUnderstand=true**) y el operario no sabe qué significa, **detiene todo el proceso** y devuelve la maleta. No puede dejarla pasar si no entiende el riesgo.

### Resumen Conceptual

SOAP es la Caja de envío certificada.

- **No hay aliasing (Referencias):** La comunicación es por **copia y valor (Pass-by-Value)**.
  1. Tus objetos Java se serializan a texto (XML).
  2. El texto viaja.
  3. El receptor lee el texto y crea **nuevos** objetos Java (copias).

UDDI (Páginas Amarillas):

(No entra en detalle, solo concepto). Es el directorio donde:

- **Páginas Blancas:** Quién es la empresa.
- **Páginas Amarillas:** Qué servicio ofrece.
- **Páginas Verdes:** Dónde está el WSDL (la URL técnica).

## 6.6 Flujo

Supongamos que quieres sumar **5 + 5**.

1. **Tu Código (Java):** Escribe `servicio.sumar(5, 5)`. Para ti es un método normal.
2. **Stub (El Traductor):** Este es un código falso generado automáticamente. Tú crees que estás llamando al servidor, pero el Stub intercepta la llamada.
  - **Acción:** El Stub toma el **5** y el **5** y empieza a escribir XML.
3. **SOAP Engine:** Mete ese XML en un sobre SOAP.
  - **Resultado:**

XML

```

<soap:Envelope>
  <soap:Body>
    <sumar>
      <arg0>5</arg0>
      <arg1>5</arg1>
    </sumar>
  </soap:Body>
</soap:Envelope>

```

4. **Transporte:** Ese tocho de texto viaja por HTTP (Internet).

5. **Servidor (Skeleton/Tie):** Recibe el texto XML. Hace el proceso inverso (**Unmarshalling**).

- Lee `<arg0>5</arg0>` → Lo convierte a `int a = 5`.
- Lee `<arg1>5</arg1>` → Lo convierte a `int b = 5`.

6. **Ejecución:** El servidor ejecuta `return 5 + 5`.

7. **Respuesta:** El servidor convierte el `10` resultante en XML (`<return>10</return>`) y lo envía de vuelta.

## 6.7 Ejemplo

### El Servidor (Service Provider)

1. **La Interfaz (Calculadora.java)** Cambio clave: `javax.jws` → `jakarta.jws`

JAVA

```

package com.ejemplo;

import jakarta.jws.WebService;
import jakarta.jws.WebMethod;

@WebService
public interface Calculadora {
    @WebMethod
    int sumar(int a, int b);
}

```

2. **La Implementación (CalculadoraImpl.java)**

```

package com.ejemplo;

import jakarta.jws.WebService;

@WebService(endpointInterface = "com.ejemplo.Calculadora")
public class CalculadoraImpl implements Calculadora {

    @Override
    public int sumar(int a, int b) {
        System.out.println(">> Servidor: He recibido una petición para sumar "
                + a + " + " + b);
        return a + b;
    }
}

```

### 3. El Publicador (Publicador.java)

```

package com.ejemplo;

import jakarta.xml.ws.Endpoint;

public class Publicador {
    public static void main(String[] args) {
        // Publicamos el servicio en localhost
        String url = "http://localhost:8080/miCalculadora";

        System.out.println("Iniciando servidor...");
        Endpoint.publish(url, new CalculadoraImpl());

        System.out.println("Servicio publicado exitosamente.");
        System.out.println("WSDL disponible en: " + url + "?wsdl");
    }
}

```

`Endpoint.publish(url, implementor)`. Esta función es la que "enciende" el servidor.

- **Argumentos:**

1. **url (String):** La dirección web donde quieras ofrecer el servicio.
  - *Ejemplo:* `"http://localhost:8080/miCalculadora"`
  - Define tres cosas: Protocolo (http), Máquina y Puerto (localhost:8080) y el nombre del servicio (/miCalculadora).

2. **implementor (Object)**: Una instancia real de la clase que hace el trabajo.

- *Ejemplo:* `new CalculadoraImpl()`
- Es el objeto Java que contiene el código que suma los números de verdad.

- **Efectos:**

- **Arranca un mini-servidor web:** Java (JDK) tiene un servidor HTTP interno ligero. Esta línea lo inicia y empieza a escuchar en el puerto 8080.
- **Genera el WSDL:** Analiza tu clase `CalculadoraImpl` y crea el contrato XML automáticamente en memoria.
- **Expone el servicio:** A partir de este momento, si alguien envía un XML SOAP a esa URL, este objeto responderá.

# El Cliente (Service Consumer)

JAVA

```

package com.ejemplo;

import jakarta.xml.ws.Service;
import javax.xml.namespace.QName; // QName sigue siendo parte del JDK
estándar (javax)
import java.net.URL;

public class Cliente {
    public static void main(String[] args) {
        try {
            // 1. URL donde está el contrato (WSDL)
            URL wsdlURL = new URL("http://localhost:8080/miCalculadora?
wsdl");

            // 2. Qualified Name (QName) del servicio
            // Estos nombres (Namespace y ServiceName) están definidos
            dentro del XML del WSDL
            QName qname = new QName("http://ejemplo.com/",
"CalculadoraImplService");

            // 3. Crear la fábrica de servicios (Usando JAKARTA)
            Service service = Service.create(wsdlURL, qname);

            // 4. Obtener el "Stub" (el objeto proxy que implementa nuestra
            interfaz)
            Calculadora calculadoraProxy =
service.getPort(Calculadora.class);

            // 5. Invocar el método remoto
            System.out.println("Cliente: Enviando petición sumar(10,
20)...");

            // --- AQUÍ OCURRE EL MARSHALLING (Java -> XML SOAP) ---
            int resultado = calculadoraProxy.sumar(10, 20);
            // --- AQUÍ OCURRE EL UNMARSHALLING (XML SOAP -> Java) ---

            System.out.println("Cliente: El resultado recibido es " +
resultado);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

```

    }
}

}

```

**URL(string\_spec):**

- **Argumentos:**

- **spec (String):** La dirección donde se encuentra el **WSDL** (el contrato).
- **Nota:** Fíjate que al final se le añade **?wsdl**. Sin esto, accederías al servicio, pero el cliente lo que necesita primero es leer las instrucciones (el manual).

- **Efectos:**

- Simplemente crea un objeto que apunta a ese recurso en internet. No conecta todavía, solo prepara la dirección para que la fábrica de servicios sepa dónde ir a leer las instrucciones.

**QName(namespaceURI, localPart)**. Es el concepto más "raro" para quien viene de Java puro. **QName** significa **Qualified Name** (Nombre Cualificado).

- **Argumentos:**

1. **namespaceURI**: Es el "Espacio de Nombres". En XML, para evitar que dos servicios se llamen igual, se les pone un apellido que suele ser una URL invertida.
  - **En tu código:** "<http://ejemplo.com/>". Esto sale del **package com.ejemplo** de tu servidor.
2. **localPart**: El nombre del servicio dentro de ese espacio.
  - **En tu código:** "**CalculadoraImplService**". Por defecto, JAX-WS le añade la palabra "Service" al nombre de tu clase implementadora.

- **Efectos:**

- Sirve para **identificar de forma única** al servicio dentro del archivo WSDL.
- Un WSDL podría describir 10 servicios distintos. Con el **QName**, le estás diciendo a Java: *"De todo el archivo WSDL, quiero hablar concretamente con EL servicio llamado 'CalculadoraImplService' que pertenece a la empresa 'ejemplo.com'"*.

**Service.create(wsdlURL, qname)**

- **Argumentos:**

1. **wsdlURL**: La ubicación del archivo de instrucciones (el manual WSDL).
2. **qname**: El nombre exacto de la empresa dentro de ese manual.

- **¿Qué hace realmente? (La "Fábrica")** Esta función es el **Constructor Maestro**.

1. **Descarga y Lee**: Va a la URL que le diste, se baja el XML del WSDL y lo lee entero.

2. **Valida:** Comprueba que dentro de ese XML existe realmente un servicio que se llame como el **qname** que le has pasado. Si el nombre no coincide, aquí saltará un error.
  3. **Prepara la "Agencia":** Crea un objeto **Service** en memoria. Este objeto **NO** es la calculadora todavía. Es una "fábrica" que sabe cómo crear calculadoras (o cualquier otro puerto definido en el WSDL).
- **Efecto en tu programa:** Te devuelve un objeto **Service** (de la librería **jakarta.xml.ws.Service**). Este objeto es el padre de todos los proxies. Sin él, no puedes pedir puertos (**getPort**).

```
service.getPort(serviceEndpointInterface)
```

- **Argumentos:**
  - **serviceEndpointInterface (Class):** Le pasas la clase de la **Interfaz Java (Calculadora.class)**.
  - **Ojo:** No le pasas la implementación (el código real), solo la interfaz (el contrato). El cliente no sabe cómo se suma, solo sabe que existe un método **sumar**.
- **Efectos (La Magia del Proxy):**
  - **Crea el STUB (Proxy):** Esta función te devuelve un objeto que **parece** local (implementa la interfaz **Calculadora**), pero que es mentira.
  - Este objeto (**calculadoraProxy**) es un "traductor".
  - Cuando tú llamas a **.sumar(10, 20)** sobre este objeto, él no suma nada. Lo que hace es:
    1. Coge el 10 y el 20.
    2. Construye el XML SOAP (Marshalling).
    3. Envía el XML por la red a la URL que definiste.
    4. Espera la respuesta.
    5. Desempaquetta el XML de respuesta (Unmarshalling).
    6. Te devuelve el **int** resultado.

### Nota

En este tema miraros las cosas por encima, yo le metí mucho detalle y mucha explicación pero porque no entendía un carallo. Pero con entender un poco los conceptos principales malo será.