

# Servicios Web y SOA

## Índice

1	Introducción a los Servicios Web. Invocación de servicios web SOAP.....	4
1.1	¿Qué es un Servicio Web?.....	4
1.2	Características de los Servicios Web.....	5
1.3	Tipos de servicios Web.....	5
1.4	Arquitectura de los Servicios Web.....	6
1.5	Servicios Web SOAP y Java EE.....	7
1.6	Tecnologías básicas para Servicios Web.....	9
1.7	Interoperabilidad de los WS: Metro y JAX-WS.....	16
1.8	Los servicios Web desde la vista del Cliente.....	17
1.9	Ficheros WSDL y de esquema con Netbeans.....	18
1.10	Tipos de acceso para invocar servicios Web JAX-WS.....	25
1.11	Invocación de servicios web JAX-WS con JDK 1.6.....	26
1.12	Invocación de servicios web JAX-WS desde una clase Java con Maven.....	27
1.13	Invocación de servicios web JAX-WS desde una aplicación Web con Maven...	33
1.14	Invocación de servicios web con Netbeans.....	39
1.15	Gestor de servicios web de Netbeans.....	43
1.16	Interfaz de invocación dinámica (DII).....	45
2	Ejercicios. Invocación de Servicios Web SOAP.....	48
2.1	Repositorio Mercurial para los ejercicios.....	48
2.2	Clientes para servicio web hola.....	48
2.3	Cliente para el servicio web Calculadora.....	49
2.4	Cliente para el servicio web Parte Metereológico.....	50
3	Creación de Servicios Web SOAP.....	52
3.1	Los servicios Web desde la vista del Servidor.....	52
3.2	El modelo de programación JAX-WS.....	54
3.3	Implementación del servicio JAX-WS con el modelo de servlets .....	55

3.4 Implementación del servicio Web con el modelo EJB .....	60
3.5 Empaquetado y despliegue de un servicio Web.....	61
3.6 Creación de un servicio Web con JDK 1.6.....	64
3.7 Creación de un servicio Web JAX-WS con Maven.....	65
3.8 Creación de servicios web con Netbeans.....	67
3.9 Creación de servicios a partir de EJBs existentes.....	72
3.10 Creación de servicios a partir del WSDL.....	74
3.11 Paso de datos binarios.....	75
3.12 Servicios web con estado.....	79
4 Ejercicios. Creación de Servicios Web SOAP.....	83
4.1 Creación de un servicio web básico.....	83
4.2 Validación de NIFs.....	83
4.3 Tienda de DVDs.....	84
5 Orquestación de Servicios: BPEL.....	87
5.1 Orquestación frente a Coreografía.....	87
5.2 El lenguaje BPEL.....	91
5.3 Estructura de un proceso BPEL.....	92
5.4 Relación de BPEL con BPMN.....	100
5.5 Pasos para desarrollar un proceso de negocio con BPEL.....	101
5.6 Despliegue y pruebas del proceso BPEL.....	102
5.7 Creación y ejecución de casos de prueba.....	105
6 Ejercicios de Orquestación de servicios BPEL.....	107
6.1 Pasos previos con Netbeans.....	107
6.2 Creamos el proyecto BPEL.....	108
6.3 WSDL y esquema de nombres del proyecto BPEL.....	109
6.4 Lógica del proceso BPEL.....	111
6.5 Desplegamos el proyecto en el servidor de aplicaciones.....	116
6.6 Creamos un conductor de pruebas.....	117
6.7 Ejecutamos las pruebas sobre SynchronousSampleApplication.....	117
6.8 Cambios en la lógica de negocio.....	118
7 Procesos BPEL síncronos y asíncronos.....	120
7.1 Invocación de servicios Web.....	120

7.2 Invocación de servicios Web asíncronos.....	122
7.3 Procesos BPEL síncronos frente a procesos BPEL asíncronos.....	124
7.4 Partner Link Types en procesos asíncronos.....	125
7.5 Ciclo de vida de los procesos de negocio.....	127
7.6 Correlations.....	128
8 Ejercicios de Procesos PBPEL síncronos y asíncronos.....	137
8.1 Proceso BPEL síncrono: Servicio de orden de compra.....	137
8.2 Proceso BPEL asíncrono. Uso de correlación: Hola Mundo.....	149

## 1. Introducción a los Servicios Web. Invocación de servicios web SOAP.

El diseño del software tiende a ser cada vez más modular. Las aplicaciones se componen de una serie de componentes (servicios) reutilizables, que pueden encontrarse distribuidos a lo largo de una serie de máquinas conectadas en red.

Los Servicios Web nos permitirán distribuir nuestra aplicación a través de Internet, pudiendo una aplicación utilizar los servicios ofrecidos por cualquier servidor conectado a Internet. La cuestión clave cuando hablamos de servicios Web es la interoperabilidad entre las aplicaciones.

### 1.1. ¿Qué es un Servicio Web?

Un Servicio Web es un componente al que podemos acceder mediante protocolos Web estándar, utilizando XML para el intercambio de información.

Normalmente nos referimos con Servicio Web a una colección de procedimientos (métodos) a los que podemos llamar desde cualquier lugar de Internet o de nuestra intranet, siendo este mecanismo de invocación totalmente independiente de la plataforma que utilicemos y del lenguaje de programación en el que se haya implementado internamente el servicio.

Cuando conectamos con un servidor web desde nuestro navegador, el servidor nos devuelve la página web solicitada, que es un documento que se mostrará en el navegador para que lo visualice el usuario, pero es difícilmente entendible por una máquina. Podemos ver esto como web para humanos. En contraposición, los Servicios Web ofrecen información con un formato estándar que puede ser entendido fácilmente por una aplicación. En este caso estaríamos ante una web para máquinas.

Los servicios Web son componentes de aplicaciones distribuidas que están disponibles de forma externa. Se pueden utilizar para integrar aplicaciones escritas en diferentes lenguajes y que se ejecutan en plataformas diferentes. Los servicios Web son independientes de lenguaje y de la plataforma gracias a que los vendedores han admitido estándares comunes de Servicios Web.

El WC3 (*World Wide Web Consortium*) define un servicio Web como un sistema software diseñado para soportar interacciones máquina a máquina a través de la red. Dicho de otro modo, los servicios Web proporcionan una forma estándar de interoperar entre aplicaciones software que se ejecutan en diferentes plataformas. Por lo tanto, su principal característica su gran **interoperabilidad** y extensibilidad así como por proporcionar información fácilmente procesable por las máquinas gracias al uso de XML. Los servicios Web pueden combinarse con muy bajo acoplamiento para conseguir la realización de operaciones complejas. De esta forma, las aplicaciones que proporcionan servicios simples pueden interactuar con otras para "entregar" servicios sofisticados

añadidos.

#### Historia de los servicios Web

Los servicios Web fueron "inventados" para solucionar el problema de la **interoperabilidad** entre las aplicaciones. Al principio de los 90, con el desarrollo de Internet/LAN/WAN, apareció el gran problema de integrar aplicaciones diferentes. Una aplicación podía haber sido desarrollada en C++ o Java, y ejecutarse bajo Unix, un PC, o un computador *mainframe*. No había una forma fácil de intercomunicar dichas aplicaciones. Fué el desarrollo de XML el que hizo posible compartir datos entre aplicaciones con diferentes plataformas hardware a través de la red, o incluso a través de Internet. La razón de que se llamasen servicios Web es que fueron diseñados para residir en un servidor Web, y ser llamados a través de Internet, típicamente via protocolos HTTP, o HTTPS. De esta forma se asegura que un servicio puede ser llamado por cualquier aplicación, usando cualquier lenguaje de programación, y bajo cualquier sistema operativo, siempre y cuándo, por supuesto, la conexión a Internet esté activa, y tenga un puerto abierto HTTP/HTTPS, lo cual es cierto para casi cualquier computador que disponga de acceso a Internet.

## 1.2. Características de los Servicios Web

Las características deseables de un Servicio Web son:

- Un servicio debe poder ser **accesible a través de la Web**. Para ello debe utilizar protocolos de transporte estándares como HTTP, y codificar los mensajes en un lenguaje estándar que pueda conocer cualquier cliente que quiera utilizar el servicio.
- Un servicio debe contener una **descripción de sí mismo**. De esta forma, una aplicación podrá saber cuál es la función de un determinado Servicio Web, y cuál es su interfaz, de manera que pueda ser utilizado de forma automática por cualquier aplicación, sin la intervención del usuario.
- Debe poder **ser localizado**. Deberemos tener algún mecanismo que nos permita encontrar un Servicio Web que realice una determinada función. De esta forma tendremos la posibilidad de que una aplicación localice el servicio que necesite de forma automática, sin tener que conocerlo previamente el usuario.

## 1.3. Tipos de servicios Web

A nivel conceptual, un servicio es un componente software proporcionado a través de un *endpoint* accesible a través de la red. Los servicios productores y consumidores utilizan mensajes para intercambiar información de invocaciones de petición y respuesta en forma de documentos auto-contenidos que hacen muy pocas asunciones sobre las capacidades tecnológicas de cada uno de los receptores.

#### Definición de endpoint

Los servicios pueden interconectarse a través de la red. En una arquitectura orientada a servicios, cualquier interacción punto a punto implica dos *endpoints*: uno que proporciona un servicio, y otro de lo consume. Es decir, que un *endpoint* es cada uno de los "elementos", en nuestro caso nos referimos a servicios, que se sitúan en ambos "extremos" de la red que sirve de canal de

comunicación entre ellos. Cuando hablamos de servicios Web, un *endpoint* se especifica mediante una URI.

A nivel técnico, los servicios pueden implementarse de varias formas. En este sentido, podemos distinguir dos tipos de servicios Web: los denominados servicios Web "grandes" ("*big*" *Web Services*), los llamaremos servicios Web SOAP, y servicios Web RESTful.

### Servicios Web SOAP

Los servicios Web SOAP, o servicios Web "*big*", utilizan mensajes XML para comunicarse que siguen el estándar SOAP (*Simple Object Access Protocol*), un lenguaje XML que define la arquitectura y formato de los mensajes. Dichos sistemas normalmente contienen una descripción legible por la máquina de la descripción de las operaciones ofrecidas por el servicio, escrita en WSDL (*Web Services Description Language*), que es un lenguaje basado en XML para definir las interfaces sintácticamente.

El formato de mensaje SOAP y el lenguaje de definición de interfaces WSDL se ha extendido bastante, y muchas herramientas de desarrollo, por ejemplo Netbeans, pueden reducir la complejidad de desarrollar aplicaciones de servicios Web.

El diseño de un servicio basado en SOAP debe establecer un contrato formal para describir la interfaz que ofrece el servicio Web. WSDL puede utilizarse para describir los detalles del contrato, que pueden incluir mensajes, operaciones, *bindings*, y la localización del servicio Web. También deben tenerse en cuenta los requerimientos no funcionales, como por ejemplo las transacciones, necesidad de mantener el estado (*addressing*), seguridad y coordinación

En este módulo vamos a hablar únicamente en los Servicios Web SOAP.

### Servicios Web RESTful

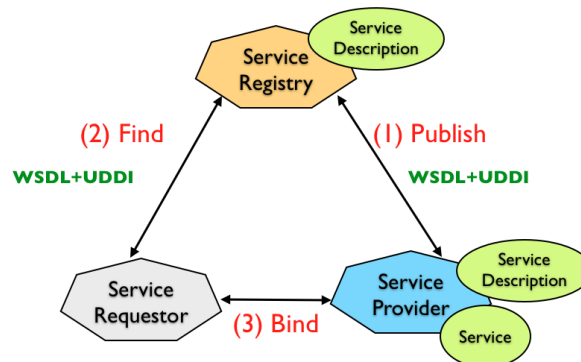
Los servicios Web RESTful (*Representational State Transfer Web Services*) son adecuados para escenarios de integración básicos *ad-hoc*. Dichos servicios Web se suelen integrar mejor con HTTP que los servicios basado en SOAP, ya que no requieren mensajes XML o definiciones del servicio en forma de fichero WSDL

Los servicios Web REST utilizan estándares muy conocidos como HTTP, SML, URI, MIME, y tienen una infraestructura "ligera" que permite que los servicios se construyan utilizando herramientas de forma mínima. Gracias a ello, el desarrollo de servicios RESTful es barato y tiene muy pocas "barreras" para su adopción.

## 1.4. Arquitectura de los Servicios Web

Los servicios Web presentan una arquitectura orientada a servicios que permite crear una definición abstracta de un servicio, proporcionar una implementación concreta de dicho servicio, publicar y localizar un servicio, seleccionar un instancia de un servicio, y utilizar dicho servicio con una elevada interoperabilidad. Es posible desacoplar la

implementación del servicio Web y su uso por parte de un cliente. También es posible desacoplar la implementación del servicio y de cliente. Las implementaciones concretas del servicio pueden desacoplarse a nivel de lógica y transporte. La siguiente figura muestra el diagrama de una arquitectura orientada a servicios.



Arquitectura orientada a Servicios

El proveedor del servicio define la descripción abstracta de dicho servicio utilizando un lenguaje de descripción de Servicios Web (**WSDL: Web Services Description Language**). A continuación se crea un Servicio concreto a partir de la descripción abstracta del servicio, produciendo así una descripción concreta del servicio en WSDL. Dicha descripción concreta puede entonces publicarse en un servicio de registro como por ejemplo **UDDI (Universal Description, Discovery and Integration)**. Un cliente de un servicio puede utilizar un servicio de registro para localizar una descripción de un servicio, a partir de la cual podrá seleccionar y utilizar una implementación concreta de dicho servicio.

La descripción abstracta se define en un documento WSDL como un **PortType**. Una instancia concreta de un Servicio se define mediante un elemento **port** de un WSDL (consistente a su vez en una combinación de un PortType, un **binding** de codificación y transporte, más una dirección). Un conjunto de **ports** definen un elemento **service** de un WSDL.

## 1.5. Servicios Web SOAP y Java EE

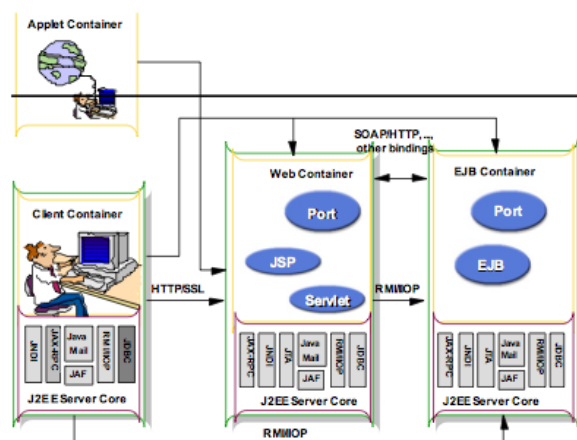
Como ya se ha dicho, en este módulo vamos a centrarnos en los servicios Web SOAP. Aunque no existe una definición comúnmente aceptada para este tipo de servicios, utilizaremos la siguiente extraída de la especificación **JSR-109 (Implementing Web Services)**, que define el modelo de programación y arquitectura del soporte de ejecución (*run-time*) para implementar servicios Web en Java.

### Definición de Servicio Web:

Un servicio Web es un componente con las siguientes características:

- Implementa los métodos de una interfaz descrita mediante un WSDL. Dichos métodos se implementan utilizando un EJB de sesión de tipo *Stateless/Singleton* o bien un componente web JAX-WS
- Un servicio Web puede tener publicada su interfaz en uno o más "registros" durante su despliegue
- La implementación de un Servicio Web, la cual utiliza solamente la funcionalidad descrita por su especificación, puede desplegarse en cualquier servidor de aplicaciones que cumple con las especificaciones Java EE
- Los servicios requeridos en tiempo de ejecución (*run-time*), tales como atributos de seguridad, se separan de la implementación del servicio. Se utilizarán herramientas adicionales que pueden definir dichos requerimientos durante el ensamblado o despliegue
- Un contenedor actúa como mediador para acceder al servicio

La especificación de Java EE para servicios Web define una serie de relaciones arquitectónicas requeridas para dichos servicios, que mostramos en la siguiente figura. Se trata de **relaciones lógicas** que no imponen requerimiento alguno para el proveedor del contenedor sobre cómo estructurar los contenedores y los procesos. Como añadido para la plataforma Java EE se incluye un componente **port** que depende de la funcionalidad de contenedor proporcionada por los contenedores web y EJB, y del transporte SOAP/HTTP.



Arquitectura Java EE

Los servicios Web para Java EE requieren que un componente **Port** pueda ser referenciado desde un cliente, así como desde los contenedores web y EJB. No se requiere que haya un *Port* accesible desde un contenedor de *applets*.

Los servicios Web para Java EE pueden implementarse de dos formas: como una clase Java que se ejecuta en un contenedor Web (según el modelo de programación definido en JAX-WS, y que veremos más adelante), o como un EJB de sesión *stateless* o *singleton* en un contenedor EJB.



El contenedor del servicio Web debe proporcionar la gestión del ciclo de vida de la implementación del servicio, además de proporcionar soporte adicional para la gestión de concurrencia de la invocación de los métodos del servicio, y soporte para la seguridad.

## 1.6. Tecnologías básicas para Servicios Web

Tenemos una serie de tecnologías, todas ellas basadas en XML, que son fundamentales para el desarrollo de Servicios Web. Estas tecnologías son independientes tanto del SO como del lenguaje de programación utilizado para implementar dichos servicios. Por lo tanto, serán utilizadas para cualquier Servicio Web, independientemente de la plataforma sobre la que construyamos dichos servicios (como puede ser J2EE o .NET).

Los protocolos utilizados en los Servicios Web se organizan en una serie de capas:

Capa	Descripción
<i>Transporte de servicios</i>	Es la capa que se encarga de transportar los mensajes entre aplicaciones. Normalmente se utiliza el protocolo <b>HTTP</b> para este transporte, aunque los servicios web pueden viajar mediante otros protocolos de transferencia de hipertexto como SMTP, FTP o BEEP.
<i>Mensajería XML</i>	Es la capa responsable de codificar los mensajes en XML de forma que puedan ser entendidos por cualquier aplicación. Puede implementar los protocolos XML-RPC o <b>SOAP</b> .
<i>Descripción de servicios</i>	Se encarga de definir la interfaz pública de un determinado servicio. Esta definición se realiza mediante <b>WSDL</b> .
<i>Localización de servicios</i>	Se encarga del registro centralizado de servicios, permitiendo que estos sean anunciados y localizados. Para ello se utiliza el protocolo <b>UDDI</b> .

A continuación vamos a hablar con un poco más de detalle sobre las tecnologías de mensajería, descripción de servicios y localización. Más concretamente nos referimos a SOAP, WSDL y UDDI.

### 1.6.1. SOAP

Se trata de un protocolo derivado de XML que nos sirve para intercambiar información entre aplicaciones.

Normalmente utilizaremos SOAP para conectarnos a un servicio e invocar métodos remotos, aunque puede ser utilizado de forma más genérica para enviar cualquier tipo de contenido. Podemos distinguir dos tipos de mensajes según su contenido:

- **Mensajes orientados al documento:** Contienen cualquier tipo de contenido que queramos enviar entre aplicaciones.

- **Mensajes orientados a RPC:** Este tipo de mensajes servirá para invocar procedimientos de forma remota (*Remote Procedure Calls*). Podemos verlo como un tipo más concreto dentro del tipo anterior, ya que en este caso como contenido del mensaje especificaremos el método que queremos invocar junto a los parámetros que le pasamos, y el servidor nos deberá devolver como respuesta un mensaje SOAP con el resultado de invocar el método.

Puede ser utilizado sobre varios protocolos de transporte, aunque está especialmente diseñado para trabajar sobre HTTP.

Dentro del mensaje SOAP podemos distinguir los siguientes elementos:



Elementos de un mensaje SOAP

- Un sobre (*Envelope*), que describe el mensaje, a quien va dirigido, y cómo debe ser procesado. El sobre incluye las definiciones de tipos que se usarán en el documento. Contiene una cabecera de forma opcional, y el cuerpo del mensaje.
- Una cabecera (*Header*) opcional, donde podemos incluir información sobre el mensaje. Por ejemplo, podemos especificar si el mensaje es obligatorio (debe ser entendido de forma obligatoria por el destinatario), e indicar los actores (lugares por donde ha pasado el mensaje).
- El cuerpo del mensaje (*Body*), que contiene el mensaje en si. En el caso de los mensajes RPC se define una convención sobre como debe ser este contenido, en el que se especificará el método al que se invoca y los valores que se pasan como parámetros. Puede contener un error de forma opcional.
- Un error (*Fault*) en el cuerpo del mensaje de forma opcional. Nos servirá para indicar en una respuesta SOAP que ha habido un error en el procesamiento del mensaje de petición que mandamos.

Hemos visto como los mensajes SOAP nos sirven para intercambiar cualquier documento XML entre aplicaciones. Pero puede ocurrir que necesitemos enviar en el mensaje datos que no son XML, como puede ser una imagen. En ese caso tendremos que recurrir a la especificación de mensajes SOAP con anexos.

Los mensajes SOAP con anexos añaden un elemento más al mensaje:



Elementos de un mensaje SOAP con Anexos

- El anexo (*Attachment*), puede contener cualquier tipo de contenido (incluido el XML). De esta forma podremos enviar cualquier tipo de contenido junto a un mensaje SOAP.

Nuestro mensaje podrá contener tantos anexos como queramos.

Un ejemplo de mensaje SOAP es el siguiente:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <ns:getTemperatura xmlns:ns="http://j2ee.ua.es/ns">
      <area>Alicante</area>
    </ns:getTemperatura>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

En él estamos llamando a nuestro método `getTemperatura` para obtener información meteorológica, proporcionando como parámetro el área de la que queremos obtener la temperatura.

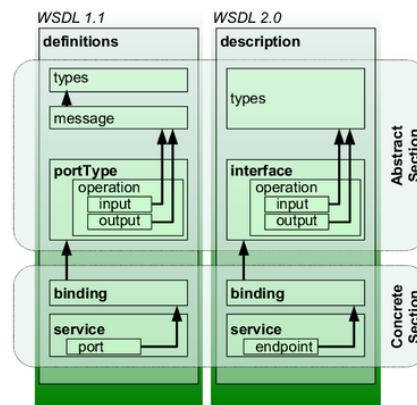
Podemos encontrar la especificación de SOAP y SOAP con anexos publicada en la página del W3C, en las direcciones <http://www.w3.org/TR/SOAP/> y <http://www.w3.org/TR/SOAP-attachments> respectivamente.

### 1.6.2. WSDL

**WSDL (Web Services Description Language)** es un lenguaje basado en XML utilizado para describir la funcionalidad que proporciona un servicio Web. Una descripción WSDL (fichero WSDL) de un servicio web proporciona una descripción entendible por la máquina (*machine readable*) de la interfaz del servicio Web, indicando cómo se debe llamar al servicio, qué parámetros espera, y qué estructuras de datos devuelve.

La versión actual de WSDL es la 2.0, en la que se ha cambiado el significado de la **D** por

**Definition.** La nueva versión plantea cambios de nomenclatura y estructura del fichero xml que contiene la descripción del servicio. En este módulo vamos a utilizar la versión anterior, la 1.1, puesto que es la que está soportada actualmente por Netbeans, así como por BPEL, que estudiaremos más adelante. En la figura siguiente mostramos la estructura que siguen los ficheros WSDL en ambas versiones, en donde podemos observar el cambio de nomenclatura en la versión 2.0.



Estructura de un documento WSDL

WSDL describe un servicio utilizando varios elementos (etiquetas xml). Dichos elementos podemos clasificarlos como abstractos o concretos. La **parte WSDL abstracta** describe las operaciones y mensajes con detalle. En otras palabras, la parte abstracta de un WSDL especifica **QUÉ** hace el servicio:

- Qué operaciones están disponibles
- Qué entradas, salidas, y mensajes de error tienen las operaciones
- Cuáles son las definiciones de los tipos para los mensajes de entrada, salida y error

En el mundo Java, podemos pensar en la parte abstracta de un WSDL como en la definición de una interfaz o una clase abstracta, con la definición de sus métodos, pero no sus implementaciones. La parte abstracta de un WSDL contiene dos componentes principales:

- Las operaciones que forman la definición de la interfaz
- Los tipos de datos para los parámetros de entrada, salida y error, de las operaciones

**La parte WSDL concreta** describe el cómo y dónde del servicio:

- Cómo tiene que llamar un cliente al servicio
- Qué protocolo debería usar
- Dónde está disponible el servicio

En el mundo Java podemos pensar en la parte concreta de un WSDL como en la implementación de la parte abstracta, aunque en términos de servicios Web, solamente describe dónde se encuentra dicha implementación para utilizarse. La parte concreta de un WSDL contiene dos componentes principales:

- Información de *enlazado* (*binding*) sobre el protocolo a utilizar
- La dirección en donde localizar el servicio

Vamos a ver con un poco más de detalle los elementos WSDL (tanto la parte abstracta como la concreta):

- **definitions:** Es el elemento raíz y permite especificar el espacio de nombres del documento *target namespace*, el nombre, y otros prefijos utilizados en el documento WSDL. Un ejemplo de definición de prefijo es:  
*xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/"*. Este prefijo especifica que todos los elementos dentro del documento de esquemas con el *target namespace* *"http://schemas.xmlsoap.org/wSDL/"* tendrán el prefijo *wSDL*.
- **types:** Se utiliza para definir los tipos de datos que se intercambiarán en el mensaje. Podemos definir dichos tipos directamente dentro de este elemento, o importar la definición de un fichero de esquema (fichero *xsd*). La definición de tipos puede verse como las definiciones Java de clase, con variables que pueden ser tipos primitivos o referencias a otras clases u objetos. Los tipos primitivos se definen en los espacios de nombres del *Schema* y normalmente nos referimos a ellos como *built-in types*. Éstos incluyen tipos simples tales como *string*, *int*, *double*,...
- **message:** Define los distintos mensajes que se intercambiarán durante el proceso de invocación del servicio. Se deberán definir los mensajes de entrada y salida para cada operación que ofrezca el servicio. Los mensajes muestran descripciones abstractas de los datos que se van a intercambiar.
- **portType:** Contiene una colección de una o más operaciones. Para cada operación indica cuáles son los mensajes de entrada y salida, utilizando para ello los mensajes definidos en el apartado anterior. Los *portTypes* son, por lo tanto, colecciones abstractas de operaciones soportadas por un servicio
- **binding:** Indica el protocolo de red y el formato de los datos para las operaciones de un *portType*. Los *bindings* son definiciones concretas de los *portTypes*. Un *portType* puede tener múltiples *bindings* asociados. El formato de datos utilizado para los mensajes de las operaciones del *portType* puede ser orientado al documento u orientado a RPC. Si es orientado al documento tanto el mensaje de entrada como el de salida contendrán un documento XML. Si es orientado a RPC el mensaje de entrada contendrá el método invocado y sus parámetros, y el de salida el resultado de invocar dicho método, siguiendo una estructura más restrictiva.
- **service:** Define el servicio como una colección de elementos *port* a los que se puede acceder. Un *port* se define asociando una dirección de red con un *binding*, de los definidos en el documento. Dicha dirección de red es la dirección (URL) donde el servicio actúa, y por lo tanto, será la dirección a la que las aplicaciones deberán conectarse para acceder al servicio.

Un documento WSDL de ejemplo es el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<definitions targetNamespace="http://jaxwsHelloServer/"
name="HelloService">
  <types>
```

```

<xsd:schema>
<xsd:import namespace="http://jaxwsHelloServer/"
  schemaLocation=
    "http://localhost:8080/JAXWSHelloAppServer/jaxwsHello?xsd=1"/>
</xsd:schema>
</types>

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

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

<portType name="Hello">
  <operation name="sayHello">
    <input wsam:Action="http://jaxwsHelloServer/Hello/sayHelloRequest"
      message="tns:sayHello"/>
    <output wsam:Action=
      "http://jaxwsHelloServer/Hello/sayHelloResponse"
      message="tns:sayHelloResponse"/>
  </operation>
</portType>

<binding name="HelloPortBinding" type="tns:Hello">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document"/>
  <operation name="sayHello">
    <soap:operation soapAction=""/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>

<service name="HelloService">
  <port name="HelloPort" binding="tns:HelloPortBinding">
    <soap:address location=
      "http://localhost:8080/JAXWSHelloAppServer/jaxwsHello"/>
  </port>
</service>
</definitions>

```

En este ejemplo se define un **servicio**, denominado `HelloService`. Dicho servicio está formado por el **port** `HelloPort`, al cual se accede con protocolo *http* (binding `transport=http`) y con mensajería *soap* con estilo *document* (binding `style=document`). Dicho **port** proporciona la operación (funcionalidad) `sayHello` (operation name= "sayHello"). Dicha operación utiliza la codificación *literal* en los mensajes de entrada y salida (body use= "literal"). Esto es lo que se conoce como operación de tipo **document/literal**.

El elemento `portType` define la operación `sayHello` a partir de los mensajes de entrada y salida que la componen.

En los elementos `message` vemos la descripción de los mensajes de entrada y salida de la operación `sayHello`. El mensaje de entrada contiene un dato de tipo `tns:sayHello`, y

el de salida es de tipo "tns:sayHelloResponse" (la cadena de saludo devuelta por el servicio). La definición de los tipos se encuentra en el fichero de esquema referenciado por la propiedad *schemaLocation* en una sub-etiqueta de <types>

Podemos encontrar la especificación de WSDL 1.1 publicada en la página del W3C, en la dirección <http://www.w3.org/TR/wsdl>.

### 1.6.3. UDDI

---

UDDI nos permite localizar Servicios Web. Para ello define la especificación para construir un directorio distribuido de Servicios Web, donde los datos se almacenan en XML. En este registro no sólo se almacena información sobre servicios, sino también sobre las organizaciones que los proporcionan, la categoría en la que se encuentran, y sus instrucciones de uso (normalmente WSDL). Tenemos por lo tanto 3 tipos de información relacionados entre sí:

- *Páginas blancas*: Datos de las organizaciones (dirección, información de contacto, etc).
- *Páginas amarillas*: Clasificación de las organizaciones (según tipo de industria, zona geográfica, etc).
- *Páginas verdes*: Información técnica sobre los servicios que se ofrecen. Aquí se dan las instrucciones para utilizar los servicios. Es recomendable que estas instrucciones se especifiquen de forma estándar mediante un documento WSDL.

Además, UDDI define una API para trabajar con dicho registro, que nos permitirá buscar datos almacenados en él, y publicar datos nuevos.

De esta forma, una aplicación podrá anunciar sus servicios en un registro UDDI, o bien localizar servicios que necesitemos mediante este registro.

Esta capacidad de localizar servicios en tiempo de ejecución, y de que una aplicación pueda saber cómo utilizarlo inmediatamente gracias a la descripción del servicio, nos permitirá realizar una integración débilmente acoplada de nuestra aplicación.

La interfaz de UDDI está basada en SOAP. Para acceder al registro se utilizarán mensajes SOAP, que son transportados mediante protocolo HTTP.

Podemos encontrar la especificación de UDDI, documentación, y más información en la dirección <http://uddi.xml.org/>.

Estos registros se utilizan normalmente de forma interna en organizaciones para tener un directorio organizado de servicios. Podemos encontrar varios registros proporcionados por diferentes proveedores. Destacamos **jUDDI**, un registro *open-source* de Apache. Este registro consiste en una aplicación web Java que puede instalarse en cualquier servidor con soporte para este tipo de aplicaciones, como puede ser Tomcat, y una base de datos, que podrá ser instalada en diferentes SGBD (MySQL, Postgres, Oracle, etc).

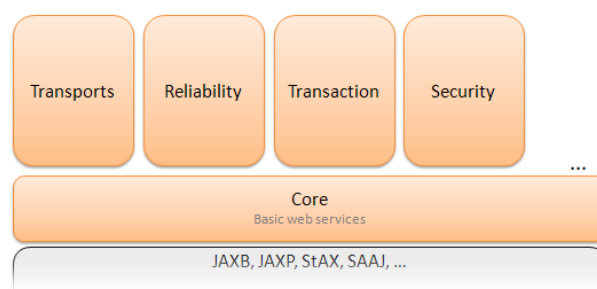
## 1.7. Interoperabilidad de los WS: Metro y JAX-WS

La interoperabilidad de los servicios Web constituye una iniciativa de Sun y Microsoft. El principal objetivo es proporcionar productos que sean capaces de interoperar a través de diferentes plataformas.

Metro (<http://metro.java.net>) es el producto resultante de la iniciativa de Sun para la interoperabilidad de los servicios Web utilizando la plataforma Java. De igual forma WCF (*Windows Communication Foundation*) es la aportación de Microsoft para la plataforma .NET.

Metro, por lo tanto, constituye la implementación por parte de Sun, de la pila (colección de tecnologías) de servicios Web (*web service stack*). La versión actual es la 2.1.1 y está formada por tres componentes principales, que podemos ver en la figura que se adjunta:

- Metro/WSIT 2.1.1: WSIT es un conjunto de tecnologías (Web Services Interoperable Technologies) que permiten la interoperabilidad con .NET. A nivel de transporte, Metro permite utilizar HTTP, MTOM, SOAP/TCP. También proporciona fiabilidad en el envío de mensajes (*Reliability*), implementando las especificaciones WS-ReliableMessaging. Asimismo Metro permite habilitar el uso de transacciones atómicas, para lo cual proporciona una implementación de las especificaciones WS-Coordination y WS-Atomic Transaction. La seguridad también está contemplada en Metro mediante la implementación de las especificaciones WS-Security y WS-Trust. Bootstrapping: WSDL; WS-Policy; WS-MetadataExchange
- JAX-WS RI 2.2.5: Es la implementación de referencia del estándar JAX-WS (especificación JSR 224: *Java API for XML-Based Web Services*). Proporciona las características básicas para la interoperabilidad de los servicios Web (*WS-I Basic Profile*: mensajería SOAP, WSDL, publicación de servicios en UDDI; *WS-I Attachment Profile*: utilización de SOAP con anexos; *WS-I Addressing*: utilización de espacios de nombres y ficheros de esquema)
- JAXB RI 2.2.4-1: Implementación de referencia del API para la capa de enlazado de datos (**JAXB**: *Java Architecture for XML binding*)



Componentes de Metro

Con JAX-WS, una invocación de una operación a un servicio web se representa con un



protocolo basado en XML, como por ejemplo SOAP. La especificación SOAP define la estructura del "envoltorio" (*envelope*) del mensaje, reglas de codificación, y convenciones para representar invocaciones y respuestas del servicio web. Estas llamadas y respuestas son transmitidas como mensajes SOAP (archivos XML) a través de HTTP.

Si bien los mensajes SOAP son complejos, la API JAX-WS oculta esta complejidad al desarrollador de la aplicación. En la parte del servidor, el desarrollador especifica las operaciones del servicio web definiendo métodos en una interfaz escrita en lenguaje Java. El desarrollador también codifica una o más clases que implementan dichos métodos. Los programas cliente también son fáciles de codificar. Un cliente crea un *proxy* (un objeto local que representa el servicio) y a continuación simplemente invoca los métodos sobre el *proxy*. Con JAX-WS, el desarrollador no necesita generar o "parsear" mensajes SOAP. El *runtime* de JAX-WS convierte las llamadas y respuestas del API en y desde los mensajes SOAP.

La siguiente figura muestra la interacción entre un cliente y un servicio web a través de JAX-WS.



Comunicación entre un servicio Web y un cliente a través de JAX-WS

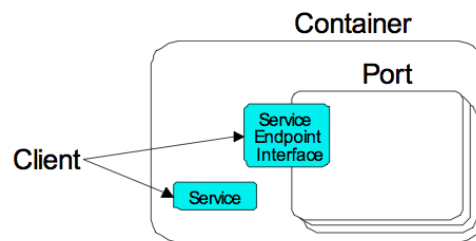
Metro (y consecuentemente, JAX-WS) está incluido en Glassfish 3.x por defecto, por lo que haremos uso de él para implementar nuestros Servicios Web y clientes correspondientes.

## 1.8. Los servicios Web desde la vista del Cliente

La vista del cliente de un servicio Web es bastante similar a la de un *Enterprise JavaBean*. Un cliente de un servicio Web puede ser otro servicio Web, un componente Java EE (componente web, componente EJB), incluyendo una aplicación cliente Java EE, o una aplicación Java arbitraria. Una aplicación no Java o un servicio Web para una aplicación no Java EE también podrían ser clientes de un Servicio Web, pero vamos a ceñirnos a la plataforma Java EE.

El cliente de un servicio Web puede ser remoto (no se requiere que resida en la misma máquina que el servicio Web) y se proporciona una total transparencia al respecto (el cliente no puede distinguir si está accediendo a un servidor local o remoto).

La siguiente figura ilustra la vista del cliente de un servicio Web, proporcionada por el proveedor del componente *Port* y el contenedor en el que éste reside. Además se incluye la clase/interfaz **Service** (**SI**: Service Interface), y la interfaz del **Endpoint** del servicio (**SEI**: Service Endpoint Interface).



Vista de un cliente de un servicio Web

La clase/interfaz **Service** (SI) define los métodos que un cliente puede utilizar para acceder a un **Port** de un servicio Web. Un cliente NO crea o destruye un Port. Utiliza la clase/interfaz Service para obtener el acceso a un Port. La clase/interfaz Service se define en la especificación JAX-WS, pero su comportamiento viene definido en el documento WSDL proporcionado por el proveedor del servicio Web. Las herramientas de despliegue del contenedor proporcionan una implementación de los métodos de la clase/interfaz Service generada por JAX-WS.

El cliente accede a una implementación de un servicio Web utilizando el SEI. Dicho SEI es especificado por el proveedor del servicio. Las herramientas del despliegue y el *run-time* del contenedor proporciona las clases en la parte del servidor que van a atender las peticiones SOAP sobre la implementación de dicho servicio de los métodos especificados en el SEI.

Un Port no tiene identidad para el cliente, que lo debe considerar como un objeto sin estado.

JAX-WS define un modelo de programación en el que se realiza un mapeado de un documento WSDL a Java. Dicho mapeado proporciona una factoría (*Service*) para seleccionar qué *Port* (agregado en el servicio) desea usar el cliente. Como veremos más adelante, la herramienta JAX-WS que proporciona las clases necesarias en la parte del cliente para poder acceder a un servicio web se denomina **wsimport**. En general, el transporte, codificación, y dirección del *Port* son transparentes para el cliente. El cliente solamente necesita realizar llamadas sobre la interfaz del *endpoint* del servicio (*Service Endpoint Interface*), utilizando el *PortType* correspondiente para acceder a dicho servicio.

En esta sesión vamos a explicar el modelo de programación del cliente de un servicio Web utilizando JAX-WS, pero antes vamos a ver cómo trabajar con ficheros wsdl y ficheros de esquema con Netbeans.

## 1.9. Ficheros WSDL y de esquema con Netbeans

Saber "leer" y/o crear un fichero WSDL es importante para trabajar con servicios Web SOAP. Por ello vamos a ver cómo, de forma sencilla, se puede trabajar con ficheros WSDL y de esquema utilizando Netbeans

### Importante

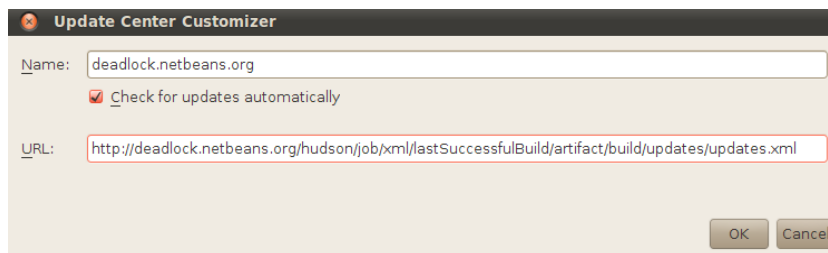
Para poder editar ficheros WSDL y de esquema aprovechando las ventajas de edición que nos proporciona NetBeans, tenemos que instalar el plugin correspondiente.

## Cómo instalar el plugin en Netbeans para editar ficheros WSDL y xsd

Para instalar el plugin, lo haremos desde el menú de NetBeans. Para ello iremos a *Tools->Plugins->Settings->Add*. A continuación tenemos que proporcionar un nombre (para identificar el sitio que contiene los plugins, y añadirlo a la lista de sitios en donde NetBeans buscará actualizaciones de los plugins), y la URL del sitio desde el cual nos podemos descargar e instalar el plugin.

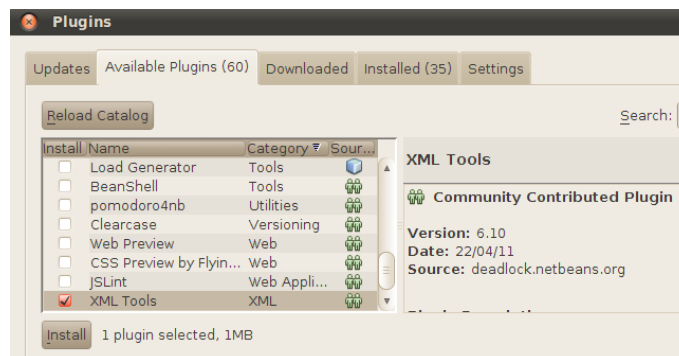
- Name: Deadlock.Netbeans.org
- URL: <http://deadlock.netbeans.org/hudson/job/xml/lastSuccessfulBuild/artifact/build/updates/updates.xml>

A continuación pulsamos OK.



URL de descarga del plugin XML

En la pestaña "Available Plugins", marcaremos "XML Tools", y procedemos a su instalación:



Instalación del plugin XML

## FICHEROS DE ESQUEMA

**XML Schema** es una recomendación del W3C, que proporciona mecanismos para definir la estructura, contenido y semántica de un documento XML. Un fichero WSDL utiliza el

"lenguaje de esquemas" (*XML Schema*) para definir los tipos de datos y/o elementos utilizados por las operaciones de un servicio Web.

El bloque de construcción principal de un documento XML es **element**. La definición de un elemento *element* debe contener una propiedad `name`, que representa su nombre, y una propiedad `type` para indicar el tipo de elemento. Podemos utilizar alguno de los tipos predefinidos (*built-in types*), por ejemplo *xs:string*, *xs:integer*, o bien podemos definir nuevos tipos utilizando etiquetas *simpleType* o *complexType*.

Por ejemplo, podemos crear la siguiente definición del elemento *Customer\_Addres* que representa la dirección de un cliente:

```
<xs:element name="CustomerAddress" type="xs:string"/>
```

Ahora podemos utilizar la definición anterior para expresar la estructura del siguiente mensaje de respuesta de nuestro servicio Web

```
<message name="msgResponse">
  <part name="parameters" element="tns:CustomerAddress"/>
</message>
```

Según esto, un ejemplo de mensaje de respuesta por parte de nuestro servicio Web podría ser éste:

```
<Customer_address>Calle de los Pinos, 37</Customer_address>
```

En lugar de utilizar tipos primitivos, podemos definir nuevos tipos utilizando la etiqueta *ComplexType* (es un contenedor de elementos). Por ejemplo vamos a definir los tipos *CustomerType* y *AddressType* y el elemento *Customer*, la siguiente forma:

```
<xsd:element name="Customer" type="tns:CustomerType"/>
<xsd:complexType name="CustomerType">
  <xsd:sequence>
    <xsd:element name="Phone" type="xsd:integer"/>
    <xsd:element name="Addresses" type="tns:AddressType"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="AddressType">
  <xsd:sequence>
    <xsd:element name="Address1" type="xsd:string"/>
    <xsd:element name="Address2" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

Ahora podemos crear un segundo mensaje de respuesta en nuestro WSDL que haga uso del nuevo tipo *CustomerInfo*:

```
<message name="msgResponse2">
```

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

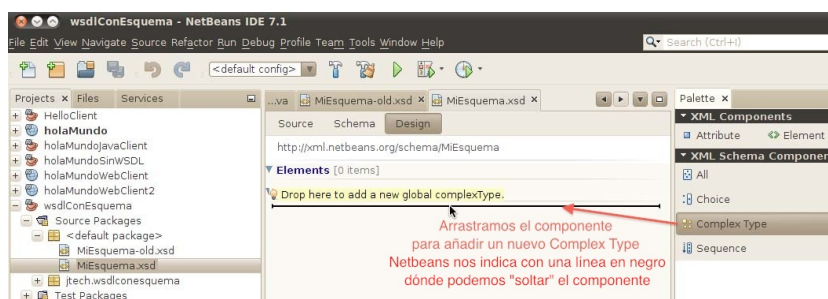
Un ejemplo de mensaje de respuesta con esta definición podría ser:

```
<Customer>
  <Phone>12345678</Phone>
  <Address1>Calle de los Pinos, 37</Address1>
  <Address2>Calle de los Manzanos, 25</Address2>
</Customer>
```

Para crear ficheros de esquema desde Netbeans, tenemos que crear un nuevo fichero con extensión xsd. Con botón derecho, seleccionamos *New->Other->XML->XML Schema*. Debemos proporcionar un nombre para nuestro fichero .xsd, por ejemplo *MiEsquema*. Vamos explicar como crear, de forma visual, el esquema anterior.

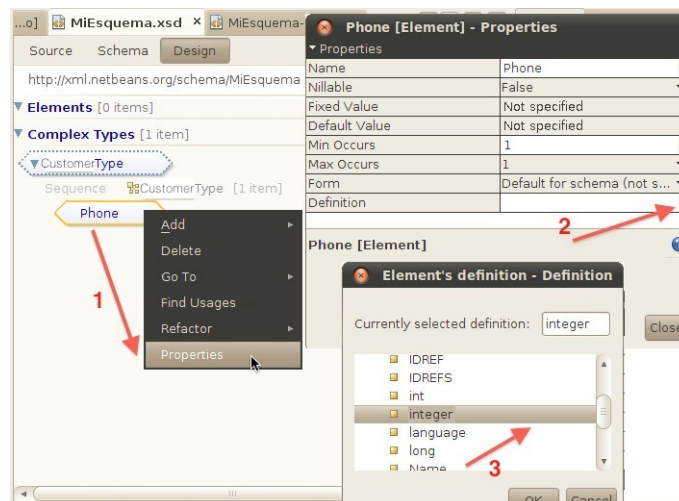
El editor de ficheros de esquemas de Netbeans nos permite trabajar con tres vistas: la vista de fuentes (*Source*), en la que podemos trabajar directamente con xml, la vista de esquema (*Schema*): que nos muestra el contenido del fichero en forma de árbol, o en forma de "columnas", y la vista de diseño (*Design*). Esta última vista nos permite editar un fichero de esquema de forma visual, arrastrando componentes desde una paleta y editando sus propiedades, de forma que con unos cuantos "clicks" de ratón Netbeans genera el correspondiente código xml, que podemos ver en la vista de fuentes. Las tres vistas están sincronizadas en todo momento, de forma que los cambios en cualquiera de ellas, se reflejarán automáticamente en las otras dos.

Vamos a crear el tipo *CustomerType*. Para ello tendremos que "arrastrar" con el botón izquierdo del ratón el componente *ComplexType* desde la paleta de componentes de la derecha, y situarlo justo debajo de *Complex Types*, en el editor de diseño. Cuando nos situemos con el ratón en el lugar correcto, nos aparecerá una línea en negro y sólo entonces podremos "soltar" el dedo del ratón, y nos creará un nuevo elemento *ComplexType* en la vista de diseño. Vamos a ponerle el nombre *CustomerType*. Si seleccionamos el componente, haciendo doble "click" sobre el mismo podemos cambiar el nombre del componente. Si seleccionamos el componente y pulsamos botón derecho, nos aparece un menú, con la opción "Properties" con el que podemos borrar, añadir elementos,..., y editar las propiedades del componente en cualquier momento.



Arrastramos el componente ComplexType a la vista de diseño

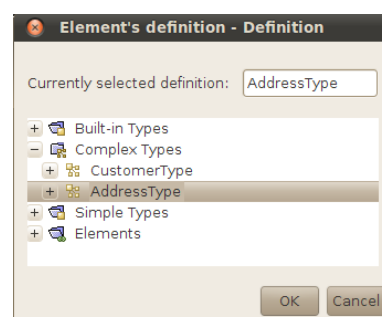
A continuación añadiremos un componente *element* a *CustomerType*, arrastrando dicho componente desde la paleta hasta situarnos sobre *CustomerType* antes de soltar el dedo del ratón. Llamaremos al nuevo elemento "Phone". Seguidamente, con el botón derecho, elegimos la opción "Properties" del menú contextual, y pinchamos sobre el botón con tres puntos correspondiente al ítem "Definition", y seleccionamos el tipo predefinido integer (dentro del grupo *Built-In Types*)



Añadimos el elemento Phone a AddressType

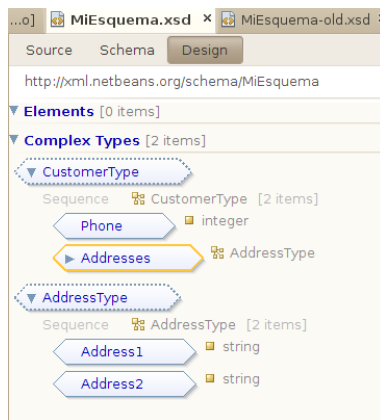
Repetimos los pasos anteriores para añadir un nuevo componente *ComplexType* al que llamaremos "AddressType", con los sub-elementos "Address1", y "Address2", ambas de tipo "string".

Para finalizar, crearemos un nuevo sub-elemento de "CustomerType", al que llamaremos "Addresses". En este caso, el tipo que asociaremos a dicho elemento será el tipo "AddressType", dentro del grupo "Complex Types"



Seleccionamos el tipo AddressType

El resultado final en la vista de diseño debe ser éste:



Vista de diseño del esquema MiEsquema.xsd

Si cambiamos a la vista de fuentes (*Source*) veremos lo siguiente:

```

1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4    targetNamespace="http://xml.netbeans.org/schema/MiEsquema"
5    xmlns:tns="http://xml.netbeans.org/schema/MiEsquema"
6    elementFormDefault="qualified">
7    <xsd:complexType name="CustomerType">
8      <xsd:sequence>
9        <xsd:element name="Phone" type="xsd:integer"/>
10       <xsd:element name="Addresses" type="tns:AddressType"/>
11      </xsd:sequence>
12    </xsd:complexType>
13    <xsd:complexType name="AddressType">
14      <xsd:sequence>
15        <xsd:element name="Address1" type="xsd:string"/>
16        <xsd:element name="Address2" type="xsd:string"/>
17      </xsd:sequence>
18    </xsd:complexType>
19  </xsd:schema>
20

```

Vista de fuentes del esquema MiEsquema.xsd

## FICHEROS WSDL

Ya hemos explicado el contenido de un fichero WSDL. Netbeans nos permite crear ficheros WSDL de forma "visual", simplificando de forma significativa el proceso de edición de la especificación WSDL de nuestro servicio web.

Para crear un fichero wsdl con NetBeans, activamos el menú contextual con el con botón derecho y seleccionamos *New->Other->XML->WSDL Document*. Podemos elegir entre crear un documento WSDL abstracto o concreto. En el primer caso solamente proporcionamos el *portType* y las operaciones que ofrece nuestro servicio. En el segundo caso, además, proporcionaremos la información de *binding* y tipo de codificación del servicio. La siguiente figura muestra la creación de un fichero wsdl concreto con nombre "conversion", y con un enlazado "SOAP", con codificación "Document/Literal".

**New WSDL Document**

**Steps**

1. Choose File Type
2. **Name and Location**
3. Abstract Configuration
4. Concrete Configuration

**Name and Location**

File Name:

Project:

Folder:

Created File:

Target Namespace:

WSDL Type: ☐ Abstract WSDL Document ☒ Concrete WSDL Document

Binding:

Type:

Documento WSDL concreto con codificación Document/Literal

El siguiente paso es determinar la **configuración abstracta** de nuestro WSDL. Tal y como se aprecia en la siguiente figura, se trata de indicar el nombre del *portType*, la operación que proporciona, así como el tipo de operación (elegiremos una operación síncrona, es decir, de tipo petición-respuesta (*Request-Response*)), y los mensajes de entrada, salida y situaciones excepcionales (*Fault*).

**New WSDL Document**

**Steps**

1. Choose File Type
2. Name and Location
3. **Abstract Configuration**
4. Concrete Configuration

**Abstract Configuration**

Port Type Name:

Operation Name:

Operation Type:

**Input:**

Message Part Name	Element Or Type
euros	ns:euroType

**Output:**

Message Part Name	Element Or Type
ptas	ns:ptasType

**Fault:**

Message Part Name	Element Or Type
-------------------	-----------------

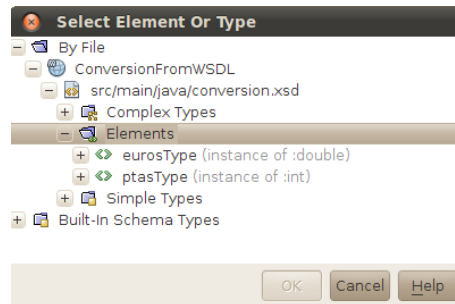
☒ Generate partnerlinktype automatically.

Configuración abstracta del documento WSDL

En un documento WSDL de tipo *document/literal*, utilizaremos un tipo XML predefinido, o bien de tipo `<element>` para las diferentes "partes" de los mensajes de entrada/salida/excepciones. En este último caso, tendremos que disponer de la definición del tipo en el fichero de esquema (fichero *xsd*) que asociaremos al documento WSDL. La siguiente figura muestra que los tipos "euroType" y "ptasType" se encuentran definidos

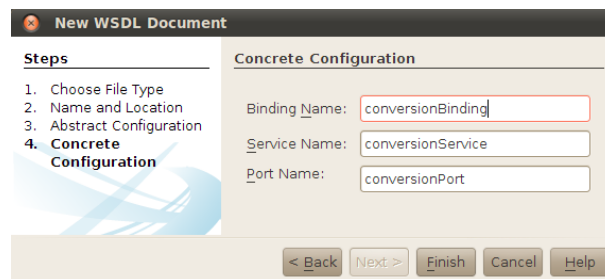


en el fichero "conversion.xsd"



Selección del tipo de las partes de los mensajes

Finalmente especificaremos la **configuración concreta** del WSDL, indicando el nombre de *binding*, así como el nombre del servicio y del *port* asociado.



Configuración concreta del documento WSDL

En la siguiente sesión veremos que, a partir de un wsdl, podemos generar un servicio web.

## 1.10. Tipos de acceso para invocar servicios Web JAX-WS

Tenemos dos formas diferentes de invocar un Servicio Web utilizando JAX-WS (llamaremos servicios web JAX-WS a aquellos servicios que utilizan la librería JAX-WS):

- **Creación de un stub estático:** Consiste en generar una capa de *stub* por debajo del cliente de forma automática. Dicho *stub* implementará la misma interfaz que el servicio, lo cuál nos permitirá desde nuestro cliente acceder al Servicio Web a través del *stub* tal y como si estuviéramos accediendo directamente al servicio.

Para utilizar este mecanismo es recomendable contar con alguna herramienta dentro de nuestra plataforma que nos permita generar dicho *stub*, para no tener que encargarnos nosotros de realizar esta tarea manualmente.

- **Utilización de la Interfaz de Invocación Dinámica (DII):** Esta forma de acceso nos permitirá hacer llamadas a procedimientos de nuestro Servicio Web de forma dinámica, sin crear un *stub* para ello. Utilizaremos este tipo de invocación cuando no conozcamos la interfaz del servicio *a priori*, en cuyo caso para invocar a dicho servicio deberemos proporcionar únicamente los nombres de los métodos a utilizar

mediante una cadena de texto.

Podremos utilizar esta interfaz dinámica aunque no contemos con un documento WSDL que nos indique la interfaz y datos de nuestro servicio. En este caso, deberemos proporcionar manualmente esta información, para así poder acceder al servicio correctamente.

Vamos a ver cómo invocar servicios web JAX-WS (crear un cliente web) utilizando un *stub* estático con jdk 1.6, con Maven, y con Netbeans. Y después comentaremos cómo utilizar la Interfaz de Invocación Dinámica.

Cuando utilizamos un *stub* estático, los pasos generales para crear un cliente son:

1. Codificar la clase cliente, implementando el acceso al servicio Web utilizando las interfaces de los stubs que se generarán con *wsimport*
2. Generar los artefactos necesarios del servicio web para poder conectar con dicho servicio Web desde el cliente (mediante la utilidad *wsimport*)
3. Compilar la clase cliente (empaquetar y desplegar si es necesario)
4. Ejecutar el cliente

### 1.11. Invocación de servicios web JAX-WS con JDK 1.6

A partir de JDK 1.6 se incluye en Java SE la librería JAX-WS y las herramientas necesarias para crear e invocar servicios. Podemos encontrar las clases de la API de JAX-WS dentro del paquete `javax.xml.ws` y en subpaquetes de éste.

Para crear un cliente en JDK 1.6 (o con JAX-WS en versiones anteriores de JDK) utilizaremos la herramienta *wsimport*, que tomará como entrada el documento WSDL del servicio al que queremos acceder y producirá un conjunto de clases Java que nos permitirán acceder al servicio. Esta herramienta se puede invocar desde línea de comando:

```
wsimport -s <src.dir> -d <dest.dir> -p <pkg> <wsdl.uri>
```

El documento WSDL (`<wsdl.url>`) se especificará mediante su ruta en el disco o mediante su URL. Podemos proporcionar otros parámetros para indicar la forma en la que se debe generar el *stub*, como el directorio donde queremos que guarde los fuentes de las clases generadas (`<src.dir>`), el directorio donde guardará estas clases compiladas (`<dest.dir>`), y el paquete en el que se generará este conjunto de clases (`<pkg>`).

Por ejemplo podríamos utilizar el siguiente comando para crear el cliente de un servicio HolaMundo que se encuentra definido en `http://jtech.ua.es/HolaMundo/wsdl/HolaMundoSW.wsdl`, separando los fuentes en el directorio `src` y las clases compiladas en `bin`, y generando todas estas clases dentro de un paquete `es.ua.jtech.servcweb.hola.stub`:

```
wsimport -s src -d bin -p es.ua.jtech.servcweb.hola.stub  
http://jtech.ua.es/HolaMundo/wsdl/HolaMundoSW.wsdl
```

Con esto se generarán una serie de clases que nos permitirán acceder al servicio web e invocar sus operaciones desde nuestro cliente. Dentro de estas clases tendremos una que recibirá el mismo nombre que el servicio, y que heredará de la clase `Service`. Deberemos instanciar esta clase, y a partir de ella obtendremos el *stub* para acceder a un puerto del servicio. Este *stub* tendrá la misma interfaz que el servicio web, y a partir de él podremos invocar sus operaciones. En el ejemplo del servicio `HolaMundo` accederíamos al servicio de la siguiente forma:

```
HolaMundoSWService service = new HolaMundoSWService();
HolaMundoSW port = service.getHolaMundoSW();

System.out.println("Resultado: " + port.saluda("Miguel"));
```

El servicio ofrece una operación `saluda` que recibe un nombre, y genera un saludo incluyendo dicho nombre. En este ejemplo vemos como podemos invocar la operación `saluda` del servicio a partir del *stub* (`port`) como si se tratase de un método de un objeto local.

## 1.12. Invocación de servicios web JAX-WS desde una clase Java con Maven

Ahora vamos a ver paso a paso cómo crear nuestro cliente de un servicio web JAX-WS utilizando Maven. Para ello necesitamos conocer la dirección del WSDL del servicio web al que vamos a acceder. Para este ejemplo, supongamos que dicha dirección es: `http://localhost:8080/holaMundo/hola?WSDL`

El contenido del fichero `wsdl` es el siguiente:

```
<definitions
  xmlns:wsu=
    "http://docs.oasis-open.org/wss/2004/01/
      oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wsp="http://www.w3.org/ns/ws-policy"
    xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
    xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://sw/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="http://sw/"
    name="hola">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://sw/"
        schemaLocation="http://localhost:8080/HolaMundo/hola?xsd=1"/>
    </xsd:schema>
  </types>

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

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

```

<portType name="Hola">
  <operation name="hello">
    <input wsam:Action="http://sw/hola/helloRequest"
      message="tns:hello"/>
    <output wsam:Action="http://sw/hola/helloResponse"
      message="tns:helloResponse"/>
  </operation>
</portType>

<binding name="HolaPortBinding" type="tns:Hola">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document"/>
  <operation name="hello">
    <soap:operation soapAction=""/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>

<service name="hola">
  <port name="HolaPort" binding="tns:HolaPortBinding">
    <soap:address location="http://localhost:8080/HolaMundo/hola"/>
  </port>
</service>
</definitions>

```

Analizando la definición **ABSTRACTA** del servicio Web en el wsdl anterior, podemos ver que en la etiqueta `<portType>` (interfaz del servicio) se indica que el servicio ofrece la operación con nombre *hello*, que tiene asociado un mensaje de entrada de tipo *hello* y un mensaje de salida de tipo *helloResponse*. Los tipos de los mensajes de entrada y salida están definidos en un fichero de esquema importado en `<types>`, en el atributo *schemaLocation*, y cuyo valor podemos observar que es `http://localhost:8080/HolaMundo/hola?xsd=1` (en muchas ocasiones, y por comodidad, en lugar de importar el fichero de esquema, podemos incluir la definición de dicho esquema en el propio fichero wsdl). Si accedemos a dicha dirección podemos ver que el fichero de esquema contiene lo siguiente:

```

<xs:schema xmlns:tns="http://sw/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
    version="1.0" targetNamespace="http://sw/">
  <xs:element name="hello" type="tns:hello"/>
  <xs:element name="helloResponse" type="tns:helloResponse"/>

  <xs:complexType name="hello">
    <xs:sequence>
      <xs:element name="name" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="helloResponse">
    <xs:sequence>
      <xs:element name="return" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

```

```

</xs:sequence>
</xs:complexType>
</xs:schema>

```

En el fichero de esquema (con extensión .xsd), podemos observar que los mensajes *hello* y *helloResponse* contienen cadenas de caracteres. Por lo tanto, nuestro servicio acepta un mensaje con una cadena de caracteres como entrada y como resultado también nos devuelve otra cadena de caracteres.

Analizando la definición **CONCRETA** del servicio Web en el wsdl anterior, podemos ver en la etiqueta <port> se indica que la dirección para acceder al componente *Port* de nuestro servicio Web es: *http://localhost:8080/HolaMundo/hola*. El nombre del componente es **HolaPort**, y se deberá acceder al componente *HolaPort* a través de una instancia de la clase **Hola**.

Ahora vamos a comenzar a desarrollar nuestro cliente para el servicio web anterior. En primer lugar crearemos un proyecto java con maven, por ejemplo utilizando el arquetipo por defecto *maven-archetype-quickstart* mediante:

```

mvn archetype:generate -DgroupId=expertoJava
                        -DartifactId=HolaMundoJavaClient
                        -Dversion=1.0-SNAPSHOT
                        -DarchetypeArtifactId=maven-archetype-quickstart
                        -DinteractiveMode=false

```

En el *pom.xml* generado tenemos que incluir, dentro de la etiqueta <build> (y subetiqueta <plugins>), los siguientes *plugins*

- **jaxws-maven-plugin**: para ejecutar la utilidad **wsimport** de JAX-WS y generar los stubs del servicio web. Vamos a configurar la ejecución de la *goal wsimport* de forma que, a partir del wsdl situado en: *http://localhost:8080/HolaMundo/hola?WSDL*, genere los stubs necesarios, que por defecto se almacenan en el directorio *target/jaxws/wsimport/java*. Por defecto está asociado a la fase *generate-sources* del ciclo de vida de Maven. *Wsimport* se ejecutará, por tanto, antes de compilar los fuentes del proyecto.
- **exec-maven-plugin**: para ejecutar la aplicación java desde maven

El código para el plugin *jaxws-maven-plugin* quedaría como:

```

<plugin>
  <groupId>org.jvnet.jax-ws-commons</groupId>
  <artifactId>jaxws-maven-plugin</artifactId>
  <version>2.2</version>
  <executions>
    <execution>
      <goals>
        <goal>wsimport</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <!-- opcionalmente podríamos indicar el paquete en el que
         queremos que se generen los ficheros

```

```

    <packageName>wsClient</packageName>
    -->
    <wsdlUrls>
      <wsdlUrl>
        http://localhost:8080/HolaMundo/hola?WSDL
      </wsdlUrl>
    </wsdlUrls>
    <verbose>true</verbose>
  </configuration>

  <dependencies>
    <dependency>
      <groupId>javax.xml</groupId>
      <artifactId>webservicex-api</artifactId>
      <version>1.4</version>
    </dependency>
  </dependencies>
</plugin>

```

El código para el plugin *exec-maven-plugin* sería el siguiente:

```

<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <version>1.2.1</version>
  <executions>
    <execution>
      <goals>
        <goal>java</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <mainClass>expertoJava.App</mainClass>
  </configuration>
</plugin>

```

Adicionalmente necesitamos incluir en nuestro *pom.xml* la dependencia con la librería *webservicex-rt*

```

<dependency>
  <groupId>com.sun.xml.ws</groupId>
  <artifactId>webservicex-rt</artifactId>
  <version>1.4</version>
  <scope>compile</scope>
</dependency>

```

También nos aseguraremos de incluir la codificación del código fuente como una propiedad en nuestro fichero pom:

```

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

```

A continuación, compilaremos nuestro proyecto para generar los *stubs* necesarios con

*wsimport*, y poder realizar llamadas al servicio web.

```
mvn compile
```

Podemos ver en pantalla que, al compilar, se ejecuta *wsimport*, se parsea el WSDL, y a continuación se genera el código (stubs). También se muestran los nombres de los ficheros generados (ya que hemos incluido la etiqueta `<verbose>true</true>` en la configuración de *wsimport* en nuestro *pom*).

```
[INFO] Scanning for projects...
[INFO]
[INFO]
-----
[INFO] Building HolaMundoJavaClient 1.0-SNAPSHOT
[INFO]
-----
[INFO]
[INFO] --- jaxws-maven-plugin:1.10:wsimport (default) @
HolaMundoJavaClient ---
[INFO] Processing: http://localhost:8080/HolaMundo/hola?WSDL
[INFO] jaxws:wsimport args: [-s, ..., -d, ..., -verbose, -catalog,
(omitimos los detalles de la ejecución de wsimport)
parsing WSDL...

generating code...

sw/Hello.java
sw/HelloResponse.java
sw/Hola.java
sw/Hola_Service.java
sw/ObjectFactory.java
sw/package-info.java

compiling code...

javac -d ... (omitimos el comando de compilación)
[INFO]
[INFO] --- maven-resources-plugin:2.5:resources (default-resources) @
HolaMundoJavaClient ---
...
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory
/home/especialista/MavenProjects/
    HolaMundoJavaClient/src/main/resources
[INFO]
[INFO] --- maven-compiler-plugin:2.3.2:compile (default-compile) @
HolaMundoJavaClient ---
[INFO] Compiling 1 source files to
/home/especialista/MavenProjects/HolaMundoJavaClient/target/classes
[INFO]
-----
[INFO] BUILD SUCCESS
[INFO]
-----
...
```

Podemos ver que *wsimport* genera en **target/generated-sources/sw** los siguientes ficheros:

- **Hola.java**: contiene la Interfaz del servicio (SEI del servicio)
- **Hola\_Service.java**: contiene la Clase heredada de Service, que utilizaremos para

acceder al componente Port de nuestro servicio web a través del SEI. Para ello debemos utilizar el método "Hola getHolaPort()" de la clase Hola\_Service

- **ObjectFactory.java**: Contiene una factoria de métodos para recuperar representaciones java a partir de definiciones XML
- **Hello.java, HelloResponse.java**: son las clases que representan mensajes de nuestro WSDL
- **package-info.java**: fichero con información sobre las clases generadas

Ahora vamos a codificar la llamada al servicio web desde nuestra clase Java App:

```
package jtech;

public class App
{
    public static void main( String[] args )
    {
        try {
            //Primero accedemos a un objeto Service
            sw.Hola_Service service = new sw.Hola_Service();
            //a través de él accedemos al Port
            sw.Hola port = service.getHolaPort();
            java.lang.String name = "amigos de los Servicios Web";
            //utilizamos el Port para llamar al WS a través del SEI
            java.lang.String result = port.hello(name);
            System.out.println("Result = "+result);
        } catch (Exception ex) {
            // TODO handle custom exceptions here
        }
    }
}
```

Podemos comprobar en el código que nuestro cliente no es incapaz de distinguir si los métodos a los que está llamando se están ejecutando en local o en remoto, ni tampoco puede saber cómo está implementado el servicio (si es una clase java, un servlet, ejb, ...).

De igual forma, en el código anterior se aprecia que el cliente **NO TIENE CONTROL** sobre el ciclo de vida del servicio Web, de forma que **no puede crear o destruir instancias** de un servicio Web (componentes *Port*). El cliente **SÓLO ACCEDE** a un componente *Port*. El ciclo de vida de los *Ports* o instancias de la implementación de un Servicio web, son generadas por el *run-time* en el que reside dicho servicio Web (típicamente en un servidor de aplicaciones).

Una vez creada nuestra clase cliente, vamos a empaquetar:

```
mvn package
```

... y finalmente ejecutar nuestra aplicación java

```
mvn exec:java
```

El resultado debe ser:

```
Hola amigos de los Servicios Web !
```



Lógicamente, siempre y cuando el servicio esté desplegado en un servidor de aplicaciones que esté en marcha! :)

### 1.13. Invocación de servicios web JAX-WS desde una aplicación Web con Maven

Vamos a ver a cómo podemos invocar un servicio web JAX-WS desde un Servlet y/o JSP con Maven. Utilizaremos el mismo wsdl del ejemplo del apartado anterior.

En este caso comenzaremos por generar un proyecto Web java EE 6 con Maven:

```
mvn archetype:generate -DgroupId=jtech
                        -DartifactId=HolaMundoWebClient
                        -Dversion=1.0-SNAPSHOT
                        -DarchetypeArtifactId=webapp-javaee6
                        -DarchetypeGroupId=org.codehaus.mojo.archetypes
                        -DinteractiveMode=false
```

#### Importante

No olvides incluir el parámetro `-DarchetypeGroupId=org.codehaus.mojo.archetypes`, si no lo haces así, Maven no encontrará el arquetipo `webapp-javaee6` en el catálogo de arquetipos.

Vamos a comentar brevemente el contenido del fichero ***pom.xml*** que hemos generado. Tenemos las siguientes dependencias y *plugins*.

- librería **javaee-web-api (versión 6.0)**: necesaria para utilizar *servlets* con anotaciones
- plugin **maven-compiler (version 2.3.2)**: necesitamos configurar la versión de los fuentes y ejecutables java (1.6), y para compilar las librerías del directorio "endorsed". Dicho directorio se contendrá versiones actualizadas de librerías de anotaciones de servicios Web.
- plugin **maven-war (version 2.1.1)**: utilizado para empaquetar nuestro proyecto ignorando el fichero *web.xml* a la hora de hacer el *build* de nuestro proyecto
- plugin **maven-dependency (version 2.1)**: lo utilizaremos para copiar en el directorio "endorsed" (target/endorsed) la librería *javaee-endorsed-api-6.0.jar*. Esta librería permite utilizar anotaciones de servicios Web y se utilizará en la librería correspondiente de *jax-ws* que viene incluida por defecto en *jdk 6*

Para poder compilar y desplegar nuestro cliente de servicio Web en *glassfish*, vamos a añadir dos *plugins* más:

- plugin **jaxws-maven-plugin (version 2.2)**: con el que ejecutaremos la *goal wsimport*, igual que hicimos para el cliente Java del ejemplo anterior.
- plugin **glassfish (version 2.1)**: para desplegar el war generado en el servidor de aplicaciones, utilizando la *goal glassfish:deploy*. Para configurar el *plugin*, vamos a indicar la contraseña utilizando el fichero *master-password*, que contiene la contraseña codificada del administrador del dominio, y que está situado en el directorio raíz de dicho dominio (en nuestro caso *glassfish/domains/domain1*).

Por defecto, la contraseña del administrador del dominio es *changeit*, y esta contraseña NO se guarda (es decir, no se crea el *master-password*). Para crear dicho fichero tendremos que utilizar los siguientes comandos:

```
/opt/glassfish-3.2.2/bin/asadmin stop-domain
```

```
/opt/glassfish-3.2.2/bin/asadmin change-master-password
--savemasterpassword=true domain1
```

Este comando nos solicitará la contraseña actual y la nueva. Introducimos en ambos casos *changeit*. Como resultado se habrá creado el fichero `glassfish/domains/domain1/master-password`, que referenciaremos en nuestro `pom`

La configuración del **plugin de glassfish** quedaría como sigue:

```
<plugin>
  <groupId>org.glassfish.maven.plugin</groupId>
  <artifactId>maven-glassfish-plugin</artifactId>
  <version>2.1</version>
  <configuration>
    <user>admin</user>
    <passwordFile>
      /opt/glassfish-3.2.2/glassfish/domains/domain1/master-password
    </passwordFile>
  </configuration>
</plugin>
<glassfishDirectory>/opt/glassfish-3.2.2/glassfish</glassfishDirectory>
<domain>
  <name>domain1</name>
  <adminPort>4848</adminPort>
  <httpPort>8080</httpPort>
</domain>
<components>
  <component>
    <name>HolaMundoWebClient</name>
  </component>
</components>
<artifact>target/HolaMundoWebClient-1.0-SNAPSHOT.war</artifact>
</plugin>
```

Hemos configurado el despliegue de forma que el nombre del componente desplegado en Glassfish es *HolaMundoWebClient*. Dicho componente tendrá asociado el *war* de nuestra aplicación Web. Si después de desplegar la aplicación en Glassfish, ejecutamos el comando `asadmin list-applications`, veremos que uno de las aplicaciones desplegadas es *HolaMundoWebClient*.

#### Importante

Antes de realizar el despliegue de nuestra aplicación tenemos que asegurarnos de que en nuestro directorio HOME no tengamos el fichero `.asadminpass`. Si es así, procederemos a renombrarlo (por ejemplo lo podemos sustituir por `.asadminpass-old`).

Ahora vamos a implementar el servlet (por ejemplo lo podemos llamar *NewServlet.java*). Para ello utilizaremos la anotación `@WebServlet`. Especificaremos los atributos *name*,

con el nombre del servlet, y el atributo *urlPatterns*, en el que indicaremos la url con la que se accede a dicho servlet.

Una posible implementación puede ser la siguiente:

```
package expertoJava;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.xml.ws.WebServiceRef;
import sw.Hola_Service;

@WebServlet(name = "Hola", urlPatterns = {"/Hola"})
public class NewServlet extends HttpServlet {
    @WebServiceRef
    private Hola_Service service;

    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet HolaMundo</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet HolaMundo desde " +
request.getContextPath() + "</h1>");

            try { // Obtenemos el Port
                sw.Hola port = service.getHolaPort();
                java.lang.String name = "amigos de los Servicios Web";
                // Llamamos a la operación correspondiente del SEI
                java.lang.String result = port.hello(name);
                out.println("Result = "+result);
            } catch (Exception ex) {
                // Manejo de excepciones
            }

            out.println("</body>");
            out.println("</html>");
        } finally {
            out.close();
        }
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {
        processRequest(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
response)
```

```

        throws ServletException, IOException {
            processRequest(request, response);
        }

        @Override
        public String getServletInfo() {
            return "Short description";
        }
    }
}

```

Maven nos generará como raíz del contexto por defecto de nuestro servlet el nombre del artefacto generado, es decir, *HolaMundoWebClient-1.0-SNAPSHOT*. Si quisiéramos utilizar otro nombre, como por ejemplo *HolaMundoWebClient*, tendremos que añadir el fichero *glassfish-web.xml* (en el directorio *src/main/webapp/WEB-INF/*) con el siguiente contenido:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE glassfish-web-app PUBLIC
"-//GlassFish.org//DTD GlassFish Application Server 3.1 Servlet
3.0//EN"
"http://glassfish.org/dtds/glassfish-web-app_3_0-1.dtd">
<glassfish-web-app>
  <context-root>HolaMundoWebClient</context-root>
</glassfish-web-app>

```

Ahora ya estamos en disposición de compilar, empaquetar, desplegar y ejecutar nuestro cliente. Los stubs necesarios para realizar la llamada al servicio Web se generarán durante la compilación. A continuación empaquetaremos la aplicación, lo cual generará el war *target/HolaMundoWebClient-1.0-SNAPSHOT.war*.

```
mvn compile
```

```
mvn package
```

Antes de desplegar la aplicación nos aseguraremos de que el servidor de aplicaciones esté en marcha. Podemos arrancarlo desde línea de comandos con:

```
/opt/glassfish-3.1.2.2/bin/asadmin start-domain
```

Utilizando el comando *asadmin list-domains*, podemos ver qué dominios se están ejecutando. A continuación desplegamos nuestra aplicación.

El siguiente paso es desplegar dicho war en el servidor de aplicaciones Glassfish. Recordemos que el componente desplegado se llamará: *HolaMundoWebClient*.

```
mvn glassfish:deploy
```

Para comprobar que nuestra aplicación web se ha desplegado, podemos usar el comando *asadmin list-applications*.

Con el *plugin* de glassfish podemos también eliminar el componente desplegado de glassfish:

```
mvn glassfish:undeploy
```

O redespregar dicho componente (sería equivalente a hacer un *undeploy*, seguido de un *deploy*):

```
mvn glassfish:redeploy
```

Finalmente, podemos ver la ejecución de nuestro servlet, accediendo a la siguiente dirección, y suponiendo que hemos configurado la raíz del contexto como *HolaMundoWebClient* en el fichero *glassfish-web.xml*:

```
http://localhost:8080/HolaMundoWebClient/Hola
```

Podríamos querer pasar el parámetro de entrada directamente en la llamada desde el navegador, para lo que tendremos que incluir en nuestro código del *servlet* las siguientes líneas:

```
String cadena = request.getParameter("x");
...
out.println("<p>" + port.hello(cadena) + "%lt;/p>");
```

Ahora podemos pasar los parámetros de entrada al servlet de la siguiente forma:

```
http://localhost:8080/HolaMundoWebClient/Hola?x=pepe%20de%20Alicante
```

Los %20 son espacios en blanco. *x* es el nombre del parámetro de entrada.

Si en vez de un parámetro, nuestra aplicación requiere más de uno, se deben pasar las parejas "variable=valor" separadas por el carácter &.

### Ficheros de despliegue: web.xml

El uso de servlets con anotaciones hace que **NO** sea necesario proporcionar el fichero con la configuración del despliegue (fichero web.xml). A continuación mostramos el código de dicha configuración (**web.xml**), que puede incluirse en el directorio **src/main/webapp/WEB-INF**:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <servlet>
    <servlet-name>Hola</servlet-name>
    <servlet-class>expertoJava.NewServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Hola</servlet-name>
    <url-pattern>/Hola</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>
      30
    </session-timeout>
  </session-config>
</web-app>
```

## Uso de la anotación @WebServiceRef

Java EE 6 permite "inyectar" una referencia a un servicio Web remoto, a través del uso de la anotación **@WebServiceRef**. Utilizando esta anotación, el contenedor inyectará una instancia del WS (un objeto **Service**) en tiempo de ejecución. Ya hemos visto que, a partir del WSDL, es posible escribir un *proxy (stub)* java para invocar a un servicio Web (WS). La herramienta *wsimport* genera, a partir del WSDL, las clases e interfaces java que hacen posible la ejecución de una llamada a un WS por un cliente. Las interfaces generadas representan el SEI (*Service Endpoint Interface*), a través del cual podemos realizar llamadas locales al WS remoto vía HTTP.

El cliente puede obtener una instancia del *proxy* del WS utilizando la anotación **@WebServiceRef** de la siguiente forma:

```
...
@WebServiceRef
private MiServicio myService;
...
MiPuerto myPort=myService.getMiPuertoPort();
...
```

**MiServicio** es una clase que hereda de **javax.xml.ws.Service**, y es generada por la herramienta *wsimport*.

Si nuestro cliente NO reside en un contenedor (NO es gestionado por el contenedor), entonces NO soporta el uso de dicha anotación. Es decir, podemos hacer uso de **@WebServiceRef** cuando nuestro cliente sea, por ejemplo, un *servlet* o un *EJB*, pero no cuando sea una clase java plana. En este último caso tendremos que crear la instancia nosotros mismos (tendremos que hacer un **new**) del servicio directamente en nuestro código):

```
private MiServicio myService = new MiServicio();
...
MiPuerto myPort=myService.getMiPuertoPort();
...
```

La anotación **@WebServiceRef** puede soportar, entre otros, los siguientes parámetros:

- **name**: identifica la referencia al servicio Web. El valor de *name* es un nombre local a la aplicación que utiliza dicho recurso (WS). El nombre puede ser un nombre JNDI absoluto o relativo al espacio de nombres JNDI de java: *java:comp/env*. El valor por defecto es la cadena vacía
- **wsdlLocation**: identifica la URL del documento WSDL del servicio web referenciado. Por defecto contiene la cadena vacía. Si dicho valor está presente, "sobreescribe" (*overrides*) la localización del documento WSDL que se haya especificado en la anotación *WebService* de la clase del servicio referenciado que ha sido generada en el cliente (en nuestro caso la clase *Hola\_Service* generada por

- wsimport)
- **type:** identifica el tipo del recurso como un objeto de una clase Java. Por defecto su valor es *Object.class*
- **value:** identifica el tipo del recurso como un objeto de una clase Java. Por defecto su valor es *Service.class*

### Invocación del servicio Web desde una página JSP

Para ilustrar un ejemplo de una invocación al servicio Web desde una página jsp, simplemente tendremos que incluir la llamada a dicho WS desde `src/main/webapp/index.jsp`, por ejemplo, así:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <h1>Hola Mundo!</h1>
  <%
    try {
      sw.Hola_Service service = new sw.Hola_Service();
      sw.Hola port = service.getHolaPort();
      String name = "amigos de los Servicios Web";
      String result = port.hello(name);
      out.println("Result = "+result);
    } catch (Exception ex) {
      // TODO handle custom exceptions here
    }
  %>
  </body>
</html>
```

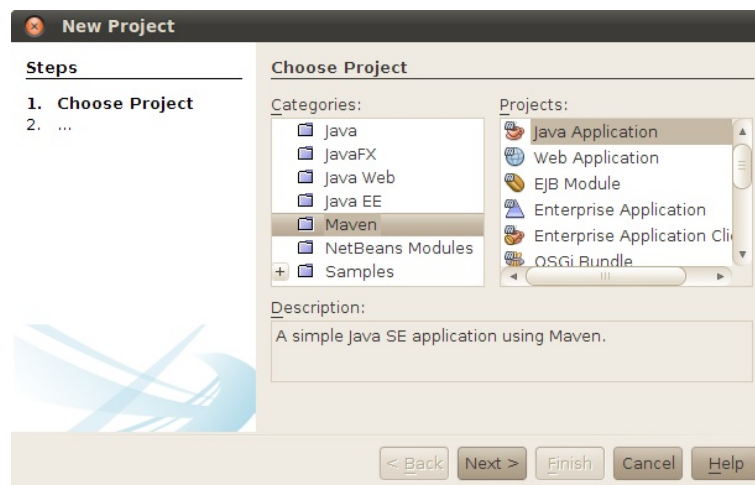
Una vez compilado y re-desplegado nuestro cliente Web, para comprobar que se ejecuta la invocación al servicio Web desde la página **jsp**, accederemos a la dirección:

```
http://localhost:8080/HolaMundoWebClient/
```

### 1.14. Invocación de servicios web con Netbeans

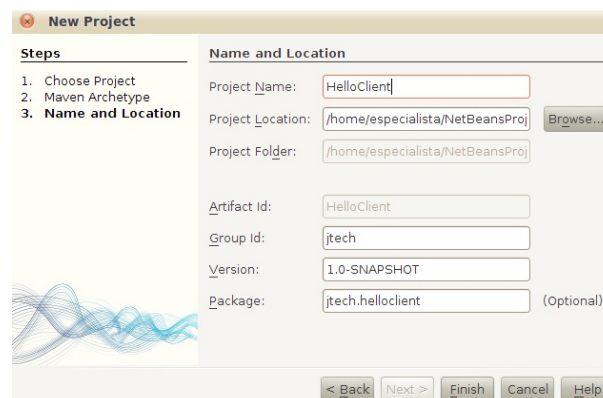
Vamos a ver ahora cómo crear un cliente de servicios web de forma visual mediante Netbeans. Este entorno utilizará internamente las librerías y herramientas estándar de Sun para crear *stub* del cliente.

En primer lugar, deberemos crear un proyecto, o utilizar uno ya existente. Este proyecto podrá ser de cualquier tipo (aplicación Java, aplicación Web, etc). Para nuestro ejemplo crearemos un proyecto Maven Java.



Creación de un cliente desde Netbeans

El proyecto se llamará `HelloClient` y tendrá una clase principal `App` en el paquete `jtech.helloclient`.

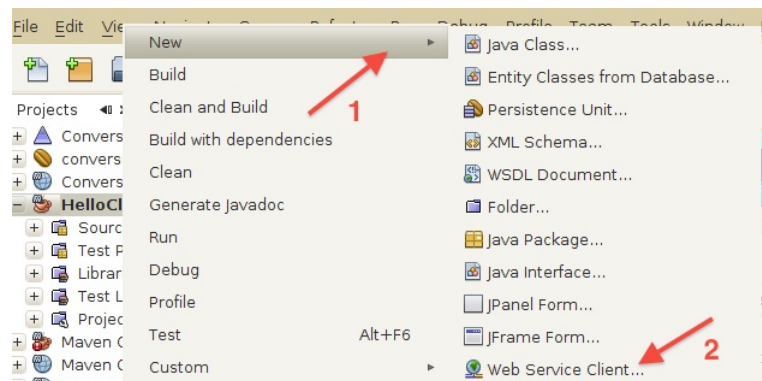


Datos de la creación del proyecto

Por defecto, la versión del código fuente es la 1.5. Podríamos cambiar a otra versión más actual (por ejemplo la 1.6) pulsando con botón derecho **Properties->Sources** y editando el campo **Source/Binary format**.

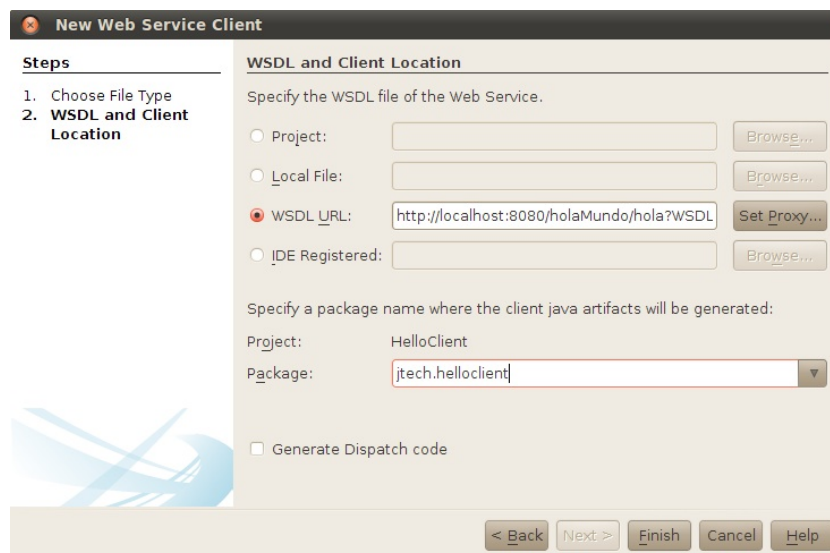
Una vez tengamos el proyecto java creado, podremos añadir el *stub* para acceder a un servicio web pulsando con el botón derecho sobre el proyecto y seleccionando la opción **New->Web Service Client ...**





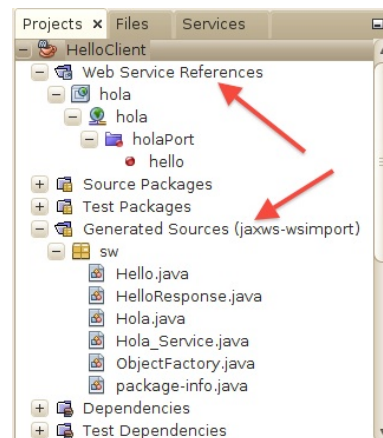
### Creación de un cliente de servicio Web

Se abrirá un asistente para crear el *stub* del cliente. Aquí deberemos especificar en primer lugar el documento WSDL que define el servicio al que vamos a acceder, por ejemplo indicando la URL en la que se encuentra. En segundo lugar especificaremos el paquete en el que queremos que se generen las clases del *stub*.



### Datos para la creación del stub en el cliente

Finalmente, una vez introducidos estos datos pulsaremos el botón *Finish*, tras lo cual se invocará a *wsimport* para generar el *stub* del servicio. Una vez generado el *stub* del cliente, podremos ver en la carpeta *Generated Sources (jaxws-wsimport)* del proyecto las clases generadas. En la carpeta *Web Service References* veremos la referencia al puerto y operaciones que proporciona el servicio Web referenciado.



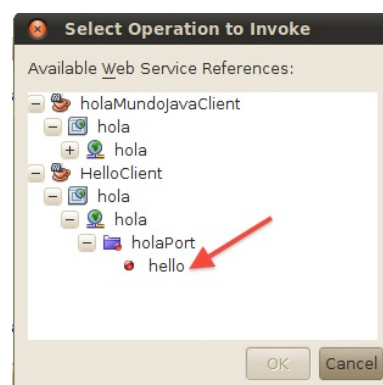
Ficheros generados por wsimport y referencias al WS

Una vez creado el *stub* del cliente, deberemos utilizarlo en nuestra aplicación para acceder al servicio. Vamos a suponer que lo queremos invocar desde la clase principal de nuestra aplicación (App), aunque lo mismo podría hacerse para invocarlo desde cualquier otra clase, o incluso desde un JSP. Para crear el código que llame al servicio pulsaremos con el botón derecho del ratón en el lugar del fuente de nuestra clase en el que queramos insertar la llamada (en nuestro ejemplo, en el fichero App.java dentro del método main), y con el botón derecho seleccionaremos la opción **Insert Code...**, y a continuación **Call Web Service Operation**.

#### Nota

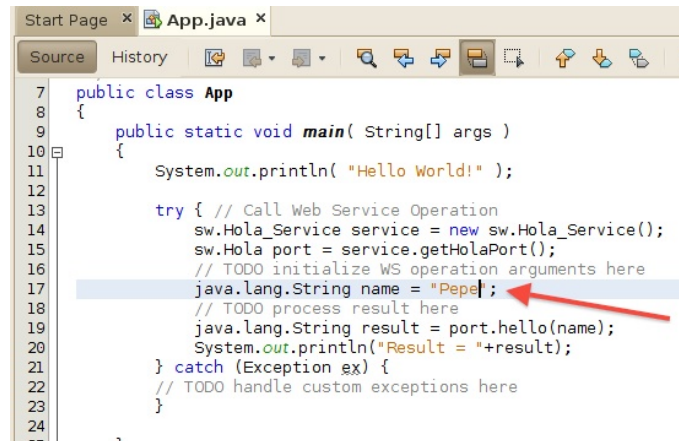
De forma alternativa podríamos añadir el código que invoca el servicio arrastrando el la operación *hola* de la carpeta *Web Service References*, hasta el lugar del código donde queremos realizar la llamada.

Se abrirá una ventana en la que deberemos seleccionar la operación que queramos invocar. En nuestro caso seleccionaremos la operación *hello* del servicio *hola* y pulsamos el botón **OK**



Selección de la operación del WS a invocar en el cliente

Una vez hecho esto, Netbeans generará en nuestra clase el código que hace la llamada al servicio. Debemos editar este código para especificar los parámetros que vamos a proporcionar en la llamada.



```

7 public class App
8 {
9     public static void main( String[] args )
10    {
11        System.out.println( "Hello World!" );
12
13        try { // Call Web Service Operation
14            sw.Hola_Service service = new sw.Hola_Service();
15            sw.Hola port = service.getHolaPort();
16            // TODO initialize WS operation arguments here
17            java.lang.String name = "Pepe";
18            // TODO process result here
19            java.lang.String result = port.hello(name);
20            System.out.println("Result = "+result);
21        } catch (Exception ex) {
22            // TODO handle custom exceptions here
23        }
24    }
25 }

```

Edición de la llamada al servicio web

Con esto ya podremos ejecutar nuestra aplicación cliente (con el botón derecho sobre nuestro proyecto, pulsando *Run* ), que se conectará al servicio web hola. El resultado se mostrará en el panel de salida de Netbeans.



```

Test Results  Output x  Tasks
Retriever Output x  HelloClient x
Compiling 1 source file to /home/especialista/NetBeansProjects/HelloClient/target/classes
[exec:exec]
Hello World!
Result = Hello Pepe !
BUILD SUCCESS

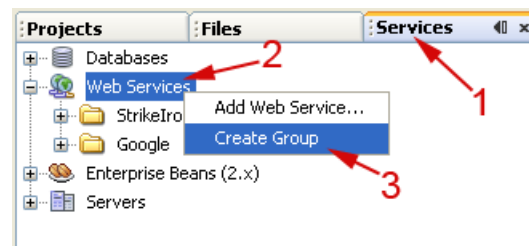
```

Ejecución del cliente java del servicio Web hola

## 1.15. Gestor de servicios web de Netbeans

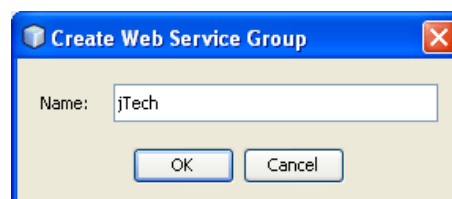
En Netbeans también encontramos un gestor de servicios web, en el que tenemos disponibles algunos servicios web externos proporcionados por terceros (Google, StrikeIron), y al que podemos añadir otros servicios, propios o ajenos. De esta forma podemos tener un catálogo de servicios que podremos incorporar de forma rápida a nuestras aplicaciones, e incluso probarlos desde el propio entorno.

Podemos acceder a este gestor de servicios a través de la sección *Web Services* de la ventana *Services* de Netbeans. Aquí veremos una lista de servicios que ya vienen incluidos en Netbeans y que podremos probar o incorporar en nuestras aplicaciones. Vamos a añadir un nuevo servicio. Para ello antes creamos un grupo en el que incluirlo. Pulsamos con el botón derecho sobre *Web Services* y seleccionamos *Create Group* para crear el nuevo grupo.



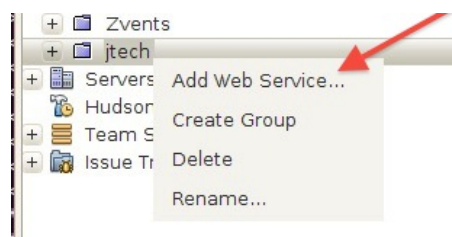
Gestor de servicios web de Netbeans

Vamos a añadir un servicio *HelloService*. Vamos a crear un grupo de nombre *jTech* en el que incluiremos los servicios proporcionados por dicha entidad.



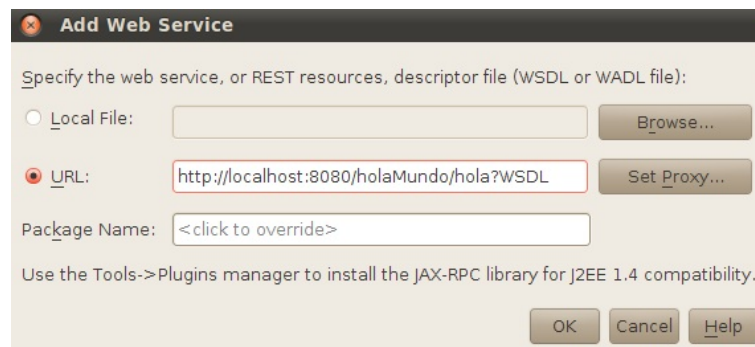
Creación de un nuevo grupo de servicios

Pulsamos sobre el grupo con el botón derecho y seleccionamos *Add Web Service ...* para añadir el nuevo servicio.



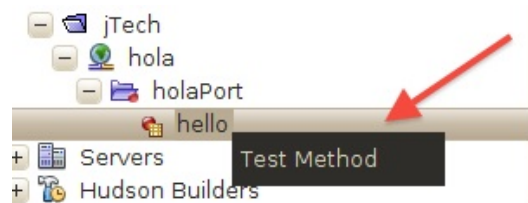
Añadimos un nuevo servicio en el grupo creado

Como datos del servicio proporcionamos la dirección del documento WSDL que lo define. Además podemos también especificar el paquete en el cual generará la librería con el *stub* necesario para acceder al servicio. Cuando terminemos de introducir los datos pulsamos *Add* para añadir el nuevo servicio.



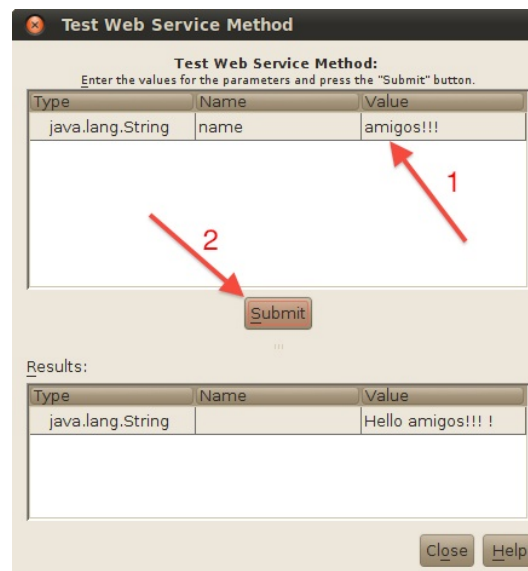
### Datos del servicio web para el gestor de servicios

Una vez tenemos el servicio añadido, podemos desplegarlo para ver las operaciones que ofrece y probarlas pulsando sobre ellas con el botón derecho y seleccionando *Test Method*



### Prueba de un servicio desde el gestor de servicios

Nos aparecerá una ventana en la que deberemos introducir los parámetros necesarios para invocar el servicio, y tras ello pulsar sobre el botón *Submit* para invocarlo y ver el resultado obtenido.



### Datos para la prueba del servicio y resultado obtenido

También podemos utilizar los servicios del gestor en nuestras aplicaciones de forma sencilla. Simplemente tendremos que arrastrar la operación que queramos invocar sobre el lugar del código fuente en el que queramos hacer la llamada. Al hacer esto se añadirá a nuestro proyecto una librería (JAR) con el *stub* necesario para acceder al servicio.

## 1.16. Interfaz de invocación dinámica (DII)

Mediante esta interfaz ya no utilizaremos un *stub* para invocar los métodos del servicio, sino que nos permitirá invocar los métodos de forma dinámica, indicando simplemente el nombre del método que queremos invocar como una cadena de texto, y sus parámetros

como un *array* de objetos.

Esto nos permitirá utilizar servicios que no conocemos previamente. De esta forma podremos implementar por ejemplo un *broker* de servicios. Un *broker* es un servicio intermediario, al que podemos solicitar alguna tarea que necesitemos. Entonces el *broker* intentará localizar el servicio más apropiado para dicha tarea en un registro de servicios, y lo invocará por nosotros. Una vez haya conseguido la información que requerimos, nos la devolverá. De esta forma la localización de servicios se hace totalmente transparente para nosotros.

Podremos acceder con esta interfaz tanto si contamos con un documento WSDL como si no contamos con él, pero en el caso de que no tengamos el WSDL deberemos especificar en el código todos los datos incluidos en estos documentos que necesitemos y de los que en este caso no disponemos (*endpoint*, parámetros y tipos, etc).

### 1.16.1. A partir de un documento WSDL

Vamos a ver el caso en el que contamos con el documento WSDL que describe el servicio. El primer paso será conseguir el objeto `Service` igual que hicimos en el caso anterior. La clase `javax.xml.ws.Service` es una abstracción que representa un servicio WSDL (etiqueta *service*).

La clase `Service` actúa como una factoría que proporciona tanto *proxies* para el *endpoint* de un servicio (de forma estática utilizando los stubs generados), como instancias para llamadas a operaciones remotas de forma dinámica.

Un ejemplo de uso dinámico de `Service` es el siguiente:

```
ServiceFactory sf = ServiceFactory.newInstance();
Service serv = sf.createService(
    new URL("http://localhost:8080/HolaMundo/hola?WSDL"),
    new QName("http://localhost:8080/HolaMundo", "hello"));
```

Utilizaremos el objeto `Call` para hacer las llamadas dinámicas a los métodos del servicio. Deberemos crear un objeto `Call` correspondiente a un determinado puerto y operación de nuestro servicio:

```
Call call = serv.createCall(
    new QName("http://localhost:8080/HolaMundo", "holaPort"),
    new QName("http://localhost:8080/HolaMundo", "hello"));
```

El último paso será invocar la llamada que hemos creado:

```
String result = (String) call.invoke(
    new Object[] { "Miguel" });
```

A este método le debemos proporcionar un *array* de objetos como parámetro, ya que debe poder utilizarse para cualquier operación, con diferente número y tipo de parámetros.

Como tampoco se conoce *a priori* el valor devuelto por la llamada, deberemos hacer una conversión *cast* al tipo que corresponda, ya que nos devuelve un `Object` genérico.

### 1.16.2. Sin un documento WSDL

Si no contamos con el WSDL del servicio, crearemos un objeto `Service` proporcionando únicamente el nombre del servicio:

```
ServiceFactory sf = ServiceFactory.newInstance();
Service serv = sf.createService(
    new QName("http://jtech.ua.es", "hola"));
```

A partir de este objeto podremos obtener un objeto `Call` para realizar una llamada al servicio de la misma forma que vimos en el caso anterior:

```
Call call = serv.createCall(
    new QName("http://localhost:8080/HolaMundo", "holaPort"),
    new QName("http://localhost:8080/HolaMundo", "hello"));
```

En este caso el objeto `Call` no tendrá ninguna información sobre las características del servicio, ya que no tiene acceso al documento WSDL que lo describe, por lo que deberemos proporcionárselas nosotros explícitamente.

En primer lugar, deberemos especificar el *endpoint* del servicio, para que sepa a qué dirección debe conectarse para acceder a dicho servicio:

```
call.setTargetEndpointAddress(endpoint);
```

Una vez especificada esta información, deberemos indicar el tipo de datos que nos devuelve la llamada a la operación que vamos a invocar (en nuestro ejemplo `saluda`):

```
QName t_string =
    new QName("http://www.w3.org/2001/XMLSchema", "string");
call.setReturnType(t_string);
```

Por último, indicaremos los parámetros de entrada que toma la operación y sus tipos:

```
QName t_string =
    new QName("http://www.w3.org/2001/XMLSchema", "string");
call.addParameter("string_1", t_string, ParameterMode.IN);
```

Una vez hecho esto, podremos invocar dicha operación igual que en el caso anterior:

```
String result = (String) call.invoke(
    new Object[] { "Miguel" });
```



## 2. Ejercicios. Invocación de Servicios Web SOAP

### 2.1. Repositorio Mercurial para los ejercicios

Todos los proyectos para los ejercicios de este módulo los crearemos, por ejemplo, en el directorio `NetBeansProjects/servc-web-expertojava`.

En Bitbucket debemos crear en nuestra cuenta el repositorio **`servc-web-expertojava`**. En Netbeans lo configuraremos como repositorio *push* por defecto, pulsando el botón derecho sobre cualquier proyecto y seleccionando la opción *Mercurial > Properties >... e introduciendo sus datos (tal y como ya habéis hecho para los ejercicios del módulo de componentes enterprise)*.

Cuando inicializamos un repositorio de Mercurial local desde Netbeans, éste crea el fichero `.hgignore`. Dado que los dos primeros ejercicios los vamos a hacer sin utilizar Netbeans, podemos inicializar el repositorio de Mercurial local creando un proyecto maven java en la carpeta `servc-web-experto-java`. Después seleccionamos dicho proyecto e inicializamos el repositorio local con la opción *Team > Mercurial > Initialize Project* indicando el directorio `NetBeansProjects/servc-web-expertojava`.

Con esto se habrá creado el fichero `.hgignore` en el directorio `NetBeansProjects/servc-web-expertojava`. Tal y como vimos en los ejercicios de la primera sesión del módulo de componentes enterprise, tendremos que ejecutar los comandos `hg add .hgignore` y `hg commit -m "Añadido .hgignore"`, desde el directorio `NetBeansProjects/servc-web-expertojava` para incluir añadir dicho fichero en el repositorio de Bitbucket.

Para subir los proyectos de los ejercicios que no requieren Netbeans a Bitbucket, podemos abrirlos desde Netbeans y utilizar las opciones *Mercurial > Commit* (desde el menú contextual de los proyectos), y posteriormente haremos *Mercurial > Share > Push to default*.

### 2.2. Clientes para servicio web hola

En las plantillas, disponéis de un war con la implementación del servicio web *hola*. Despliega dicho fichero: `HolaMundo.war` en el servidor de aplicaciones Glassfish para poder realizar el ejercicio. Puedes hacerlo desde línea de comandos, primero arranca el servidor con `/opt/glassfish-3.1.2.2/bin/asadmin start-domain domain1`, y después despliega el fichero con `/opt/glassfish-3.1.2.2/bin/asadmin/deploy HolaMundo.war`. En la consola de administración de glassfish (<http://localhost:4848/>) verás que se ha añadido la aplicación *HolaMundo* a la lista de aplicaciones ya desplegadas (también podrías comprobarlo desde línea de comandos con `/opt/glassfish-3.1.2.2/bin/asadmin list-applications`). En el nodo de



Aplicaciones verás el componente *HolaMundo*, pincha en *Ver punto final*. Aparecerá la dirección en dónde se ha generado el fichero wsdl asociado al servicio (dirección */HolaMundo/hola?wsdl*). También puedes ver que glassfish ha generado un *driver* de pruebas en */HolaMundo/hola?Tester*. Ejecuta el test para probar el método que proporciona el servicio.

Implementa un cliente, con nombre *HolaMundoJavaClient*, desde una clase Java (utiliza Maven desde línea de comandos) para el servicio hola a partir de su descripción wsdl (ver apartado *Invocación de servicios web JAX-WS desde una clase Java con Maven*). (0,5 puntos)

Cuando crees el arquetipo de Maven, utiliza los siguientes nombres para:

- groupId: expertoJava
- artifactId: HolaMundoJavaClient

Implementa un cliente, desde un *jsp* (utiliza Maven desde línea de comandos) para el servicio hola a partir de su descripción wsdl (ver apartado *Invocación de servicios web JAX-WS desde una aplicación Web con Maven*, subapartado *Invocación del servicio web desde una página JSP*). (0,5 puntos)

Cuando crees el arquetipo de Maven, utiliza los siguientes nombres para:

- groupId: expertoJava
- artifactId: HolaMundoWebClient

## 2.3. Cliente para el servicio web Calculadora

En las plantillas, disponéis de un war con la implementación del servicio web *Calculadora*. Despliega dicho fichero: *Calculadora.war* en el servidor de aplicaciones Glassfish para poder realizar el ejercicio. Accediendo a la consola de administración de Glassfish verás que se ha añadido la aplicación *Calculadora* a la lista de aplicaciones ya desplegadas. Verás el componente *Calculadora* de tipo *Servlet*, pincha en *Ver punto final*. Aparecerá la dirección en dónde se ha generado el fichero wsdl asociado al servicio (dirección */Calculadora/calculadora?wsdl*). Fíjate que, en este caso, en el wsdl se declara el mensaje <fault message> con el nombre *NegativeNumberException*. También puedes ver que Glassfish ha generado un *driver* de pruebas en */Calculadora/calculadora?Tester*. Ejecuta el test para probar el método que proporciona el servicio.

### Nota

El cliente de prueba no trata correctamente la recepción de un SOAP Fault como respuesta, ya que la excepción que genera éste hace que falle la misma aplicación de pruebas. Para poder tratar la excepción de forma correcta deberemos hacerlo con un cliente propio.

Implementa un cliente, desde un *servlet* utilizando Maven para el servicio Calculadora a partir de su descripción wsdl(ver apartado *Invocación de servicios web JAX-WS desde*

una aplicación Web con Maven). (1 punto)

Cuando crees el arquetipo de Maven, utiliza los siguientes nombres para:

- groupId: expertoJava
- artifactId: CalculadoraWebClient

El nombre del componente para el plugin de Glassfish será: *CalculadoraClient*. Configurar la raíz del contexto como *CalculadoraWeb* en el fichero *glassfish-web.xml*.

La url del servlet indicada en el atributo *url-patterns* de la anotación *@WebServlet* será: *calculadoraWebCliente*.

A continuación mostramos ejemplos de llamadas al servicio y el resultado por pantalla correspondiente:

**Caso 1.** Introducimos los parámetros en la llamada http:

```
http://localhost:8080/CalculadoraWeb/calculadoraWebCliente?x=10&y=20
```

El resultado en pantalla será:

```
El resultado de la suma es: 30
El primer parámetro es: 10
El segundo parámetro es: 20
```

**Caso 2.** No pasamos los parámetros en la llamada http:

```
http://localhost:8080/CalculadoraWeb/calculadoraWebCliente
```

El resultado en pantalla será:

```
Por favor, indique los números a sumar
```

**Caso 3.** Alguno de los parámetros es negativo:

```
http://localhost:8080/CalculadoraWeb/calculadoraWebCliente?x=10&y=-3
```

El resultado en pantalla será:

```
El segundo sumando es negativo. Solo puedo sumar números positivos
```

## 2.4. Cliente para el servicio web Parte Metereológico

En las plantillas, disponéis de un war con la implementación del servicio web *Parte Metereologico*. Despliega dicho fichero: *ParteMetereologico.war* en el servidor de aplicaciones Glassfish para poder realizar el ejercicio.

Analizando su wsdl descubre qué dos operaciones ofrece el servicio

Registra el servicio en el gestor de servicios de Netbeans con el nombre de grupo "expertoJava", prueba el servicio e implementa un cliente java Maven desde Netbeans para dicho servicio (el nombre del proyecto será Meteorologia-client).

Dicho cliente java debe realizar una llamada a las dos operaciones del servicio y mostrar en pantalla los resultados obtenidos. (1 punto)

Cuando crees el arquetipo de Maven, utiliza los siguientes nombres para:

- groupId: expertoJava
- artifactId: Meteorologia-Client

En este enlace podéis consultar un ejemplo de cómo de utilizar la clase *XMLGregorianCalendar*:

<http://nachxs.wordpress.com/2010/04/26/como-usa-xmlgregorianCalendar-en-java/>

### 3. Creación de Servicios Web SOAP

Vamos a crear nuestros propios Servicios Web, que ofrecerán una serie de métodos a los que se podrá llamar mediante RPC desde cualquier lugar de Internet mediante protocolos estándar (mensajes SOAP).

Deberemos por lo tanto ser capaces de interpretar en nuestras aplicaciones los mensajes SOAP entrantes de petición para la invocación de un método. Posteriormente, invocaremos el método solicitado, y con el resultado que nos devuelva deberemos construir un mensaje SOAP de respuesta y devolvérselo al cliente.

Si tuviésemos que introducir nosotros el código para interpretar este mensaje de entrada, y generar manualmente el mensaje de respuesta, el desarrollo de Servicios Web sería una tarea altamente costosa.

Es más, si se forzase al programador a componer el mensaje SOAP manualmente cada vez que desarrolle un Servicio Web, es muy probable que cometa algún error y no respete exactamente el estándar SOAP. Esto sería un grave problema para la interoperabilidad de los Servicios Web, que es una de las características que perseguimos con esta tecnología.

Para evitar estos problemas, utilizaremos librerías que nos permitan leer o generar mensajes SOAP para la invocación de métodos remotos, como es el caso de la API JAX-WS.

Además, para facilitar aún más la tarea de desarrollar Servicios Web, normalmente contaremos con herramientas que a partir de las clases que implementan nuestro servicio generen automáticamente todo el código necesario para leer el mensaje SOAP de entrada, invocar el método, escribir el mensaje SOAP de salida, y devolverlo al cliente.

Por lo tanto, nosotros deberemos centrarnos únicamente en la tarea de programar la funcionalidad que implementan nuestros servicios, olvidándonos del mecanismo de invocación de éstos.

JAX-WS es una especificación estándar de Sun Microsystems, pero no todos los servidores de aplicaciones utilizan esta librería para gestionar los Servicios Web. Por ejemplo, es el caso de Weblogic, que aunque está basado en JAX-WS, mantiene algunas extensiones propietarias sobre dicha API. Nos centraremos por lo tanto en el desarrollo de servicios con Netbeans y Glassfish, que incorpora las últimas versiones de las librerías estándar.

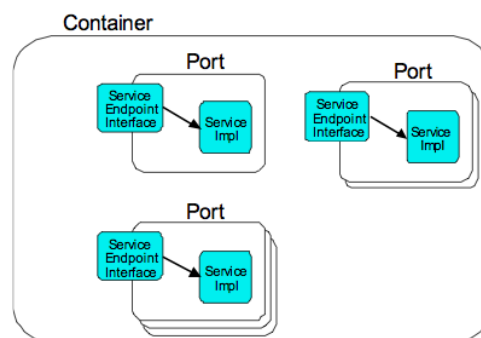
#### 3.1. Los servicios Web desde la vista del Servidor

Como ya hemos visto, un documento WSDL define la interoperabilidad de los servicios Web e incluye la especificación sobre requerimientos de transporte y formato de los datos a través de la red. En general, un WSDL no impone ningún requerimiento sobre el

modelo de programación del cliente o del servidor. La especificación de servicios Web para Java EE (JSR-109) define tres formas de implementar la lógica de negocio de servicio Web:

- Como un Bean de Sesión sin estado: la implementación del servicio Web (componente **Port**) se realiza creando un Bean de sesión sin estado, que implementa los métodos del SEI (*Service Endpoint Interface*) tal y como se describe en la especificación de EJB 3.0
- Como una clase Java: en este caso se implementa el *Port* como un *Servlet* JAX-WS
- Como un *Singleton Session Bean*: en este caso se crea un *singleton session bean* que implementa los métodos de un SEI según la especificación de EJB 3.1

Un componente **Port** define la vista del servidor de un Servicio Web. Cada *Port* proporciona un servicio en una dirección física particular definida por el atributo *address* de la definición `<port>` de un WSDL. Un componente Port "sirve" una petición de operación definida en un `<portType>` de un WSDL. La implementación del servicio (*Service Implementation Bean*) depende del contenedor del componente Port, pero en general es una clase Java que puede implementar los métodos definidos en el SEI (*Service Endpoint Interface*). El SEI es un mapeado java del `<portType>` y `<binding>` asociado a un `<port>` de un WSDL. Un servicio Web es un conjunto de *Ports* que difieren únicamente en su dirección física, y son mapeados en componentes Port separados, cada uno con su potencialmente único, pero probablemente compartido, Service Implementation Bean. La siguiente figura muestra la vista del servidor de un Servicio Web.



Vista del servidor de un servicio Web

El ciclo de vida del Port está completamente controlado por el contenedor, pero en general sigue el mismo ciclo de vida que el del propio contenedor. Un Port es creado e inicializado por el contenedor antes de que la primera llamada recibida en la dirección del `<port>` del WSDL sea servida. Un Port es destruido por el contenedor cuando éste considera que sea necesario hacerlo, como por ejemplo cuando el propio contenedor es  *shutting down*. Una implementación de un servicio Web JAX-WS reside en un contenedor Web, y por lo tanto puede desplegarse en un servidor Web o un servidor de aplicaciones, una implementación EJB, en cambio, reside en un contenedor EJB y sólo podrá

desplegarse en un servidor de aplicaciones.

Un componente **Port** asocia una dirección de puerto (*port address* de un WSDL) con la implementación del servicio (*Service Implementation Bean*). En general, el componente **Port** "pospone" la definición de los requerimientos del contenedor del servicio a la fase de despliegue, indicando dichos requerimientos en el descriptor de despliegue del componente. Un contenedor proporciona un *listener* en la dirección del puerto (*port address* de un WSDL) y un mecanismo para "enviar" la petición a la implementación del servicio Web. Un contenedor también proporciona servicios en tiempo de ejecución, tales como restricciones de seguridad y mapeados de referencias lógicas a referencias físicas de objetos distribuidos y recursos.

Un desarrollador declara un componente **Port** en un descriptor de despliegue de servicios Web. El descriptor de despliegue incluye el documento WSDL que describe el *PortType* y *binding* del servicio Web. Cuando se usa JAX-WS, no es necesario proporcionar dicho descriptor de despliegue. La mayor parte de la información del descriptor de despliegue se incluye en las anotaciones de la implementación del servicio. Podríamos utilizar el descriptor de despliegue para "sobreescribir" o mejorar la información proporcionada por las anotaciones de la implementación del servicio.

La plataforma Java EE 6, soporta las siguientes implementaciones de servicios Web: como un componente Web JAX-WS en un contenedor de *Servlets*, y como un componente EJB de sesión *stateless* o *singleton*.

El empaquetado de un servicio Web en un módulo Java EE es específico de la metodología de implementación, pero sigue los requerimientos para un fichero EJB-JAR o fichero WAR. Contiene los ficheros de clases java del SEI y los documentos WSDL del servicio Web. Además contiene el descriptor XML de despliegue que define los *Ports* del servicio y su estructura.

### 3.2. El modelo de programación JAX-WS

Para desarrollar una implementación de un servicio web (*web service endpoint*) podemos optar por dos puntos de partida: una clase Java que implementa el servicio Web o un fichero WSDL. Cuando comenzamos por una **clase java**, utilizaremos herramientas para generar los artefactos necesarios, entre ellos el WSDL del servicio. Cuando nuestro punto de partida es un **fichero WSDL** (junto con los ficheros de esquema que describen los tipos de datos usados en el servicio), utilizaremos herramientas para generar el SEI (*Service Endpoint Interface*)

#### Sobre JAX-WS y la implementación del servicio

JAX-WS impone la existencia de un *Service Implementation Bean* anotado con *javax.jws.WebService* en un componente **Port**. Como ya hemos indicado en el apartado anterior, la implementación del servicio (*Service Implementation Bean*) va a depender del contenedor del componente **Port**, pero en general es una clase Java que puede implementar los métodos definidos en el SEI (*Service Endpoint Interface*). En los apartados siguientes veremos que

JAX-WS permite implementar un *Service Implementation Bean* como una clase Java anotada (modelo de servlets), o como un EJB de sesión sin estado (modelo EJB).

Si comenzamos por una clase java, tendremos la seguridad de que la clase que implementa el servicio tiene los tipos de datos java adecuados, pero el desarrollador tienen menos control sobre el esquema XML generado. Si comenzamos por el WSDL y esquemas, el desarrollador tiene un control total sobre qué esquema se está usando, pero menos control sobre el *endpoint* del servicio generado y de las clases que utiliza.

Nosotros vamos a explicar en primer lugar cómo crear un servicio a partir de una clase java, y también veremos cómo utilizar Netbeans para crear un servicio a partir de un wsdl.

Cuando nuestro punto de partida es una clase java, tenemos que seguir ciertas restricciones en la implementación de nuestro servicio. Una implementación válida de un servicio web es una clase java que cumple las siguientes restricciones:

- La clase debe estar anotada con `javax.jws.WebService` (o alternativamente con `javax.xml.ws.Provider`, si se desea trabajar directamente a nivel de mensajes XML)
- Podemos anotar cualquiera de sus métodos con `javax.jws.WebMethod`
- Todos sus métodos pueden lanzar la excepción `java.rmi.RemoteException` además de cualquier otra excepción propia del servicio
- Los parámetros de sus métodos y tipos de retorno deben ser compatible con JAXB (JAXB impone unas reglas para mapear tipos java con tipos de ficheros de esquema XML)
- Ningún parámetro y/o tipo de retorno pueden implementar la interfaz `java.rmi.Remote` ni directa ni indirectamente

La clase java anotada con `@WebService` define un SEI de forma **implícita** por lo que no será necesario proporcionar dicha interfaz. Podemos especificar de forma explícita una interfaz añadiendo el atributo `endpointInterface` a la anotación `@WebService`. En ese caso, sí es necesario proporcionar la interfaz que defina los métodos públicos disponibles en la clase que implementa el servicio.

Una implementación de un servicio Web que utiliza la anotación `@WebService` no es necesario que especifique la ubicación del WSDL. Si se utiliza el atributo `wsdlLocation` en la anotación `@WebService`, el fichero WSDL debe ser empaquetado junto con las clases java del servicio web.

A continuación explicaremos con más detalle cómo implementar el servicio utilizando el modelo de servlets (el servicio se ejecuta en un contenedor web) y ejb (el servicio se ejecuta en un contenedor EJB).

### 3.3. Implementación del servicio JAX-WS con el modelo de servlets

Un *Service Implementation Bean* que se ejecuta dentro de un contenedor Web debe seguir los siguientes requerimientos (algunos de los cuales ya los hemos indicado en el apartado anterior):

- La clase que implementa el servicio debe estar anotada con `javax.jws.WebService` (o alternativamente, con `javax.xml.ws.Provider`)
- Si se implementa el servicio Web partiendo de código java, la anotación `javax.jws.WebService` NO es necesario especificar el SEI, la clase que implementa el servicio implícitamente define un SEI. Los métodos de negocio de la implementación deben ser **públicos**, y NO pueden ser **final** o **static**. Solamente los métodos anotados con `@WebMethod` en la implementación del servicio son expuestos al cliente.
- Si se implementa el servicio Web a partir del WSDL, el SEI generado a partir del WSDL debe estar anotado con `javax.jws.WebService`. La implementación del servicio debe estar también anotada con `javax.jws.WebService`. La implementación del servicio puede implementar el SEI, aunque no es necesario. En este caso se deben implementar todos los métodos indicados en el SEI. Los métodos de negocio de la implementación deben ser **públicos**, y NO pueden ser **final** o **static**. Se puede implementar métodos adicionales que no figuren en el SEI.
- La implementación del servicio debe tener un constructor público por defecto
- La implementación del servicio no debe guardar el estado
- La clase debe ser pública, no puede ser *final* ni *abstracta*
- La clase no debe implementar el método `finalize()`

Por ejemplo, podemos implementar nuestro servicio Web como:

```
package jaxwsHelloServer;

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

@WebService
public class Hello {
    private String message = new String("Hola, ");

    public void Hello() {}

    @WebMethod
    public String sayHello(String name) {
        return message + name + ".";
    }
}
```

Con esto habremos implementado la funcionalidad del servicio como una clase Java ordinaria, sin necesitar tener conocimientos de ninguna librería adicional.

De forma opcional, podemos añadir al servicio un campo `context` en el que se inyectará un objeto `WebServiceContext` que nos dará acceso al contexto del servicio:

```
...
@WebService
public class Hello {
    @Resource
```



```
private WebServiceContext context;

...

}
```

Dado que realmente el servicio es un componente web, a través de este objeto podremos tener acceso a componentes de la API de *servlets* como la petición HTTP (`HttpServletRequest`), la sesión (`HttpSession`), etc.

### 3.3.1. Anotaciones

Podemos especificar la forma en la que se crea el servicio mediante diferentes anotaciones. Las principales anotaciones disponibles son:

@WebService	<p>Indica que la clase define un servicio web. Se pueden especificar como parámetros los nombres del servicio (<code>serviceName</code>), del componente Port (<code>portName</code>), del SEI del servicio (<code>name</code>), de su espacio de nombres (<code>targetNamespace</code>), y de la ubicación del WSDL (<code>wsdlLocation</code>), que figurarán en el documento WSDL del servicio:</p> <pre>@WebService(name="ConversionPortType",     serviceName="ConversionService",     portName="ConversionPort",     targetNamespace="http://jtech.ua.es",     wsdlLocation="resources/wsdl/")</pre>
@SOAPBinding	<p>Permite especificar el estilo y la codificación de los mensajes SOAP utilizados para invocar el servicio. Por ejemplo:</p> <pre>@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,     use=SOAPBinding.Use.LITERAL,     parameterStyle=         SOAPBinding.ParameterStyle.WRAPPED)</pre>
@WebMethod	<p>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:</p> <pre>@WebMethod(operationName="eurosAptas") public int euro2ptas(double euros) {     ... }</pre>
@Oneway	<p>Indica que la llamada a la operación no debe esperar ninguna respuesta. Esto sólo lo podremos hacer con métodos que devuelvan <code>void</code>. Por ejemplo:</p> <pre>@Oneway() @WebMethod() public void publicarMensaje(String mensaje) {     ... }</pre>

	<pre>} // ... }</pre>
@WebParam	<p>Permite indicar el nombre que recibirán los parámetros en el fichero WSDL:</p> <pre>@WebMethod(operationName="eurosAptas") public int euro2ptas(     @WebParam(name="CantidadEuros",         targetNamespace="http://jtech.ua.es")     double euros) {     ... }</pre>
@WebResult	<p>Permite indicar el nombre que recibirá el mensaje de respuesta en el fichero WSDL:</p> <pre>@WebMethod(operationName="eurosAptas") @WebResult(name="ResultadoPtas",     targetNamespace="http://jtech.ua.es") public int euro2ptas(double euros) {     ... }</pre>
@WebFault	<p>Se utiliza para anotar excepciones Java. Cuando utilizamos esta anotación en una excepción estamos indicando que cuando sea lanzada por una operación del servicio web debe generar un mensaje SOAP de respuesta con un <i>SOAP Fault</i> que nos indique el error producido. En el lado del cliente la clase con dicha excepción se habrá generado en el <i>stub</i> para el acceso al servicio, y al recibir el mensaje SOAP con el error el <i>stub</i> lanzará la excepción correspondiente. Es decir, para el desarrollador será como si la excepción saltase directamente desde el servicio hasta el cliente.</p> <pre>@WebFault public class ConversionFaultException extends Exception {     public ConversionFaultException(String msg)     {         super(msg);     } }</pre>

### 3.3.2. Estilo y codificación del servicio

Hemos visto que mediante la anotación `@SOAPBinding` podemos cambiar el estilo y la codificación del servicio. Los posibles estilos son:

- `SOAPBinding.Style.RPC`: Se utilizan mensajes SOAP orientados a RPC, en los que se codifican en XML las llamadas a métodos remotos.
- `SOAPBinding.Style.DOCUMENT`: Se utilizan mensajes SOAP orientados al documento. Dado que en estos mensajes se puede incluir cualquier tipo de documento XML, también se pueden utilizar para invocar operaciones de servicios.

Podemos especificar también la codificación:

- `SOAPBinding.Use.LITERAL`: Esta es la única codificación aceptada en el WS-I Basic Profile (BP), que da soporte a los servicios de tipo *document/literal* y *RPC/literal*.
- `SOAPBinding.Use.ENCODED`: Se trata de una codificación que permite representar una mayor variedad de estructuras de datos que la anterior, pero está desaprobada por el BP por ser la causa de gran cantidad de incompatibilidades entre servicios. De hecho JAX-WS es incompatible con los servicios de este tipo. Esta codificación se suele utilizar con servicios de tipo RPC, dando lugar al tipo *RPC/encoded*.

En el caso de los servicios de tipo *document/literal*, también podemos especificar la forma en la que se representan los tipos de datos de los parámetros de las operaciones:

- `SOAPBinding.ParameterStyle.BARE`: Los parámetros se pasan directamente.
- `SOAPBinding.ParameterStyle.WRAPPED`: Los parámetros se pasan envueltos en tipos de datos complejos.

Por defecto los servicios serán del tipo *document/literal/wrapped*.

### 3.3.3. Requerimientos de un servicio Web JAX-WS

---

#### Tipos de datos compatibles

Cuando trabajamos con JAX-WS, los tipos de datos que podremos utilizar como tipo de los parámetros y de valor de retorno de los métodos de nuestro servicio serán los tipos soportados por JAXB.

Podremos utilizar cualquiera de los tipos básicos de Java:

```
boolean
byte
double
float
int
long
short
char
```

Además, también podremos utilizar cualquiera de los *wrappers* de estos tipos básicos:

```
java.lang.Boolean
java.lang.Byte
java.lang.Double
java.lang.Float
java.lang.Integer
java.lang.Long
java.lang.Short
java.lang.Character
```

Las siguientes clases de Java también son aceptadas como tipos válidos por JAX-WS:

```
java.lang.String
java.math.BigDecimal
java.math.BigInteger
java.util.Calendar
java.util.Date
javax.xml.namespace.QName
```

```
java.net.URI
```

Además de estos datos, se permitirá el uso de colecciones cuyos elementos podrán ser de cualquiera de los tipos admitidos. Estas colecciones podrán ser *arrays*, tanto unidimensionales como multidimensionales, o clases del marco de colecciones de Java:

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

### 3.4. Implementación del servicio Web con el modelo EJB

Se puede utilizar un *Stateless Session Bean*, tal y como se define en la especificación de *Enterprise Java Beans*, para ser desplegado en un contenedor EJB. También se puede utilizar un *Singleton Session Bean* tal y como se define en la especificación EJB 3.1, para implementar un servicio Web JAX-WS para ser desplegado en un contenedor EJB.

Los requerimientos para crear una implementación de un servicio como un EJB de sesión sin estado y *singleton*, son las mismas que hemos visto en el apartado anterior para el modelo de programación con *servlets*.

Podemos anotar un *bean* de sesión sin estado con la anotación `javax.ejb.Stateless`. En este caso, la clase del *bean* ya no debe implementar la interfaz `javax.ejb.SessionBean`.

Un EJB de sesión sin estado y *singleton* que implementen un servicio Web utilizando la API de JAX-WS debería utilizar `javax.xml.ws.WebServiceContext`, el cual puede inyectarse utilizando la anotación `@Resource`, tal y como hemos visto en el ejemplo de la sección anterior.

En el caso de utilizar un *bean* de sesión *singleton* se utiliza la anotación `javax.ejb.Singleton`.

Por ejemplo, podemos implementar nuestro servicio Web como un *Stateless Session Bean* de la siguiente forma:

```
package jaxwsHelloServer;

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

@WebService
```

```

@Stateless
public class Hello {
    private String message = new String("Hola, ");

    public void Hello() {}

    @WebMethod
    public String sayHello(String name) {
        return message + name + ".";
    }
}

```

### 3.5. Empaquetado y despliegue de un servicio Web

Los componentes **Port** pueden empaquetarse en un fichero WAR o en un fichero EJB-JAR. Los componentes *Port* empaquetados en un fichero WAR deben usar una implementación de un servicio con el modelo de programación de servlets JAX-WS. Los componentes *Port* empaquetados en un fichero EJB-JAR deben utilizar un *stateless* o *singleton bean* para implementar el servicio web.

El desarrollador es responsable de empaquetar (bien incluyendo directamente, o referenciando), los siguientes elementos:

- el fichero WSDL (opcional si se utilizan anotaciones JAX-WS)
- la clase SEI (opcional con JAX-WS)
- la clase que implementa el servicio y sus clases dependientes
- los artefactos portables generados por JAX-WS (clases java generadas cuando se implementa un servicio Web a partir de una clase java, para ayudar al *marshaling/unmarshaling* de las invocaciones y respuestas del servicio web, así como de las excepciones específicas del servicio devueltas por dicho servicio)
- descriptor de despliegue en un módulo java EE (opcional si se usan anotaciones JAX-WS. Cualquier información contenida en este fichero "sobreescribe" la correspondiente información especificada con anotaciones)

El fichero WSDL se suele almacenar en un directorio `wsdl`, para su publicación durante el despliegue.

El descriptor de despliegue es específico del módulo que contenga el servicio web. En el caso del **empaquetado EJB**, el fichero descriptor de despliegue con la información del servicio web se localizaría en `META-INF/webservices.xml`. El directorio `wsdl` estará en `META-INF/wsdl`. Si utilizamos un **war** para empaquetar nuestro servicio web, el descriptor de despliegue se localiza en `WEB-INF/webservices.xml`, y el directorio que contiene el fichero WSDL se localiza en `WEB-INF/wsdl`.

Adicionalmente, se puede empaquetar el fichero de **catálogo** `jax-ws-catalog.xml` en el directorio `META-INF/jax-ws-catalog.xml`, en el caso de un módulo EJB, y `WEB-INF/jax-ws-catalog.xml`, en el caso de un módulo web. El fichero de catálogo se utiliza fundamentalmente para resolver las referencias a documentos de servicios web, específicamente documentos WSDL y ficheros de esquema.

Un ejemplo del contenido del fichero *jax-ws-catalog.xml* es el siguiente:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog"
  prefer="system">
  <system systemId="http://localhost:8080/holaMundo/hola?WSDL"
    uri="wsdl/localhost_8080/holaMundo/hola.wsdl"/>
  <system systemId="http://localhost:8080/holaMundo/hola?xsd=1"
    uri="wsdl/localhost_8080/holaMundo/hola.xsd_1.xsd"/>
</catalog>
```

## Despliegue del servicio Web

JAX-WS 2.2.2 RI (Implementación de referencia de JAX-WS. RI= *Reference Implementation*) soporta dos modelos de despliegue para publicar servicios web. Una posibilidad es utilizar el modelo definido por JSR-109 (*Web Services for Java EE*), que utiliza el fichero *webservices.xml* para definir el despliegue de los servicios. Otra posibilidad es utilizar el modelo de despliegue específico de JAX-WS RI, que define la configuración del despliegue en los ficheros *web.xml* y *sun-jaxws.xml*. Esta segunda opción es la que vamos a comentar aquí.

Nosotros vamos a utilizar Glassfish 3.1 para realizar el despliegue. Glassfish incluye todas las librerías y clases necesarias para desarrollar/desplegar servicios web en la plataforma Java EE, de forma que no necesitaremos incluir ningún descriptor de despliegue en nuestro empaquetado, ni siquiera el fichero *web.xml* en el caso de utilizar el modelo de servlets.

En el caso de utilizar el modelo *servlets*, y por lo tanto empaquetar nuestras clases en un *war*, este tipo de despliegue (sin utilizar ningún descriptor) NO es portable. Para un contenedor no Java EE, por ejemplo un Tomcat, el fichero *war* que contiene el servicio web no es diferente al *war* de una aplicación *servlet/jsp*. Si queremos trabajar con servicios web, tenemos que elegir una implementación de JAX-WS e incluir en el *war* las librerías necesarias de JAX-WS.

Otra cuestión con respecto al tipo de despliegue sin descriptores, es que la implementación de nuestro servicio web no sería "descubierta" automáticamente por el contenedor (por ejemplo Tomcat). Para ello deberíamos incluir algunos descriptores de despliegue para "decirle" a la librería JAX-WS cómo queremos que se desplieguen nuestros servicios web. Estos descriptores son específicos de la librería que estemos utilizando. Así, por ejemplo, si decidimos desplegar nuestro servidor en un Tomcat, tendríamos que añadir en el directorio WEB-INF los ficheros *sun-jaxws.xml* y *web.xml*. Ambos ficheros contendrán información para realizar el "despliegue" de los servicios web.

## Descriptores de despliegue JAX-WS: *web.xml* y *sun-jaxws.xml*

En el fichero *web.xml* declaramos el *listener* JAX-WS *WSServletContextListener*, que inicializa y configura el *endpoint* (componente *port*) del servicio web, y el *servlet*

JAXWS `WSServlet`, que es el que sirve las peticiones al servicio, utilizando la clase que implementa dicho servicio. Un ejemplo de contenido de fichero `web.xml` podría ser éste:

```
<?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"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <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>
```

El fichero `sun-jaxws.xml` contiene la definición de la implementación del *endpoint* del servicio. Cada *endpoint* representa un *port* WSDL, y contiene toda la información sobre la clase que implementa el servicio, *url-pattern* del *servlet*, información de *binding*, ubicación del fichero WSDL, y nombres "cualificados" (*qualified names*) del *port* y *service* del WSDL. Si no especificamos la ubicación del fichero WSDL, éste será generado y publicado durante el despliegue. Por ejemplo, en el siguiente fichero `sun-jaxws.xml` indicamos que la implementación de nuestro servicio viene dada por la clase `ws.news.NewsService` (dicha clase debe estar anotada con `@WebService`).

```
<?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>
```

## 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 soporte de ejecución de JAX-WS se encargará de convertir los mensajes SOAP de las llamadas entrantes al servicio en llamadas al API java de

JAX-WS, y viceversa (convertir la respuesta java en mensaje SOAP). El contenedor será el responsable de:

- "Escuchar" en un puerto determinado, o en la URI de la implementación del servicio, esperando peticiones SOA/HTTP)
- "Parsear" el mensaje de entrada, dependiendo del tipo de "enlazado" *binding* del servicio
- "Mapear" el mensaje a la clase y método correspondiente, de acuerdo con los datos de despliegue del servicio
- Crear los objetos java adecuados para el sobre (*envelope*) SOAP de acuerdo a la especificación JAX-WS
- Invocar al *Service Implementation Bean* y al método de instancia con los parámetros java adecuados
- Capturar la respuesta de la invocación si el estilo es petición-respuesta
- Mapear los objetos de respuesta java al mensaje SOAP
- Crear el *envelope* adecuado del mensaje para su transporte
- Enviar el mensaje al cliente del servicio Web

### Despliegue del servicio web

La herramienta de despliegue comienza el proceso examinando el artefacto desplegado para determinar qué módulos contienen servicios Web, para ello analiza las anotaciones de servicios web o los descriptores de despliegue contenidos en el módulo. A continuación obtiene la información de enlazado (*binding*), despliega los componentes y servicios web definidos en el módulo. Seguidamente publica los documentos WSDL que representan a los servicios web desplegados, configura al servidor e inicia la aplicación.

## 3.6. Creación de un servicio Web con JDK 1.6

Igual que en el caso de los clientes de servicios web, a partir de la versión 1.6 de JDK se incluyen herramientas para generar servicios web a partir de una clase java. Concretamente la herramienta que se utilizará para generar el servicio es *wsgen*, que al igual que *wsimport* se podrá utilizar tanto en línea de comando como en forma de tarea de ant.

Lo primero que deberemos hacer es compilar la clase que implementa el servicio al igual que cualquier otra clase Java, con la herramienta *javac*. Una vez hecho esto, generaremos el servicio con *wsgen* a partir de la clase compilada. Utilizaremos *wsgen* de la siguiente forma:

```
wsgen -cp <classpath> -s <src.dir> -d <dest.dir>
      <nombre.clase.servicio>
```

La clase que implementa el servicio (<nombre.clase.servicio>) se especificará mediante su nombre completo, es decir, incluyendo el nombre del paquete al que pertenece. Podemos proporcionar otros parámetros para indicar la forma en la que se



deben generar las clases, como el directorio donde queremos que guarde los fuentes de las clases generadas (<src.dir>), el directorio donde guardará estas clases compiladas (<dest.dir>), y el classpath, en el que deberá encontrarse la clase especificada.

En el caso concreto del servicio `Hello` definido anteriormente, podríamos generar las clases necesarias (después de haber compilado la clase `Hello`) de la siguiente forma:

```
wsgen -cp bin -s src -d bin
      jaxwsHelloServer.Hello
```

Con esto habremos creado las clases necesarias para publicar el servicio. Con JDK 1.6 no será necesario contar con un servidor de aplicaciones para publicar este servicio, sino que lo podremos publicar desde cualquier aplicación Java. Podemos publicar el servicio de la siguiente forma:

```
package jaxwsHelloServer;

import javax.xml.ws.Endpoint;

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

El método `Endpoint.publish` utiliza por defecto un contenedor servidor HTTP "ligero" que viene incluido con Java SE 6 y además nos permite desplegar nuestro servicio web sin tener que empaquetar ni desplegar nuestra aplicación. Esto es particularmente útil y práctico durante el desarrollo. De esta forma no es necesario tener en marcha Glassfish, Tomcat o cualquier otro servidor. De hecho, si previamente tenemos en marcha una instancia de algún otro servidor utilizando el mismo puerto, al hacer la llamada a `Endpoint.publish` nos dará un error informándonos de que dicho puerto ya está en uso.

Cuando ejecutemos la aplicación, podremos acceder al WSDL del servicio a través de cualquier navegador en la siguiente dirección:

```
http://localhost:8080/ServicioWeb/Hello?WSDL
```

### 3.7. Creación de un servicio Web JAX-WS con Maven

Vamos a ilustrar cómo, a partir de una clase java, y utilizando el modelo de programación de *servlets*, podemos construir, empaquetar y desplegar un servicio Web JAX-WS con Maven. Utilizaremos para ello la clase `expertoJava.Hola` como clase que va a implementar nuestro servicio Web.

Comenzamos creando una aplicación Web con Maven (similar a la que creamos para nuestro cliente Web del servicio en la sesión anterior)

```
mvn archetype:generate -DgroupId=expertoJava
                        -DartifactId=HolaMundo
                        -Dversion=1.0-SNAPSHOT
                        -DarchetypeArtifactId=webapp-javaee6
                        -DarchetypeGroupId=org.codehaus.mojo.archetypes
                        -DinteractiveMode=false
```

Añadiremos en nuestro pom el plugin para desplegar nuestro proyecto en glassfish (**maven-glassfish-plugin**), utilizando la goal *glassfish:deploy*:

```
<plugin>
  <groupId>org.glassfish.maven.plugin</groupId>
  <artifactId>maven-glassfish-plugin</artifactId>
  <version>2.1</version>
  <configuration>
    <user>admin</user>
    <passwordFile>/opt/glassfish-3.1.2.2/glassfish/domains/domain1/master-password
    </passwordFile>
    <glassfishDirectory>/opt/glassfish-3.1.2.2/glassfish</glassfishDirectory>
    <domain>
      <name>domain1</name>
      <adminPort>4848</adminPort>
      <httpPort>8080</httpPort>
    </domain>
    <components>
      <component>
        <name>HolaMundo</name>
        <artifact>target/HolaMundo-1.0-SNAPSHOT.war</artifact>
      </component>
    </components>
  </configuration>
</plugin>
```

Para generar los artefactos necesarios en la parte del servidor del servicio web no es necesario que incluyamos ningún *plugin* adicional en el pom de nuestro proyecto (plugin *wsgen*). Durante del despliegue se generarán automáticamente los ficheros necesarios (entre ellos el SEI de nuestro servicio Web) para poder utilizar nuestro servicio Web.

#### Nombres de los artefactos generados por Maven

Por defecto, el nombre del artefacto generado por Maven está formado por el *artifactId* de nuestro proyecto seguido de la versión del mismo (por ejemplo, en nuestro ejemplo se genera el artefacto *HolaMundo-1.0-SNAPSHOT.war*). Para utilizar cualquier otro nombre de nuestra elección simplemente tendremos que indicarlo utilizando la etiqueta `<finalName>nombre-artefacto-que-queramos</finalName>`, dentro de la etiqueta `<build></build>` de nuestro *pom.xml*

Ahora creamos nuestro servicio web como la clase `src/main/java/expertoJava/Hola.java` anotada con `@WebService`. El código será el siguiente:

```
package expertoJava;

import javax.jws.WebMethod;
```

```
import javax.jws.WebParam;
import javax.jws.WebService;

@WebService
public class Hola {

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

Como ya se ha comentado, NO es necesario configurar el despliegue utilizando `/src/main/webapp/WEB-INF/web.xml` y `/src/main/webapp/WEB-INF/sun-jaxws.xml` porque vamos a desplegar nuestro servicio Web sobre Glassfish.

Ahora ya estamos en disposición de compilar y desplegar nuestro servicio web:

```
mvn clean package
```

Recordemos que el servidor de aplicaciones debe estar en marcha para poder desplegar nuestro servicio web con:

```
./asadmin start-domain --verbose (desde /opt/glassfish-3.1.2.2/bin)
mvn glassfish:deploy (desde el raíz de nuestro proyecto)
```

Podemos ver el *wsdl* generado en:

```
http://localhost:8080/HolaMundoRaizContexto/HolaService?wsdl
```

Suponemos que la raíz del contexto se ha definido como "HolaMundoRaizContexto" en el fichero `src/main/webapp/WEB-INF/glassfish-web.xml`. Por defecto, si no especificamos el nombre del servicio en el atributo *serviceName* de la anotación `@WebService`, éste es el nombre de la clase, seguido del sufijo "Service".

Dicho *wsdl* se genera automáticamente al desplegar el servicio. También se genera de forma automática código para probar los servicios desplegados. Siguiendo con nuestro ejemplo, podremos probar nuestro servicio en la dirección:

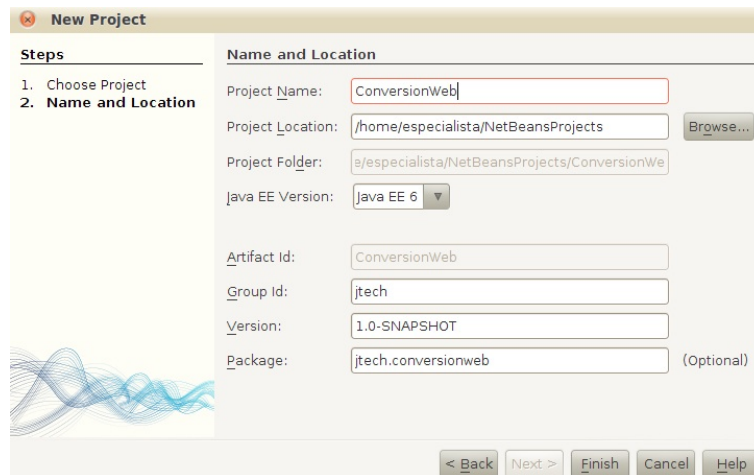
```
http://localhost:8080/HolaMundoRaizContexto/HolaService?tester
```

### 3.8. Creación de servicios web con Netbeans

Vamos a ver a continuación cómo crear servicios web paso a paso utilizando Netbeans. Seguiremos un ejemplo del servicio web de conversión (de euros a ptas y viceversa) para ilustrar el procedimiento que utilizaremos en Netbeans para crear un servicio web. Seguiremos los siguiente pasos:

Lo primero que necesitamos es un contenedor en el que crear nuestros servicios. Este contenedor será normalmente un proyecto web de Netbeans, aunque también podríamos utilizar un módulo EJB. Para nuestro ejemplo guiado vamos a crear un nuevo proyecto

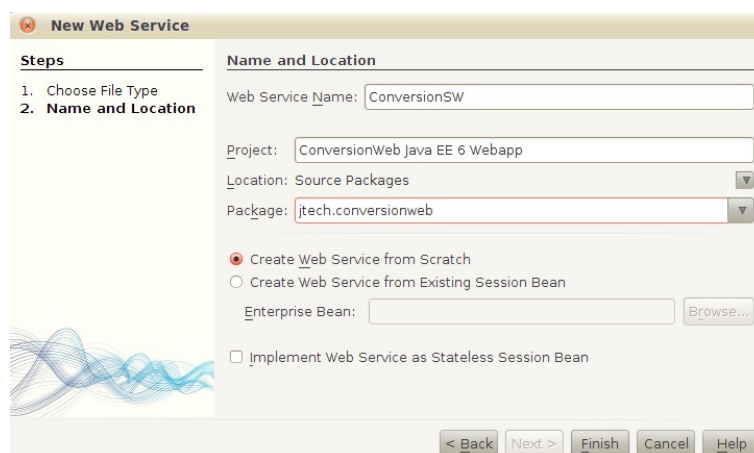
web llamado `ConversionWeb` al que añadiremos nuestro servicio.



Creación del proyecto `ConversionWeb`

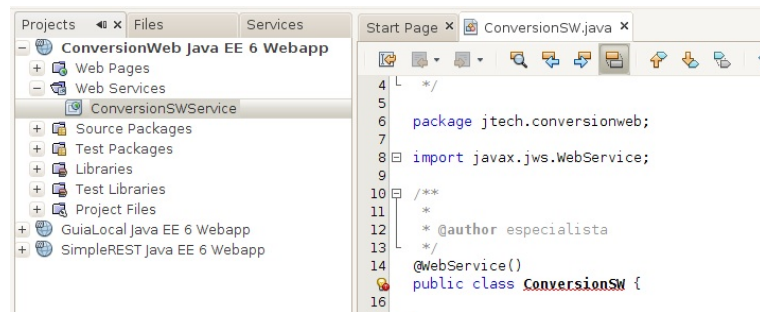
Una vez tenemos el proyecto web en el que introducir el servicio, pinchamos sobre él con el botón derecho y seleccionamos **New > Web Service ...** para añadir un servicio web.

Introduciremos el nombre que le queremos dar al servicio (nombre de la clase que implementará el servicio) y el paquete en el que estará. Aquí podemos crear un servicio web desde cero, o bien utilizar un EJB de sesión existente. Si utilizásemos esta segunda opción, los métodos del EJB se ofrecerían como operaciones del servicio web de forma automática, sin necesidad de hacer nada más. Para nuestro ejemplo vamos a quedarnos con la opción por defecto, que es crear el servicio web desde cero en una nueva clase Java plana.



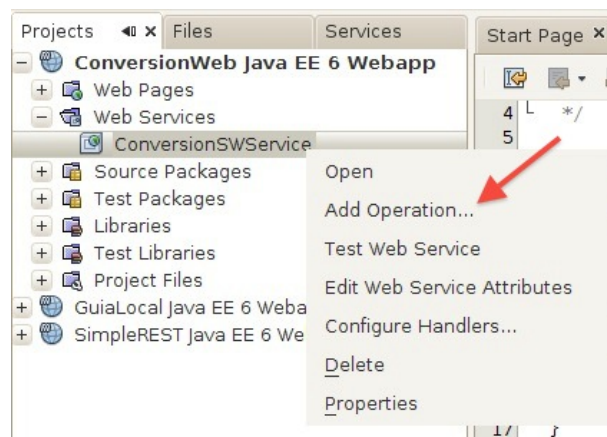
Creación del servicio Web `ConversionSW`

Una vez pulsemos el botón *Finish* se creará el servicio. En la vista de código podemos ver que se ha creado la clase `ConversionSW.java`.



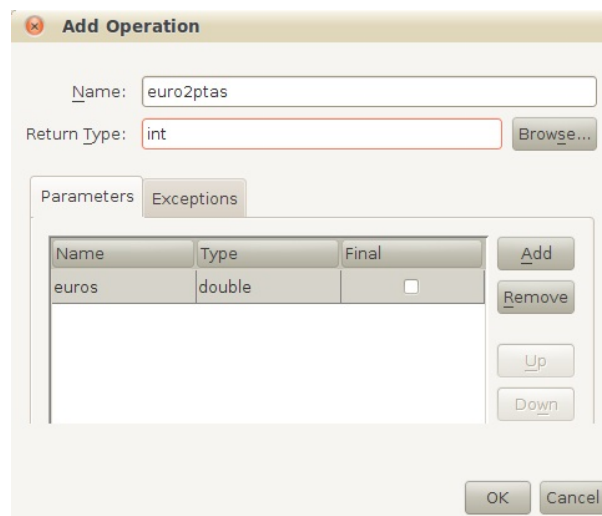
Vista de código del servicio Web creado

Ahora vamos añadir una operación a nuestro servicio, pulsando con el botón derecho sobre el servicio creado y eligiendo la opción "Add operation".



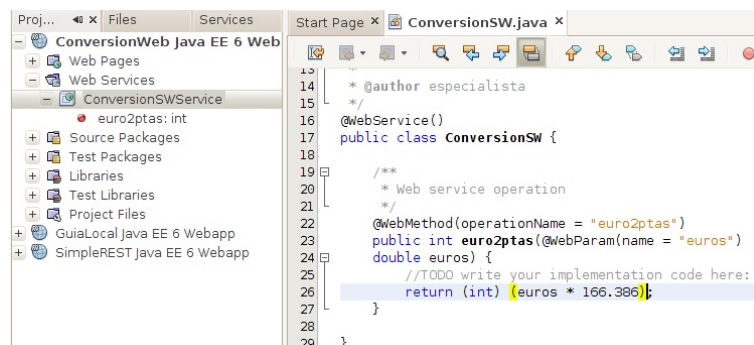
Añadimos una operación al servicio Web

Al añadir una operación deberemos especificar su nombre, el tipo de datos devuelto, y sus parámetros. En nuestro caso crearemos la función `euro2ptas`, con un parámetro `euros` de tipo `double`, y que devuelve un valor de tipo `int`.



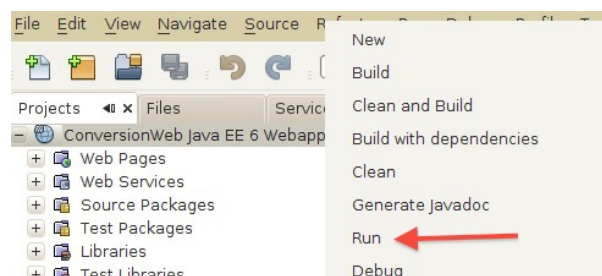
Datos de la operación euro2ptas

Una vez añadida la operación, en la vista de código vemos el esqueleto de la implementación de nuestro servicio. Debemos introducir en el método `euro2ptas` el código que realice la conversión de euros a pesetas.



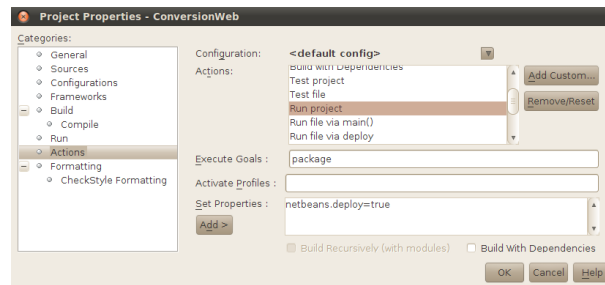
Implementación de la operación euro2ptas

Con esto ya tenemos implementado el servicio. Ahora podemos desplegar nuestro servicio con la opción "Run", con el botón derecho sobre el nodo del proyecto.



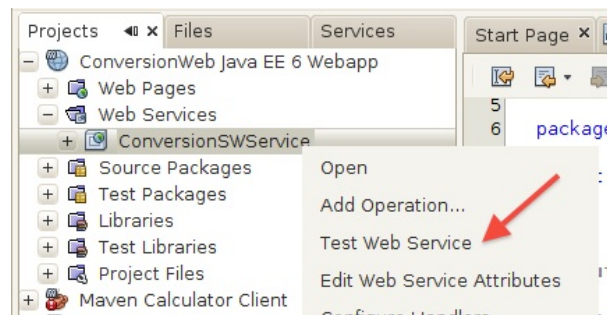
Despliegue de la aplicación ConversionSW

Podemos ver cómo está configurada por defecto la opción "Run" del menú contextual del proyecto (y también el resto de opciones de dicho menú), seleccionando "*Properties*" y a continuación *Actions* en el panel de la izquierda, y la acción "*Run Project*" de la lista de acciones de la derecha. En la siguiente figura se muestra el comando maven que se ejecuta cuando seleccionamos la opción "Run" del menú contextual del proyecto.



Acción asociada a la opción de menú contextual Run

Una vez la aplicación esté ejecutándose en el servidor, podremos probar el servicio pinchando sobre el servicio Web con el botón derecho y seleccionando la opción **Test Web Service**.



Invocación de la prueba del servicio web

Se abrirá en el navegador una web desde la que podremos probar el servicio. En ella podremos observar un **enlace al documento WSDL** que define el servicio, el cual nos resultará de utilidad cuando queramos crear un cliente que acceda a él, y la lista de operaciones que ofrece. Para cada operación tendremos cuadros de texto para introducir el valor de los parámetros de entrada que necesita, y un botón para invocarla.

### ConversionSWService Web Service Tester

This form will allow you to test your web service implementation ([WSDL File](#))

To invoke an operation, fill the method parameter(s) input boxes and click on the button labeled with the method name.

**Methods :**

public abstract int es.un.tech.serviceweb.conversion.ConversionSW.euro2Ptas(double)  
 (18.95)

Ejecución de la prueba del servicio web

Si probamos la operación *euro2ptas* pasando como parámetro *18.95*, veremos el resultado



de invocar el servicio, y además abajo en la misma página se mostrarán los mensajes SOAP utilizados para dicha invocación.

#### euro2Ptas Method invocation

---

##### Method parameter(s)

Type	Value
double	18.95

---

##### Method returned

int : "3153"

---

Resultado de la prueba del servicio web

### 3.9. Creación de servicios a partir de EJBs existentes

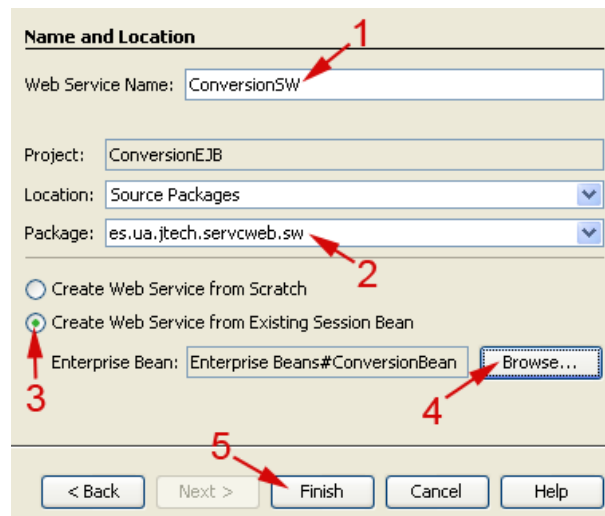
Además de poder crear servicios web desde cero, también podremos crearlos a partir de EJBs existentes. De esta forma lo que estaremos haciendo es exportar las operaciones de los EJBs en forma de servicios web, para poder acceder a ellas desde aplicaciones desarrolladas en otras plataformas o en cualquier otro lugar de la red.

Con Netbeans crear un servicio web a partir de un EJB es inmediato. Supongamos que tenemos un EJB en nuestra aplicación llamado `ConversionEJBBean`, que proporciona las operaciones `euros2ptas` y `ptas2euros`. Podremos exportar dichas operaciones en forma de servicio web de la siguiente forma:

Crearemos un nuevo servicio web en nuestro proyecto, con *New->Web Service...* al igual que en el caso anterior.

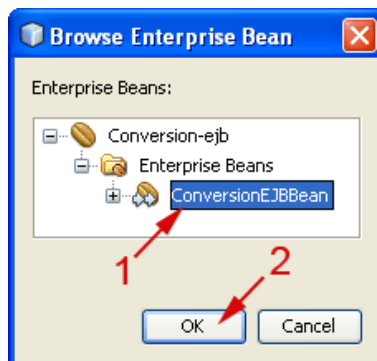
Ahora, además del nombre del servicio y el paquete en el que queremos crear sus clases, deberemos especificar que cree el servicio web a partir de un EJB (**Create Web Service from Existing Session Bean**), y pulsamos el botón *Browse ...* para seleccionar el EJB a partir del cual queramos crear el servicio.





Creación de un servicio a partir de un EJB

Seleccionaremos el EJB que queremos utilizar (en nuestro caso *ConversionEJBBean*), y pulsamos *OK* y a continuación *Finish* para finalizar la creación del servicio web.



Selección del EJB para crear el servicio

Con esto podremos acceder a las operaciones de nuestro EJB de sesión mediante un servicio web. A continuación mostramos el código de nuestro servicio Web, creado a partir del EJB existente:

```
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);
}
}

```

### 3.10. Creación de servicios a partir del WSDL

Hemos visto como crear con Netbeans servicios web a partir de código Java que ya tenemos implementado. Esta es la forma más inmediata de crear servicios web, sin embargo, si lo que buscamos es una alta interoperabilidad, no resulta la forma más adecuada de hacerlo. Podría darnos problemas sobre todo en el caso en que nuestras operaciones intercambien tipos de datos complejos, ya que podríamos tener problemas al intentar recomponer dichos tipos desde clientes de diferentes plataformas.

Lo fundamental en un servicio web SOAP es el **contrato** que existe entre cliente y servicio, es decir, el documento WSDL. Por lo tanto, a la hora de crear servicios web complejos es recomendable empezar definiendo dicho contrato. De esta forma tendremos mayor control sobre los datos que se serializan durante la invocación del servicio, con lo que podremos definir las estructuras de datos que consideremos más adecuadas para el intercambio. Una vez definido el contrato (WSDL), generaremos a partir de él el esqueleto para la implementación del servicio que cumpla con dicho contrato.

#### Creamos el WSDL y fichero de esquema

En Netbeans podremos generar un servicio web a partir de un documento WSDL de forma sencilla. Lo primero que deberemos hacer es escribir el documento WSDL, y el esquema asociado que definirá nuestro servicio. Ya hemos visto cómo hacer esto en la sesión anterior. Recordemos que tenemos que pinchar con el botón derecho sobre nuestro proyecto y seleccionar *New > Other ...*. Nos aparecerá la ventana para crear un nuevo fichero, y dentro de ella seleccionaremos la categoría *XML* y el tipo *XML Schema* (para el fichero de esquema), y *WSDL Document*, para el documento WSDL. Al continuar con el asistente, podremos introducir los datos básicos del documento WSDL, como su nombre, espacio de nombres, puertos, operaciones, mensajes, tipo de codificación, nombre del servicio, etc.

Una vez hayamos terminado de escribir el documento WSDL que actuará como contrato de nuestro servicio, deberemos crear nuestro servicio web. Para crear un servicio web que se ajuste a dicho contrato pincharemos con el botón derecho sobre nuestro proyecto y seleccionaremos *New > Web Service from WSDL...*. Tras rellenar todos los datos del asistente se generarán una serie de clases con el esqueleto de la implementación de nuestro servicio y los tipos de datos necesarios. Ahora deberemos rellenar el código de cada operación del servicio para darle su funcionalidad, y con esto habremos terminado de implementar nuestro servicio.

### 3.11. Paso de datos binarios

Supongamos que queremos crear un servicio Web que proporcione información binaria, por ejemplo ficheros de imágenes en formato *jpg*. Por defecto, la infraestructura de servicios de JAX-WS no puede informar a los clientes sobre cómo tienen que interpretar los datos binarios. Es decir, si en un mensaje SOAP incluimos datos binarios, éstos tendrán asociado el tipo *base64Binary*, que será mapeado a un array de *bytes*, y por lo tanto, el cliente tiene que saber cómo interpretar adecuadamente dichos datos.

Para poder enviar en nuestro mensaje SOAP un objeto *java.awt.Image*, por ejemplo, y que el cliente lo reciba como tal (y no como un array de bytes que posteriormente tenga que convertir al tipo *java.awt.Image*), básicamente lo que tendremos que hacer será editar el fichero de esquema generado para que devuelva datos binarios de tipo *image/jpeg*, y a continuación modificaremos el fichero wsdl para que utilice el fichero de esquema con la nueva configuración.

En el fichero de esquema (*.xsd*) tenemos que añadir el atributo *expectedContentTypes="mime\_type"* al elemento que devuelve los datos binarios (especificado mediante el atributo *type=xs:base64Binary* o *type=xs:hexBinary*). Este atributo (*expectedContentTypes*) informa al cliente de que debe mapear los datos binarios a un tipo Java (según las reglas de mapeado de tipo MIME a tipos Java), en lugar de a un array de bytes. A continuación mostramos una tabla que muestra el mapeado entre tipos MIME y tipos Java.

MIME Type	Java Type
image/gif	java.awt.Image
image/jpeg	java.awt.Image
text/plain	java.lang.String
text/xml or application/xml	javax.xml.transform.Source
*/*	javax.activation.DataHandler

#### Sobre tipos Mime

MIME es un estándar que clasifica los recursos y provee información (a los programas) acerca de cómo manejarlos. Esto permite la correcta manipulación e interpretación de diferentes tipos de archivos por parte de los programas (como navegadores). Por ejemplo, gracias a MIME, los navegadores pueden abrir correctamente un archivo ".txt" como un recurso de texto plano y no como un video u otro tipo. Cuando un tipo MIME no es especificado para un recurso, el programa que lo maneje puede "suponerlo" a partir de la extensión del mismo (por ejemplo, un archivo con la extensión ".bmp" debería contener una imagen de mapa de bits). Pero esto puede no siempre dar buenos resultados ya que una sola extensión puede asociarse a más de un formato. Por su parte, los tipos MIME son únicos. Ésta es la principal razón para utilizar los tipos MIME siempre que sea posible.

Así, por ejemplo, el siguiente elemento:

```
<element name="image" type="base64Binary"/>
```

será mapeado a **byte []**

mientras que el elemento:

```
<element name="image" type="base64Binary"
  xmlns:xmime="http://www.w3.org/2005/05/xmlmime"
  xmime:expectedContentTypes="image/jpeg"/>
```

será mapeado a **java.awt.Image**

Vamos a ilustrar el uso de este atributo con un ejemplo. Supongamos que tenemos un servicio Web que proporciona fotos, en este caso, de flores de jardín, en formato *jpeg*.

Nuestro servicio Web, denominado *FlowerService* proporciona la operación *getFlower*. Dicha operación tiene como entrada un *string* que representa el nombre de una flor, y como salida el fichero *jpg* con la foto de dicha flor, o bien una excepción en caso de no existir dicho fichero.

A continuación mostramos un extracto del wsdl con los mensajes asociados a la operación *getFlower*:

```
<!-- extraído de FlowerService.wsdl -->
<message name="getFlower">
  <part name="parameters" element="tns:getFlower"/>
</message>

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

<message name="IOException">
  <part name="fault" element="tns:IOException"/>
</message>

<portType name="FlowerService">
  <operation name="getFlower">
    <input message="tns:getFlower"/>
    <output message="tns:getFlowerResponse"/>
    <fault message="tns:IOException" name="IOException"/>
  </operation>
</portType>
```

La definición de tipos correspondiente en el fichero de esquema *xsd* es la siguiente:

```
<!-- extracto de FlowerService.xsd -->
<xs:element name="getFlower" type="tns:getFlower"/>
<xs:element name="getFlowerResponse" type="tns:getFlowerResponse"/>
<xs:element name="IOException" type="tns:IOException"/>

<xsd:complexType name="getFlower">
  <xsd:sequence>
```

```

        <xsd:element name="name" type="xs:string" minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>

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

    <xsd:complexType name="IOException">
      <xsd:sequence>
        <xsd:element name="message" type="xs:string" minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>

```

Si probamos el servicio web FlowerService mediante *Test Web Service*, veremos algo parecido a:

The screenshot shows a 'Method invocation trace' window with the following content:

**Method returned**

[B: "[B@2943e5"]

---

**SOAP Request**

```

<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header/>
  <S:Body>
    <ns2:getFlower xmlns:ns2="http://service.flower.org/">
      <name>rose</name>
    </ns2:getFlower>
  </S:Body>
</S:Envelope>

```

---

**SOAP Response**

```

<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:getFlowerResponse xmlns:ns2="http://service.flower.org/">
      <return>iVBORwOKGgoAAAAANSUhEUGAAoAAAAKACAIAAACDr15OAAcAAE
    </ns2:getFlowerResponse>
  </S:Body>
</S:Envelope>

```

### Paso de datos binarios como array de bytes

Nosotros queremos ver una imagen, y no una serie de símbolos. Sin embargo, ya que *java.awt.Image* no es un tipo de esquema válido, necesitamos configurar manualmente el fichero de esquema para devolver datos binarios con formato *image/jpeg*.

Si el servicio web lo hemos creado en un proyecto web Maven, necesitaremos incluir alguna modificación en el *pom* del proyecto. Concretamente tendremos que añadir explícitamente el *plugin wsdl* y poner la propiedad **genWsd** a **true**. Dicha propiedad genera el fichero wsdl de nuestro servicio web en el directorio por defecto

`${project.build.directory}/jaxws/wsgen/wsdl` (siendo `${project.build.directory}` el directorio *target* de nuestro proyecto). A continuación mostramos las modificaciones a realizar en el fichero *pom.xml*.

```
<!-- añadimos el plugin wsgen en nuestro pom.xml-->
<!-- para generar el ficheros wsdl + xsd en target/jaxsw/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>
```

Si volvemos a "compilar" nuestra aplicación, podemos ver que en el directorio *target/jaxws/wsgen/wsdl* se han generado los ficheros *wsdl* y *xsd* de nuestro servicio web. El siguiente paso es "copiar" dichos ficheros en el directorio *src/main/resources*. Cuando compilemos, por defecto todos los ficheros del directorio *src/main/resources* se copiarán en el WAR generado en el directorio WEB-INF/classes.

A continuación vamos a indicar de forma explícita al servidor de aplicaciones que queremos usar nuestra propia versión del fichero *wsdl* (que se encontrará en el directorio WEB-INF/classes. Si no lo hacemos así, el servidor de aplicaciones generará su propio fichero *wsdl*. Para ello, lo que tenemos que hacer es indicar de forma explícita el atributo **wsdlLocation** de nuestro servicio web, "apuntando" al directorio en donde estará el *wsdl* que queremos utilizar.

```
//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 {
    ...
  }
  ...
}
```

A continuación mostramos la modificación a realizar en el fichero *src/main/resources/FlowerService.xsd*:

```
<!-- extracto de FlowerService.xsd modificado con el atributo
expectedContentTypes-->

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

Una vez realizadas las modificaciones anteriores, si volvemos a realizar un "test" sobre nuestro servicio Web, veremos un resultado similar al siguiente:

### getFlower Method invocation

#### Method parameter(s)

Type	Value
java.lang.String	rose

#### Method returned

```
java.awt.Image : "BufferedImage@22c10c: type = 5 ColorModel:
#pixelBits = 24 numComponents = 3 color space =
java.awt.color.ICC_ColorSpace@1b02f36 transparency = 1 has alpha =
false isAlphaPre = false ByteInterleavedRaster: width = 640 height =
640 #numDataElements 3 dataOff[0] = 2"
```

Paso de datos binarios como image/jpeg

## 3.12. Servicios web con estado

Una de las características que más se han echado en falta en los primeros años de existencia de los servicios web es la capacidad de mantener un estado. Los servicios web eran servicios sin estado, en los que cada llamada era independiente de las demás, y si queríamos identificar de alguna forma al cliente que estaba realizando la llamada debíamos proporcionar como parámetro un identificador creado por nosotros que nos indicase de qué cliente se trataba. Esta carencia complicaba implementar elementos como por ejemplo un carrito de la compra mediante servicios web, al que pudiesemos añadir productos en sucesivas llamadas.

Para suplir esta carencia se han ideado diferentes técnicas, como por ejemplo el acceder a sesiones HTTP a través del objeto `WebServiceContext`. También encontramos otro

enfoque consistente en aprovechar la tecnología *WS-ReliableMessaging* para implementar el estado. Cada canal de datos tiene un identificador de sesión único, al que podemos acceder a través de la propiedad `com.sun.xml.ws.sessionid` del objeto `WebServiceContext`, y que puede ser utilizado para identificar a cada cliente y de esta forma mantener su estado.

Sin embargo, a partir de JAX-WS 2.1 aparece la tan esperada posibilidad de crear servicios web con estado (*stateful*). En este caso tendremos una instancia diferente de la clase del servicio para cada cliente. De esta forma, dentro de la clase que implementa el servicio podremos definir variables de instancia (a diferencia de los servicios *stateless*, en los que todos los campos debían ser estáticos), y cada cliente accederá a sus datos propios (almacenados en los campos de la instancia concreta a la que esté accediendo).

Estos servicios con estado están basados en la tecnología *WS-Addressing*. Esta tecnología permite identificar un *endpoint* de un servicio mediante XML, de forma independiente al protocolo de transporte que se vaya a utilizar para acceder a él. De esta forma se podrá especificar no sólo la dirección del servicio, sino también la instancia concreta a la que deseamos acceder de dicho servicio.

Vamos a ver un ejemplo sencillo de servicio con estado. Para crear un servicio de este tipo deberá estar anotado con `@Addressing`, para poder identificar desde el cliente la instancia concreta del servicio a la que conectarse, y con `@Stateful` para marcarlo como servicio web con estado. Al estar marcado de esta forma, el contenedor le inyectará de forma automática un objeto de tipo `StatefulWebServiceImpl` en un campo `manager` que será público y estático (o bien privado y accesible mediante *getters* y *setters*).

```
@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 StatefulWebServiceImpl<CuentaSW> manager;
}
```



Sabemos que cada cliente tendrá acceso a una instancia de este servicio. Pero, ¿cuándo se crea dicha instancia? ¿y quién será el encargado de crearla? No puede hacerse automáticamente cuando desde el cliente llega una petición, ya que no conocemos qué parámetros hay que pasarle al constructor (por esa razón los servicios *stateless* deben tener un constructor vacío, que es el que utiliza el contenedor para instanciarlos). Por lo tanto, estos servicios con estado deberán ser instanciados desde otros servicios. Por ejemplo, si queremos acceder a nuestra cuenta podemos hacerlo a través de un servicio BancoSW como el siguiente:

```
@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;
    }
}
```

Lo único que tiene de especial este servicio es que como resultado nos devuelve un objeto *W3CEndpointReference*, es decir, una referencia a un *endpoint* codificada mediante *WS-Addressing*. El *endpoint* al que hará referencia será a la instancia del servicio *CuentaSW* correspondiente a la cuenta solicitada. De esta forma cada cliente podrá acceder a una cuenta diferente, manteniendo cada una de ellas su estado por separado.

Podemos destacar también que la operación *export* del *manager* de la cuenta es la que genera la referencia al *endpoint*. Cuando queramos cerrar la sesión podemos utilizar *unexport* para que la instancia especificada del servicio deje de estar disponible como servicio web.

Vamos ahora a ver cómo accederemos a este servicio desde el cliente. Para ello lo primero que deberemos hacer es crear en nuestro proyecto cliente los *stubs* para acceder a los dos servicios creados anteriormente (al igual que hicimos en la sesión anterior). Una vez hecho esto podremos introducir el código del cliente como se muestra a continuación:

```
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();
```

Podemos observar que creamos los servicios BancoSW y CuentaSW igual que cualquier otro servicio. El puerto del banco también se obtiene de la misma forma que anteriormente, y a partir de él podemos llamar a la operación `abrirCuenta` para obtener el *endpoint* de la cuenta a la que queremos acceder. Ahora es cuando viene la parte diferente, ya que el puerto de la cuenta deberá obtenerse para que acceda al *endpoint* concreto que nos ha suministrado el banco. Para ello debemos utilizar una versión alternativa del método `getPort` sobre el servicio `CuentaSWService`. En esta versión deberemos suministrar tanto el *endpoint* obtenido, como la clase que define el tipo de puerto al que accederemos (`CuentaSW`). Esta versión de `getPort` sólo está disponible a partir de JAX-WS 2.1, por lo que con versiones anteriores de la librería no podremos acceder a este tipo de servicios.

## 4. Ejercicios. Creación de Servicios Web SOAP

### 4.1. Creación de un servicio web básico

Vamos a comenzar creando un servicio web básico JAX-WS con Maven (ver apartado Creación de un servicio Web JAX-WS con Maven). Este servicio web será un *Hola Mundo* en forma de servicio. La clase que implementará el servicio tendrá como nombre *HolaMundo* y una única operación `String saluda(nombre)`, que nos devolverá un mensaje de saludo incluyendo el nombre proporcionado. Por ejemplo, si al parámetro de entrada `nombre` le damos como valor "Miguel", como salida producirá la cadena "Hola Miguel, ¿Que tal?".

Cuando crees el arquetipo de Maven, utiliza los siguientes nombres para:

- groupId: expertoJava
- artifactId: Sesion2-HolaMundoWS

Configurar la raíz del contexto como "HolaMundoWS" en el fichero *glassfish-web.xml*. El nombre del servicio será *Hola* (valor del atributo *serviceName* de la anotación `@WebService`).

Crea un cliente web Maven con Netbeans para probar el funcionamiento de dicho servicio Web. El nombre del proyecto será "Sesion2-hola-WS-client (1 punto)

### 4.2. Validación de NIFs

Vamos a implementar un servicio web Maven (con Netbeans) con una serie de métodos que nos permitan validar un NIF. El servicio tendrá como nombre *ValidaDniWS*, y estará dentro de un proyecto Maven Netbeans con nombre *Sesion2-dni-WS*. Se pide:

a) Implementar la siguiente operación (0,25 puntos):

```
boolean validarDni(String dni)
```

Esta función tomará como entrada un DNI, y como salida nos dirá si es correcto o no. El resultado será `true` si el DNI es correcto (está formado por 8 dígitos), y `false` en caso contrario. Podemos utilizar un código similar al siguiente para validar el DNI:

```
Pattern pattern = Pattern.compile("[0-9]{8}");
Matcher matcher = pattern.matcher(dni);
return matcher.matches();
```

b) Implementar la siguiente operación (0,25 puntos):

```
char obtenerLetra(String dni) throws DniFault
```

Esta función tomará como entrada un DNI, y como salida nos dirá la letra que

corresponde a dicho DNI. Lo primero que deberá hacer es validar el DNI (puede utilizar para ello el método definido en el apartado anterior), y en caso de no ser correcto lanzará una excepción de tipo `DniFault` (que deberemos crear previamente) indicando el mensaje de error "El DNI no es valido". Una vez validado, calcularemos la letra que corresponda al DNI. Para ello deberemos:

- Obtener el índice de la letra correspondiente con:

```
dni % 23
```

- La letra será el carácter de la siguiente cadena que esté en la posición obtenida anteriormente:

```
"TRWAGMYFPDXBNJZSQVHLCKE"
```

#### Nota

El cliente de prueba no trata correctamente la recepción de un *SOAP Fault* como respuesta, ya que la excepción que genera éste hace que falle la misma aplicación de pruebas. Para poder tratar la excepción de forma correcta deberemos hacerlo con un cliente propio, que se implementará en el apartado (d).

- c) Implementar la siguiente operación (0,25 puntos):

```
boolean validarNif(String nif)
```

Esta función tomará como entrada un NIF, y como salida nos dirá si es correcto o no. El resultado será `true` si el NIF es correcto (está formado por 8 dígitos seguidos de la letra correcta), y `false` en caso contrario. Para hacer esta comprobación se pueden utilizar las dos funciones anteriores: se comprobarán los 8 primeros caracteres con la función `validarDni` y posteriormente se comprobará si la letra es la correcta utilizando la función `obtenerLetra`.

- d) Implementar un cliente Maven Java con Netbeans que acceda a dicho servicio e invoque la operación `obtenerLetra`. El cliente deberá crearse en un proyecto con nombre `Sesion2-dni-WS-client`. Tened en cuenta que debéis capturar la excepción que hemos definido en el apartado (b). La clase correspondiente del *stub* la podéis ver en *Files*, dentro del directorio `target/generated-sources/wsimport`, con el sufijo `_Exception`. Concretamente podéis ver que la excepción es de tipo `DniFault_Exception`. Prueba con un DNI válido, y haz una segunda llamada con un DNI inválido para comprobar que se lanza la excepción (0,25 puntos).

### 4.3. Tienda de DVDs

Nuestro negocio consiste en una tienda que vende películas en DVD a través de Internet. Para dar una mayor difusión a nuestro catálogo de películas, decidimos implantar una serie de Servicios Web para acceder a información sobre las películas que vendemos.

De cada película ofreceremos información sobre su título, su director y su precio. Esta información podemos codificarla en una clase `PeliculaTO` como la siguiente:

```
public class PeliculaTO {
    private String titulo;
    private String director;
    private float precio;

    public PeliculaTO() {}

    public PeliculaTO(String titulo, String director, float precio) {
        this.titulo = titulo;
        this.director = director;
        this.precio = precio;
    }

    // Getters y setters
    ...
}
```

Vamos a permitir que se busquen películas proporcionando el nombre de su director. Por lo tanto, el servicio ofrecerá una operación como la siguiente:

```
List<PeliculaTO> buscaDirector(String director)
```

Proporcionaremos el nombre del director, y nos devolverá la lista de películas disponibles dirigidas por este director.

En un principio, podemos crear una lista estática de películas dentro del código de nuestro servicio, como por ejemplo:

```
final static PeliculaTO[] peliculas = {
    new PeliculaTO("Mulholland Drive", "David Lynch", 26.96f),
    new PeliculaTO("Carretera perdida", "David Lynch", 18.95f),
    new PeliculaTO("Twin Peaks", "David Lynch", 46.95f),
    new PeliculaTO("Telefono rojo", "Stanley Kubrick", 15.95f),
    new PeliculaTO("Barry Lyndon", "Stanley Kubrick", 24.95f),
    new PeliculaTO("La naranja mecánica", "Stanley Kubrick", 22.95f)
};
```

Se pide:

a) Implementar el servicio utilizando Netbeans. El servicio se llamará `TiendaDvdWS`, y estará dentro de un proyecto Maven con nombre `Sesion2-tienda-WS`.

Para construir una lista con las películas cuyo director coincida con el nombre del director que se ha solicitado, podemos utilizar un código similar al siguiente, donde se ha proporcionado un parámetro `director`:

```
director = director.toLowerCase();

ArrayList<PeliculaTO> list = new ArrayList<PeliculaTO>();

for (PeliculaTO pelicula : peliculas) {
    if (pelicula.getDirector().toLowerCase().indexOf(director) != -1) {
        list.add(pelicula);
    }
}
```

```
return list;
```

Una vez implementado el servicio, desplegarlo en Glassfish y probarlo mediante el cliente de prueba generado. Observar los tipos de datos definidos en el documento WSDL generado y los mensajes SOAP para la invocación del servicio. (0,5 puntos)

b) Implementar un cliente java maven utilizando Netbeans. El cliente se llamará `Sesion2-tienda-WS-client`, y realizará una llamada al servicio mostrando, para cada película del director "kubrick", el título, el nombre del director y el precio. (0,5 puntos)

## 5. Orquestación de Servicios: BPEL

La respuesta más reciente al reto de la integración de aplicaciones es la arquitectura orientada a servicios (SOA) y las tecnologías de servicios web. El enfoque *bottom-up* de SOA considera que las diferentes aplicaciones de la empresa exponen sus funcionalidades a través de servicios web. De esta forma se puede acceder a diferentes funcionalidades de diferentes aplicaciones (tanto *legacy* como de nueva creación) de una forma común a través de servicios web.

Pero el desarrollar servicios web y exponer las funcionalidades no es suficiente. Necesitamos también alguna forma de componer dichas funcionalidades en el orden correcto. En esta sesión hablaremos de cómo realizar dicha secuenciación, concretamente nos centraremos en una de las aproximaciones posibles: la orquestación de servicios. Para ello haremos uso de un lenguaje denominado BPEL, que permite la composición de servicios según el paradigma de orquestación de servicios. El uso de BPEL hará posible una aproximación *top-down* (o aproximación orientada a los procesos) de SOA.

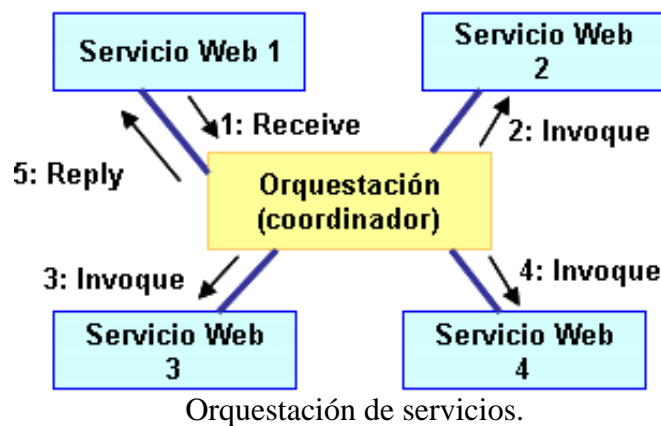
### 5.1. Orquestación frente a Coreografía

Los servicios Web "exponen" las operaciones de ciertas aplicaciones o sistemas de información. Consecuentemente, la combinación (composición) de varios servicios Web realmente implica la integración de las aplicaciones subyacentes y sus funcionalidades

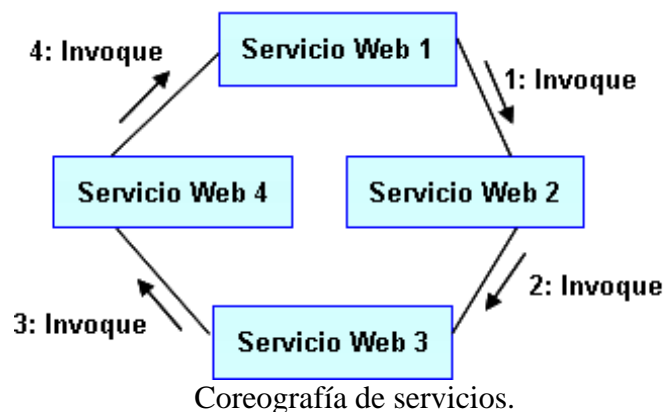
Los servicios Web pueden combinarse de dos formas:

- Orquestación
- Coreografía

Cuando se hace uso de **orquestación**, un proceso central (que puede ser otro servicio Web) lleva el control de los servicios Web implicados en la realización de una tarea y coordina la ejecución de las diferentes operaciones sobre dichos servicios Web. Los servicios Web orquestados no "conocen" (y no necesitan conocer) que están implicados en un proceso de composición y que forman parte de un proceso de negocio de nivel más alto. Solamente el coordinador central de la orquestación es "consciente" de la meta a conseguir, por lo que la orquestación se centraliza mediante definiciones explícitas de las operaciones y del orden en el que se deben invocar los servicios Web, tal y como se muestra en la siguiente figura. La orquestación se utiliza normalmente en procesos de negocio privados.



Cuando se hace uso de **coreografía**, no hay un coordinador central. En su lugar, cada servicio Web implicado en dicha coreografía "conoce" exactamente cuando ejecutar sus operaciones y con quién debe interactuar. La coreografía es un esfuerzo colaborativo centrado en el intercambio de mensajes en procesos de negocio públicos. Todos los participantes en la coreografía necesitan estar "informados" del proceso de negocio, las operaciones a ejecutar, los mensajes a intercambiar, y el tiempo a invertir en dicho intercambio de mensajes. Mostramos este esquema a continuación.



Desde la perspectiva de la composición de servicios Web para ejecutar procesos de negocio, la orquestación es un paradigma más flexible y tiene las siguientes ventajas sobre la coreografía:

- La coordinación de los procesos componentes está gestionada de forma centralizada por un coordinador conocido.
- Los servicios Web pueden incorporarse sin que sean "conscientes" de que están formando parte de un proceso de negocio mayor.
- Pueden crearse escenarios alternativos en caso de que ocurra algún fallo.

### 5.1.1. ¿Por qué orquestar Servicios Web?



Los servicios Web se utilizan actualmente tanto dentro como fuera de los límites de una organización. Los servicios Web se están convirtiendo en la tecnología común para proporcionar puntos de integración entre las aplicaciones (tanto por vendedores de grandes aplicaciones de empresa, como por negocios tradicionales de internet, como la tienda electrónica Amazon.com o la máquina de búsqueda Google, que exponen interfaces en forma de servicios Web para que dichas aplicaciones sean también "integrables").

El siguiente paso lógico en el salto a un modelo centrado en servicios (Web) es la orquestación de dichos servicios en nuevos procesos de negocio y servicios de más alto nivel. Los servicios Web proporcionan una tecnología de interfaces común que unifica el modelo de integración para todos los servicios independientemente de su origen. Los beneficios de los servicios Web, tales como su descubrimiento en tiempo de ejecución y su bajo acoplamiento, contribuyen a la orquestación de servicios Web proporcionando un acercamiento al modelado y ejecución de procesos de negocio en tiempo real. La idea es construir orquestaciones sin utilizar herramientas de programación tradicionales, tales como C# o Java, con el objetivo de reducir el esfuerzo total de producir nuevos servicios y aplicaciones basadas en servicios.

#### Recuerda

La orquestación de servicios Web pretende proporcionar una aproximación abierta, basada en estándares, para conectar servicios Web de forma conjunta con el objetivo de crear procesos de negocio de más alto nivel.

### 5.1.2. ¿Por qué orquestar servicios Web con BPEL?

Dentro de la orquestación de servicios, BPEL constituye un **lenguaje estándar** para la integración y automatización de procesos. Los procesos de negocio programados con BPEL serán capaces de ejecutarse en diferentes plataformas que cumplan dicho estándar, ofreciendo a los clientes una mayor libertad de elección.

Otras razones para utilizar orquestación de servicios y BPEL son:

- **Menores costes de mantenimiento:** con la utilización de un lenguaje de programación como C o Java, el mantenimiento puede resultar más difícil, especialmente si la lógica de negocio sufre cambios frecuentes. Además, no en todos los lenguajes de programación resulta fácil la manipulación de mensajes XML.
- **Menores costes de soporte:** Si una organización tiene la capacidad de integrar servicios Web, también será capaz de crear e invocar procesos BPEL utilizando la infraestructura de servicios Web existente. Además, BPEL tiene muchas construcciones relacionadas con la invocación de servicios Web, incluyendo el manejo de fallos, la correlación, y soporte para lógica condicional. BPEL tiene un soporte muy bueno para manejar mensajes XML a través de XPath (XPath se utiliza para referenciar partes de un mensaje XML y para realizar manipulaciones básicas de cadenas de caracteres y de números).

- Finalmente, BPEL hace que sea posible **ampliar el grupo de desarrolladores** para realizar tareas de integración de procesos de negocio y automatización de procesos, que requerían habilidades altamente especializadas. Utilizando herramientas gráficas de diseño, incluso los no desarrolladores, tales como los analistas del negocio pueden diseñar y modificar de forma sencilla las aplicaciones a medida que el negocio requiere cambios

#### Recuerda

El estándar BPEL ha sido diseñado para reducir la complejidad requerida para orquestar servicios Web, reduciendo así tiempo y costes de desarrollo, e incrementar la eficiencia y exactitud de los procesos de negocio. Sin un estándar cada organización tiene que construir su propio conjunto de protocolos de negocio propietarios, dejando poca flexibilidad para una verdadera colaboración de los servicios Web.

Además, el lenguaje BPEL proporciona soporte para:

- **Elevados tiempos de ejecución.** Normalmente, los procesos de negocio tardan bastante tiempo en completar su ejecución, particularmente si éstos implican interacciones con *partners* a través de Internet. Puede ocurrir que dichos procesos requieran minutos, horas, e incluso días antes de su finalización. Puede ocurrir que invoquen a un servicio Web y necesiten esperar la respuesta durante un periodo relativamente largo de tiempo. Si utilizásemos una aplicación Java, por ejemplo, en lugar de un proceso BPEL tendríamos que dedicar mucho tiempo a monitorizar qué procesos han terminado, y cuáles siguen todavía en ejecución. También necesitaríamos hacer un seguimiento de qué aplicaciones Java (procesos) podemos "cerrar" y cuáles tenemos que seguir manteniendo en ejecución debido a que están a la espera de recibir una respuesta.
- **Compensación.** Llamamos compensación al proceso de deshacer pasos realizados en un proceso de negocio que ya se han completado con éxito. Este concepto es uno de los más importantes en los procesos de negocio. El objetivo de la compensación es dar marcha atrás a los efectos de actividades previas que ya se han llevado a cabo como parte de un proceso de negocio que está siendo abandonado.

La compensación está relacionada con la naturaleza de la mayoría de procesos de negocio que se ejecutan durante largos periodos de tiempo y utilizan comunicación asíncrona con servicios Web *partner* débilmente acoplados. Este concepto es similar al de transacciones *ACID* (*Atomic, Consistent, Isolation, Durable*) utilizadas en sistemas de información de empresas. BPEL soporta el concepto de compensación con la posibilidad de definir manejadores de compensación, que tienen un ámbito de acción específico. A esta característica la denomina *Long-Running Transactions (LRT)*.

- **Reacción ante eventos.** Dichos eventos pueden ser mensajes o alarmas. Los eventos de tipo mensaje son generados por mensajes de entrada a través de la invocación de operaciones sobre los *port types*. Los eventos de tipo alarma están relacionados con el tiempo y se generan después de un tiempo especificado.

- **Modelado de actividades concurrentes.** La concurrencia se modela mediante la actividad <flow>. Ésta permite especificar escenarios concurrentes complejos, como por ejemplo el expresar dependencias de sincronización entre actividades. Es decir, podemos especificar qué actividades pueden comenzar y cuando (dependiendo de otras actividades), definiendo así dependencias complejas. Por ejemplo, podemos especificar que una cierta actividad, o varias actividades no pueden comenzar antes de que otra actividad o varias actividades hayan terminado.
- **Modelos con estado.** A diferencia de los servicios Web, que presentan un modelo sin estado, los procesos de negocio requieren el uso de un modelo con estado. Cuando un cliente comienza un proceso de negocio, se crea una nueva instancia. Esta instancia "vive" mientras no termine el proceso de negocio. Los mensajes enviados al proceso necesitan ser entregados a las instancias correctas del proceso de negocio. BPEL proporciona un mecanismo para utilizar datos de negocio específicos para mantener referencias a instancias específicas del proceso de negocio. A este mecanismo lo denomina *correlación*.

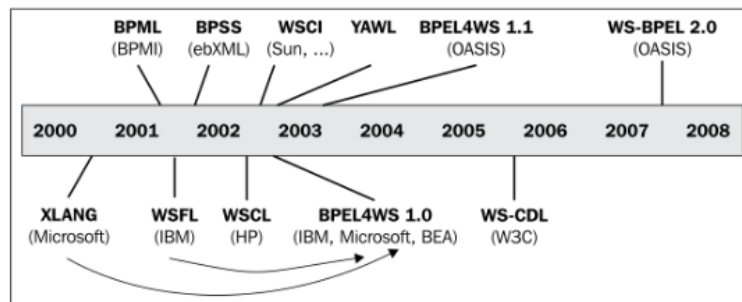
Por todo ello, BPEL resulta la opción más adecuada para orquestar procesos frente a otros lenguajes de programación, como por ejemplo Java.

## 5.2. El lenguaje BPEL

La aproximación orientada a procesos de SOA requiere un lenguaje para realizar una descripción relativamente simple de cómo pueden componerse los servicios Web convirtiéndose en procesos de negocio. Sería estupendo que tales descripciones pudiesen también ejecutarse, lo cual permitiría no sólo proporcionar definiciones de procesos abstractos, sino escribir especificaciones ejecutables de dichos procesos. BPEL es dicho lenguaje. Un proceso abstracto BPEL representa, de forma estándar, un conjunto de comportamientos observables.

BPEL (*Business Process Execution Language*) es un lenguaje de orquestación de servicios. Es un lenguaje basado en XML que soporta las tecnologías de servicios Web (incluyendo SOAP, WSDL, UDDI, WS-Reliable Messaging, WS-Addressing, WS-Coordination, y WS-Transaction).

BPEL representa una convergencia entre dos lenguajes *workflow*: WSFL (*Web Services Flow Language*) y XLANG. WSFL fue diseñado por IBM y está basado en el concepto de grafos dirigidos. XLANG fue diseñado por Microsoft y es un lenguaje estructurado en bloques. BPEL, que inicialmente recibió el nombre de BPEL4WS, combina ambas aproximaciones proporcionando un vocabulario rico para la descripción de procesos de negocio. En Abril de 2003, BEA Systems, IBM, Microsoft, SAP y Siebel Systems decidieron someter BPEL4WS 1.1 al comité técnico de OASIS para su estandarización, recibiendo el nombre de WS-BPEL 2.0 (en Septiembre de 2004). En esta sesión hablaremos simplemente de BPEL (aunque nos referiremos a WS-BPEL 2.0). La siguiente figura muestra la evolución de BPEL:



Evolución del lenguaje BPEL.

BPEL puede utilizarse dentro de una empresa y entre empresas. Dentro de una empresa, el papel de BPEL es el de estandarizar la integración de aplicaciones y extender la integración de sistemas previamente aislados. Entre empresas, BPEL permite una integración más fácil y efectiva con *partners* del negocio. BPEL es una tecnología clave en entornos en donde las funcionalidades ya están o serán expuestas via servicios Web.

BPEL ha sido diseñado específicamente como un lenguaje para la definición de procesos de negocio. BPEL soporta dos tipos diferentes de procesos de negocio:

- **Procesos de negocio ejecutables:** especifican los detalles exactos de los procesos del negocio y pueden ser ejecutados en una máquina de orquestación (*orchestration engine*). En la mayoría de los casos BPEL se utiliza para procesos ejecutables
- **Procesos abstractos de negocio:** especifican solamente el intercambio de mensajes públicos entre las partes implicadas, sin incluir detalles específicos de los flujos de los procesos. No son ejecutables y raramente se utilizan.

#### Recuerda

BPEL es un lenguaje de ejecución de procesos de negocio. Un **proceso de negocio** (*Business Process*) es un flujo de trabajo, formado por una serie de pasos o actividades, que son requeridas para completar una transacción de negocio. Las actividades del proceso de negocio pueden requerir la invocación de aplicaciones y/o intervención humana para su consecución. Un proceso de negocio típicamente tiene una duración larga, implicando a múltiples partes y/o aplicaciones tanto dentro de una empresa como entre empresas.

### 5.3. Estructura de un proceso BPEL

Un proceso BPEL especifica el orden exacto en el que deben invocarse los servicios Web participantes, tanto de forma secuencial como en paralelo. Con BPEL podemos expresar un comportamiento condicional, por ejemplo la invocación de un servicio Web puede depender del valor de una invocación previa. También podemos construir bucles, declarar variables, copiar y asignar valores, y definir manejadores de fallos, entre otras cosas. Combinando todas estas construcciones podemos definir procesos de negocio complejos de una forma algorítmica. De hecho, debido a que los procesos de negocio son esencialmente grafos de actividades, podría ser útil expresarlos utilizando diagramas de

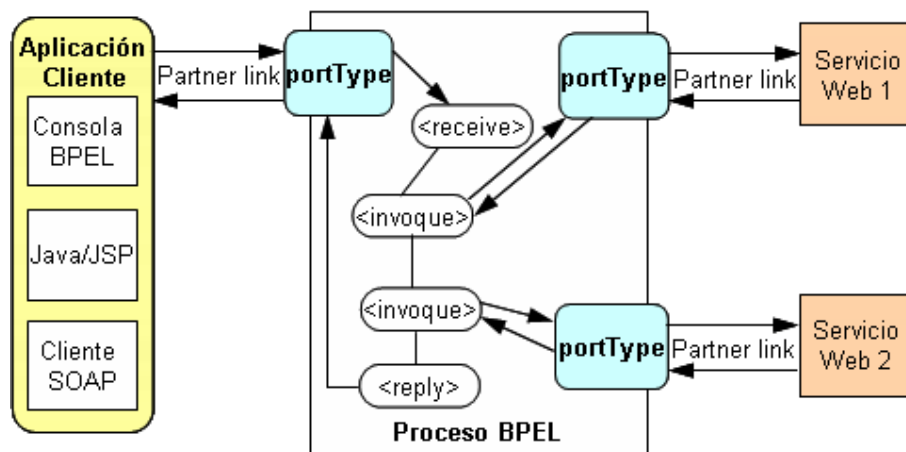
actividad UML.

En un escenario típico, el proceso de negocio BPEL recibe una petición de un cliente (que puede ser una aplicación cliente, u otro servicio Web, entre otros). Para servir dicha petición, el proceso invoca a diversos servicios Web y finalmente responde al cliente que ha realizado la llamada. Debido a que el proceso BPEL se comunica con otros servicios Web, tiene que tener en cuenta la descripción WSDL de los servicios Web a los que llama.

#### Recuerda

Un **proceso BPEL** es un servicio con estado de grano "grueso" (*large-grained*), que ejecuta unos pasos para completar una meta de negocio. Dicha meta puede ser la ejecución de una transacción de negocio, o la consecución del trabajo requerido por un servicio. Los pasos en el proceso BPEL ejecutan actividades (representadas por elementos del lenguaje BPEL) para llevar a cabo su cometido. Dichas actividades se centran en la invocación de servicios participantes para realizar las diferentes tareas y devolver los correspondientes resultados al proceso. El trabajo resultante de la colaboración de todos los servicios implicados es una **orquestración de servicios**.

Para los clientes de un proceso BPEL, éste es como cualquier otro servicio Web. Cuando definimos un proceso BPEL, esencialmente definimos un servicio Web que es una composición de otros servicios Web existentes. La interfaz del nuevo servicio Web BPEL utiliza un conjunto de **port types** a través de los cuales ofrece operaciones igual que cualquier otro servicio Web. Cada *port type* puede contener varias operaciones. Para invocar un proceso de negocio descrito con BPEL, tenemos que invocar al servicio Web resultante de la composición, tal y como mostramos en la siguiente figura.



Vista esquemática de un proceso BPEL.

Un proceso BPEL puede ser síncrono o asíncrono. Un **proceso BPEL síncrono** bloquea al cliente (aquel que usa el proceso BPEL) hasta que finaliza y devuelve el resultado a dicho cliente. Un **proceso asíncrono** no bloquea al cliente. Para ello utiliza una llamada *callback* que devuelve un resultado (si es que lo hay). Normalmente utilizaremos procesos asíncronos para procesos que consumen mucho tiempo, y procesos síncronos

para aquellos que devuelven un resultado en relativamente poco tiempo. Si un proceso BPEL utiliza servicios Web asíncronos, entonces el propio proceso BPEL normalmente también lo es. Hablaremos de los procesos asíncronos en la siguiente sesión.

La estructura básica de un documento (fichero con extensión **.bpel**) que define un proceso BPEL es la siguiente:

```
<process name="nameProcess" ... >

  <partnerLinks>
    <!-- Declaración de partner links -->
  </partnerLinks>

  <variables>
    <!-- Declaración de variables -->
  </variables>

  <sequence>
    <!-- Cuerpo principal de la definición del proceso BPEL -->
  </sequence>

</process>
```

En la etiqueta **<process>** se añaden los espacios de nombres. Aquí tenemos que definir tanto el espacio de nombres objetivo ( **targetNamespace**), como los *namespaces* para acceder a los WSDL de los servicios Web a los que invoca, y al WSDL del proceso BPEL. Tenemos que declarar también el *namespace* para todas las etiquetas y actividades BPEL (que puede ser el espacio de nombres por defecto, para así no tener que incluirlo en cada nombre de etiqueta BPEL). El *namespace* para actividades BPEL debe ser: *http://schemas.xmlsoap.org/ws/2003/03/business-process/*:

```
<process name="nameProcess"
  targetNamespace= "http://..."
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:sw1="http://..." <!-- namespace del servicio Web sw1 -->
  xmlns:sw2="http://..." <!-- namespace del servicio Web sw2 -->
  ... >

...

</process>
```

Un proceso BPEL está formado por una serie de pasos. Cada uno de los pasos se denomina **actividad**. La etiqueta **<sequence>** contiene el conjunto de pasos o actividades que conforman el servicio que proporciona el proceso BPEL. Hablaremos de las actividades un poco más adelante.

La etiqueta (o elemento) **<process>** está siempre presente en un proceso BPEL, es decir, es el mínimo requerimiento en un fichero BPEL.

### 5.3.1. Partner Links

Lo siguiente que ha de hacerse es definir los enlaces con las diferentes "partes" que interactúan con el proceso BPEL, es lo que denominamos como *partner links*. Cada *partner link* tiene asociado un *partnerLinkType* específico, y que se definirá en el WSDL correspondiente.

Vamos a explicar un poco mejor este concepto. Los procesos BPEL interactúan con servicios Web externos de dos formas distintas:

- El proceso BPEL invoca operaciones sobre otros servicios Web.
- El proceso BPEL recibe invocaciones de clientes. Uno de los clientes es el usuario del proceso BPEL, que realiza la invocación inicial. Otros clientes son, por ejemplo, servicios Web que han sido invocados por el proceso BPEL, pero realizan *callbacks* para devolver las respuestas solicitadas. En este último caso, el proceso cliente tiene dos *roles*: es invocado por el proceso BPEL e invoca al proceso BPEL.

Cada proceso BPEL tiene al menos un *partner link* cliente, debido a que tiene que haber un cliente que invoque al proceso BPEL. Por otro lado, un proceso BPEL tendrá (normalmente) al menos un *partner link* a quién invoque.

BPEL trata a los clientes como *partner links* por dos razones. La más obvia es para proporcionar soporte para interacciones asíncronas. En interacciones asíncronas (como veremos en la siguiente sesión), los procesos necesitan invocar operaciones sobre sus clientes. Dichos procesos cliente devuelven el resultado mediante *callbacks* sobre el proceso que los invocó.

La segunda razón está basada en el hecho de que el proceso BPEL puede ofrecer servicios. Estos servicios, ofertados a través de *port types*, pueden utilizarse por más de un cliente. El proceso puede querer distinguir entre diferentes clientes y ofrecerles únicamente la funcionalidad que éstos están autorizados a utilizar.

#### Recuerda

Los *partner links* definen enlaces con los *partners* (partes que interaccionan con nuestro proceso de negocio) del proceso BPEL. Dichos *partners* pueden ser: (a) Servicios invocados por el proceso; (b) Servicios que invocan al proceso; (c) Servicios que tienen ambos roles: son invocados por el proceso e invocan al proceso. Ya que un cliente debe llamar al proceso BPEL, éste debe contener **al menos una** definición de *partnerLink*.

A partir de aquí utilizaremos *ncname* y *qname* para expresar el formato que deben seguir ciertos nombres dados a los atributos de las etiquetas BPEL.

- *ncname* es una cadena de caracteres que debe comenzar siempre por una letra o un signo de subrayado.
- *qname* es un *ncname* que también debe mostrar de qué espacio de nombres XML proviene. Por lo tanto, un *qname* siempre tiene que seguir el formato *tns:ncname*, en donde *tns* es el nombre del espacio de nombres definido al comienzo del documento correspondiente.



La sintaxis para especificar los *partnerLinks* es:

```
<partnerLinks>
  <partnerLink name="ncname" partnerLinkType="qname"
               myrole="ncname" partnerRole="ncname">
  </partnerLink>
</partnerLinks>
```

Cada *partner link* es definido por un *partner link type* y un nombre de rol (o dos, como veremos a continuación).

Cada *partner link* especifica uno o dos atributos, que tienen que ver con el rol que implementa cada uno de los servicios relacionados:

- *myRole*: indica el rol del propio proceso BPEL. Cuando solamente se especifica este atributo, cualquier cliente o *partner* puede interactuar con el proceso BPEL sin ningún requerimiento adicional.
- *partnerRole*: indica el rol del partner. Si solamente definimos este atributo, estamos permitiendo la interacción con un *partner* o cliente que no imponga requerimientos sobre el proceso que haga la llamada.

Utilizaremos un único rol para una operación síncrona, ya que los resultados deben devolverse utilizando la misma operación. Utilizaremos dos roles en una operación asíncrona, ya que el rol del *partner* cambia cuando se realiza la llamada *callback* (hablaremos de las operaciones asíncronas en la siguiente sesión).

Los roles se definen en el documento WSDL de cada partner, cuando se especifican los *partnerLinkTypes*.

### **Partner Link Types**

Un *partnerLinkType* especifica la relación entre dos servicios, definiendo el rol que cada servicio implementa. Es decir, declara **cómo** interactúan las partes y lo que cada parte ofrece. Los nombres de los roles son cadenas de caracteres arbitrarias. Cada **rol** especifica exactamente un tipo *portType* WSDL que debe ser implementado por el servicio correspondiente.

Es fácil confundir los *partner link* y los *partner link types*, sin embargo:

- Los *partner link types* y los roles son extensiones especiales de WSDL definidas por la especificación BPEL. Como tales, dichos elementos se definen en los **ficheros WSDL**, **no** en el fichero del proceso BPEL.
- *Partner link* es un elemento BPEL 2.0. Por lo que se define en el **fichero del proceso BPEL**.

El siguiente código corresponde a la descripción WSDL de un proceso BPEL, al que denominaremos *Saludo*, en el que declaramos el tipo *MyPartnerLinkType*, con el rol denominado *ProveedorServicioSaludo*. El servicio que implemente el rol *ProveedorServicioSaludo*, deberá implementar *SaludoPortType*. En este caso,



*MyPartnerLinkType* describe la relación entre el cliente del proceso BPEL y el propio proceso BPEL.

```
<!-- Extracto de Saludo.wsdl -->
<partnerLinkType name="MyPartnerLinkType">
  <role name="ProveedorServicioSaludo"
    portType="SaludoPortType"/>
</role>
</partnerLinkType>
```

A continuación mostramos un extracto del fichero que describe el proceso BPEL *Saludo*):

```
<!-- Extracto de Saludo.bpel -->
<partnerLinks>
  <partnerLink name="cliente"
    partnerLinkType="MyPartnerLinkType"
    myRole="ProveedorServicioSaludo"/>
</partnerLinks>
```

Cuando utilizamos operaciones asíncronas, se definen dos roles, pero esto lo veremos en la siguiente sesión cuando hablemos de los procesos BPEL asíncronos.

#### Recuerda

Cada *partnerLink* de un proceso BPEL se relaciona con un *partnerLinkType* que lo caracteriza. Cada *partnerLinkType* se define en el **fichero WSDL**: (a) del proceso BPEL, en el caso de que describa la interacción del cliente con el propio proceso BPEL, o (b) del servicio Web al que invoca dicho proceso BPEL.

### 5.3.2. Variables

Las variables se utilizan para almacenar, reformatear, y transformar mensajes que contienen el estado del proceso. Normalmente necesitaremos una variable para cada mensaje que enviemos a los *partners* y cada mensaje que recibamos de ellos. Aunque también podemos almacenar datos que representen el estado del proceso y no se envíen a los *partners*.

Para cada variable tenemos que especificar su tipo, que puede ser:

- de tipo mensaje WSDL: indicado mediante el atributo *messageType*
- de un tipo simple de un esquema XML: indicado mediante el atributo *type*
- de tipo elemento de esquema XML: indicado mediante el atributo *element*

La sintaxis de una declaración de variable es:

```
<variables>
  <variable name="nombreVar"
    messageType="qname" //tipo mensaje
    type="qname" //tipo simple
    element="qname" /> //tipo elemento
</variables>
```

```
</variables>
```

El nombre de una variable debe ser único en su ámbito de aplicación (*scope*). Debe utilizarse **uno y sólo uno** de los tres atributos *messageType*, *type*, o *element*.

El siguiente ejemplo muestra tres declaraciones de variables. La primera de ellas declara una variable con el nombre *PeticionArticulo*, que almacena mensajes WSDL del tipo *art:MensajePeticionArticulo*. La segunda declaración define una variable denominada *DescripcionParcialArticulo* que almacena elementos XML del tipo *art:DescripcionArticulo*. La última declaración de variable lleva el nombre *Telefono* y puede almacenar un tipo de dato *string* de un esquema XML. Las dos primeras declaraciones asumen que los *messageType* y *element* correspondientes han sido declarados en el fichero WSDL.

```
<variables>
  <variable name="PeticionArticulo"
    messageType="art:MensajePeticionArticulo"/>
  <variable name="DescripcionParcialArticulo"
    element="art:DescripcionArticulo"/>
  <variable name="Telefono"
    type="xs:string"/>
</variables>
```

Cada variable tiene que declararse en un cierto ámbito y sólo pertenece a ese ámbito, y a los ámbitos dentro de éste. Podemos declarar variables globalmente en la parte inicial de las declaraciones del proceso BPEL. Para declarar variables locales a un cierto ámbito se utiliza la etiqueta **<scope>**.

Las variables se utilizan normalmente en condiciones. Podemos utilizar la función *getVariableData* para extraer valores arbitrarios de las variables. La función tiene tres parámetros: el nombre de la variable (es un parámetro obligatorio), nombre de la parte del mensaje (opcional), y el *path query* (opcional). La sintaxis es la siguiente:

```
getVariableData ('variable-name', 'part-name', <!-- optional -->
  'location-path') <!-- optional -->
```

El tercer parámetro se escribe con un lenguaje de *query*, como por ejemplo XPath

La siguiente expresión devuelve la parte del mensaje *insuredPersonData* del mensaje almacenado en la variable *InsuranceRequest*

```
getVariableData ('InsuranceRequest', 'insuredPersonData')
```

La siguiente expresión devuelve el elemento *Age* de la variable *InsuranceRequest*

```
bpws:getVariableData ('InsuranceRequest', 'insuredPersonData',
  '/insuredPersonData/ins:Age')
```

## Asignación

El copiado de datos de una variable a otra es algo que ocurre bastante a menudo en un

proceso de negocio. Para ello se utiliza la actividad *assign*. Esta actividad también puede utilizarse para copiar nuevos datos en una variable.

```
<assign>
  <copy>
    <from variable="ncname" part="ncname"/>
    <to variable="ncname" part="ncname"/>
  </copy>
</assign>
```

### 5.3.3. Actividades

Como ya hemos comentado, un proceso BPEL está formado por una serie de pasos. Cada uno de los pasos se denomina **actividad**. BPEL soporta dos tipos de actividades: primitivas y estructuradas.

Las **actividades primitivas** representan construcciones básicas, algunas de ellas son:

- **<receive>** : bloquea al proceso hasta que se recibe un determinado mensaje. Típicamente se utiliza para recibir un mensaje del cliente o un *callback* de un servicio Web *partner*.
- **<reply>** : envía un mensaje como respuesta a un mensaje recibido mediante *receive*.
- **<invoke>** : realiza una invocación sobre un servicio Web. Puede ser síncrona (*request-response*) o asíncrona (*one-way*).
- **<assign>**: asigna un valor a una variable.
- **<wait>**: para hacer que el proceso espere un cierto tiempo.
- **<throw>**: para indicar fallos y excepciones.

#### Receive

Una actividad *receive* especifica un *partnerLink*, un *portType* y una operación que puede ser invocada. Esta actividad juega un papel muy importante en el ciclo de vida de un proceso de negocio, ya que permite al proceso BPEL permanecer bloqueado a la espera de la llegada de un mensaje. Una de las formas de iniciar un proceso es utilizando una actividad *receive* con el atributo *createInstance* con el valor *yes* (por defecto, el valor de dicho atributo es *no*). Una actividad *receive* con *createInstance* con valor de *yes* debe ser una de las actividades iniciales de un proceso. Puede haber más de una de dichas actividades, en cuyo caso la primera de ellas que sea llamada iniciará el proceso. La sintaxis para *receive* es:

```
<receive partnerLink="ncname"
  portType="qname"
  operation="ncname"
  variable="ncname"
  createInstance="yes|no">
</receive>
```

#### Reply

Una actividad *reply* se utiliza para enviar una respuesta después de que se haya llamado a una actividad *receive*. La combinación de de una actividad *receive* con otra actividad *reply* crea una operación de tipo ***request-response***. La actividad *reply* se utiliza en una interacción síncrona, y especifica el mismo *partner*, *port type* y operación que la actividad *receive* que invocó al proceso. Con la actividad *reply* es posible transferir datos mediante una variable. La sintaxis de una actividad *reply* es la siguiente:

```
<reply partnerLink="ncname"
      portType="qname"
      operation="ncname"
      variable="ncname">
</reply>
```

## Invoke

Mediante una actividad *invoke* un proceso puede llamar a otro servicio Web que haya sido definido como *partner*. Esta actividad puede ser síncrona (operación de tipo *request-response*) o asíncrona (operación de tipo *one-way*).

Un uso asíncrono de esta actividad solamente necesita especificar una variable de entrada, ya que no hay una respuesta inmediata y por lo tanto no hay variable de salida. Un uso síncrono de *invoke* necesita tanto la variable de entrada como la de salida. La sintaxis es:

```
<invoke partnerLink="ncname"
      portType="qname"
      operation="ncname"
      inputVariable="ncname"
      outputVariable="ncname">
</invoke>
```

Las **actividades estructuradas** permiten combinar las actividades primitivas para especificar exactamente los pasos de los procesos de negocio. Algunas de ellas son:

- **<sequence>**: define un conjunto de actividades que se invocarán en forma de secuencia ordenada.
- **<flow>**: define un conjunto de actividades que se invocarán en paralelo.
- **<if>**: para la implementación actividades que se ejecutan dependiendo de una condición.
- **<while>**: para definir bucles.
- **<pick>**: hace que el proceso espere la llegada de algún evento y realice una determinada actividad (o conjunto de actividades), en función del evento. Básicamente se permiten dos tipos de eventos: eventos de mensaje (manejados con la etiqueta **<onMessage>**) y eventos de alarma (manejados con la etiqueta **>onAlarm**).

## 5.4. Relación de BPEL con BPMN

No hay una notación gráfica estándar para WS-BPEL. Algunos vendedores han inventado

sus propias notaciones. Estas notaciones se benefician del hecho de que la mayoría de construcciones BPEL están estructuradas en forma de bloques. Esta característica permite una representación visual directa de las descripciones de los procesos BPEL en forma de *estructurogramas*, que tienen un estilo con reminiscencias de los diagramas de Nassi-Shneiderman.

Por otro lado se ha propuesto el uso de un lenguaje de modelado de procesos sustancialmente diferente a BPEL, denominado *Business Process Modeling Notation* (BPMN), como un *front-end* gráfico para capturar las descripciones de los procesos BPEL. Se han propuesto numerosas herramientas que implementan el mapeado de BPMN a BPEL, como por ejemplo la herramienta *open-source* BPMN2BPEL. Sin embargo, el desarrollo de estas herramientas ha expuesto las diferencias fundamentales entre BPMN y BPEL, lo que hace muy difícil, y en algunos casos imposible, el generar código BPEL legible a partir de modelos BPMN. Más difícil todavía es el problema de generar código BPEL a partir de diagramas BPMN y mantener el BPMN original y el código BPEL sincronizado, en el sentido de que una modificación en uno de ellos se propague al otro.

## 5.5. Pasos para desarrollar un proceso de negocio con BPEL

---

Para desarrollar un proceso de negocio con BPEL vamos a seguir los siguientes pasos:

1. Conocer los servicios Web implicados
2. Definir el WSDL del proceso BPEL
3. Definir los *partner link types*
4. Desarrollar el proceso BPEL:
  - 4.1 Definir los *partner links*
  - 4.2 Declarar las variables
  - 4.3 Escribir la definición de la lógica del proceso

### Conocer con los servicios Web implicados

Antes de comenzar a definir el proceso BPEL, tenemos que familiarizarnos con los servicios Web a los que invoca nuestro proceso de negocio. Estos servicios se denominan *servicios Web partners*. Para cada uno de ellos deberemos conocer los *port types* que proporcionan, y las operaciones y mensajes de entrada salida definidas para dichos *port types*.

### Definir el WSDL del proceso BPEL

A continuación tenemos que exponer el proceso BPEL como un servicio Web, es decir, definir su WSDL, declarando los *port types*, sus operaciones y mensajes de entrada y salida.

### Definir los *partner link types*

Como ya hemos indicado, los *partner link types* representan la interacción entre el

proceso BPEL y los *partners* implicados, que incluyen tanto a los servicios Web a los que invoca dicho proceso BPEL, como al cliente que invoca al proceso BPEL.

Idealmente, cada servicio Web debe definir los correspondientes *partner link types* en sus ficheros WSDL. Alternativamente, podríamos definir todos los *partner links types* en el WSDL del proceso BPEL, aunque esta aproximación no es recomendada ya que viola el principio de encapsulación.

Comprender los *partner link types* es crucial para desarrollar la especificación del proceso BPEL. Algunas veces resulta útil dibujar un diagrama con todas las interacciones. Una vez que se han definido los *partner link types* y sus roles correspondientes, pasamos a definir el proceso de negocio.

### Desarrollar el proceso BPEL

Ahora ya estamos en disposición de definir el proceso BPEL. Típicamente, un proceso BPEL espera un mensaje de entrada de un cliente, el cual comienza la ejecución del proceso de negocio.

El documento que define el proceso BPEL presenta la estructura que ya hemos indicado en apartados anteriores. Recordemos que debemos añadir:

- los espacios de nombres, tanto del proceso BPEL como de los servicios Web a los que éste llama,
- los *partner links*,
- las variables del proceso y
- el cuerpo principal, que especifica el orden en el que se invocan los servicios Web *partners*. Normalmente comienzan con una actividad `<sequence>`, que permite definir varias actividades que se ejecutarán de forma secuencial.

## 5.6. Despliegue y pruebas del proceso BPEL

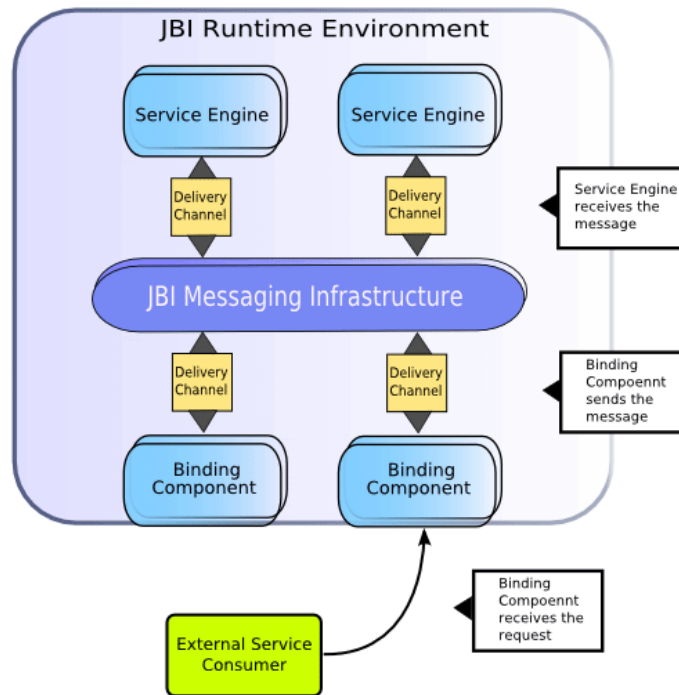
Una vez que hemos desarrollado el proceso BPEL, el último paso es su despliegue en una *BPEL engine*, seguido, si se desea, de las pruebas correspondientes antes de su utilización en producción.

Nosotros vamos a utilizar *NetBeans* como herramienta para desarrollar y desplegar procesos BPEL, por lo tanto, vamos a explicar cómo se realiza el proceso de despliegue y pruebas, utilizando *NetBeans 7.1*

### 5.6.1. Entorno de ejecución JBI (JBI Runtime Environment)

*Netbeans* utiliza el entorno de ejecución JBI (**JBI: Java Business Integration**). Dicho entorno de ejecución incluye varios componentes que interactúan utilizando un modelo de servicios. Dicho modelo está basado en el lenguaje de descripción de servicios web WSDL 2.0. Los componentes que suministran o consumen servicios dentro del entorno JBI son referenciados como máquinas de servicios (*Service Engines*). Uno de estos

componentes es la máquina de servicios BPEL (*BPEL Service Engine*), que proporciona servicios para ejecutar procesos de negocio. Los componentes que proporcionan acceso a los servicios que son externos al entorno JBI se denominan *Binding Components*.



Entorno de ejecución JBI.

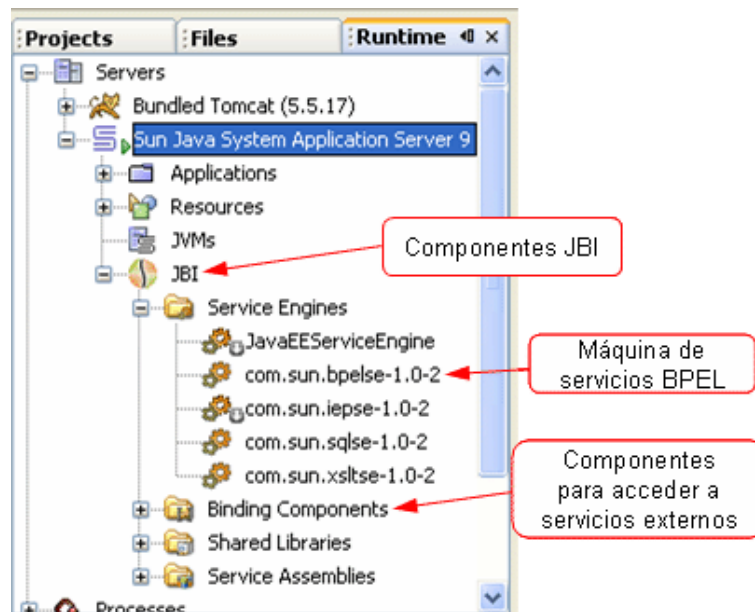
Los componentes JBI se instalan formando parte del servidor de aplicaciones Glassfish.v2x. Concretamente vamos a utilizar la versión 2.1.1 del servidor de aplicaciones Glassfish, junto con Glasfish ESB versión 2.2.

Glassfish ESB está formado por componentes que dan soporte al diseño (*design-time components*), y que se ejecutan dentro del IDE de NetBeans, y componentes que dan soporte a la ejecución (*runtime components*), que se ejecutan en el servidor de aplicaciones Glassfish.

La instalación estándar de Glassfish ESB incluye una máquina de servicios BPEL y un componente de enlazado Http, que se ejecutan en el servidor de aplicaciones, en el meta-contenedor JBI. Entre los componentes de Glassfish ESB que se ejecutan como parte del IDE de NetBeans se incluyen el editor WSDL, el editor de ensamblado de servicios (*CASA editor: Composite Application Service Assembly*). Éste último permite "ensamblar" manualmente los servicios. Esto puede ser necesario en el caso de aplicaciones SOA complejas, para las que la configuración por defecto que genera automáticamente NetBeans en las correspondientes Composite Applications no sea suficiente. Por ejemplo, podemos utilizar este editor para añadir/reemplazar elementos concretos WSDL (como ports, o bindings) que no se han especificado, cambiar la configuración por defecto del despliegue o incluir conexiones con *endpoints* de servicios

externos que se despliegan de forma separada en otras aplicaciones.

Para poder ver qué componentes JBI tenemos instalados o desplegados, desde la ventana *Services*, expandimos el nodo *Servers* y el nodo *GlassFish v2.x*, y dentro de él, el nodo JBI. Si el nodo JBI no es visible, necesitaremos arrancar el servidor de aplicaciones. Para ello pincharemos con el botón derecho sobre el nodo *GlassFish v2.x* y elegimos *Start* en el menú emergente.



Componentes JBI en Glassfish v2.x.

### 5.6.2. BPEL Service Engine

La máquina de servicios BPEL es un componente JBI (que satisface el estándar JSR 208) que proporciona servicios para ejecutar procesos de negocio desarrollados con WS-BPEL 2.0. Para desplegar un proceso BPEL, necesitamos añadirlo como un módulo JBI en un proyecto de composición de aplicaciones (*Composite Application project*).

La máquina de servicios BPEL arranca juntamente con el servidor de aplicaciones. Por lo que para desplegar y ejecutar procesos BPEL será necesario tener en marcha el servidor de aplicaciones. Cuando el servidor está en marcha, en la ventana *Services* del IDE de *NetBeans*, observaremos un distintivo de color verde al lado del nodo *GlassFish v2.x*.

La máquina de servicios BPEL está representada como el componente JBI *sun-bpel-engine* del servidor de aplicaciones.

### 5.6.3. Proyecto de aplicaciones compuestas

El término *composite application* se utiliza para referirse a aplicaciones que en lugar de



ser desarrolladas desde cero, son "ensambladas" a partir de servicios disponibles en una arquitectura SOA. Un proyecto de aplicaciones compuestas (*Composite Application project*) se utiliza para crear un ensamblado de servicios (*Service Assembly*) que puede desplegarse en el servidor de aplicaciones como un componente JBI.

Un proyecto BPEL no es directamente desplegable. Primero debemos añadir dicho proyecto BPEL, como un módulo JBI, en un proyecto *Composite Application*. A continuación podremos desplegar el proyecto *Composite Application* en la máquina de servicios BPEL. Al desplegar el proyecto hacemos que el ensamblado de servicios esté accesible para el servidor de aplicaciones, permitiendo así que sus unidades de servicios se ejecuten.

Para desplegar un proyecto *Composite Application* elegiremos la opción *Deploy Project* en el menú emergente que aparece al pinchar con el botón derecho sobre el nodo *Composite Application project*. La acción de desplegar el *Composite Application project* compila los ficheros de dicho proyecto, empaqueta el proceso BPEL compilado y los artefactos de servicios Web relacionados (incluyendo los ficheros WSDL y XSD) en un archivo, y lo despliega en el servidor de aplicaciones.

## 5.7. Creación y ejecución de casos de prueba

Dentro de un proyecto *Composite Application* que incluya un módulo JBI podemos también crear y ejecutar casos de prueba sobre el proceso BPEL, (que habrá sido previamente desplegado como un componente JBI, concretamente un *Service Assembly*, en el servidor de aplicaciones).

Los casos de prueba actúan como servicios *partner* remotos que envían mensajes SOAP al *runtime* de la máquina de servicios BPEL (*Service Engine BPEL*).

De forma simple, el proceso de interacción para las pruebas es el siguiente: el *runtime* de la máquina de servicios BPEL recibe un mensaje SOAP, crea una instancia del proceso BPEL y comienza a ejecutar dicha instancia. Un proceso BPEL puede tener muchas instancias en ejecución. El *runtime* de la máquina de servicios BPEL recibe un mensaje y, mediante correlación, lo enruta a la instancia apropiada del proceso. Si no existe todavía ninguna instancia, se crea una nueva.

Para crear y obtener los resultados de las pruebas, deberemos hacer lo siguiente:

- Añadir un caso de prueba y enlazarlo con una operación BPEL.
- Determinar las propiedades de la prueba (como por ejemplo el número de segundos que debe durar como máximo la ejecución de la prueba).
- Modificar convenientemente las entradas de las pruebas, para ello editaremos el fichero *Input.xml* según nos interese.
- Ejecutar las pruebas. El resultado de la ejecución se comparará con el resultado esperado almacenado en el fichero *Output.xml*. Si ambos resultados son diferentes, se habrá detectado un error.

En los ejercicios de la sesión veremos un ejemplo práctico de despliegue y pruebas de un proyecto BPEL.

Observaciones sobre los resultados de las pruebas:

- La primera vez que ejecutemos las pruebas obtendremos como resultado la indicación de que la prueba ha fallado (se ha detectado un error). La salida producida no coincidirá con el resultado almacenado en el fichero *Output.xml* (que inicialmente estará vacío). Al ejecutar la prueba la primera vez, el contenido de *Output.xml* será reemplazado por la salida generada en esta primera ejecución.
- Si ejecutamos la prueba de nuevo sin cambiar la entrada, las ejecuciones siguientes informarán del éxito de la prueba, ya que la salida será idéntica al contenido de *Output.xml*.
- Si cambiamos el valor de la entrada en *Input.xml* y volvemos a ejecutar la prueba, entonces:
  - Si la propiedad *feature-status* tiene como valor asignado *progress*, entonces la prueba indica éxito incluso aunque no coincida el resultado con *Output.xml*.
  - Si la propiedad *feature-status* tiene como valor asignado *done*, entonces la prueba indica fallo si no coincide el resultado y guarda el resultado obtenido.
  - Si pinchamos con el botón derecho del ratón sobre el caso de prueba y seleccionamos *Diff* sobre el menú emergente, se muestra la diferencia entre la última salida y los contenidos de *Output.xml*.
  - En cada ejecución posterior a la primera, y asumiendo que *Output.xml* ya no está nunca vacío, se preserva su contenido. Es decir, una salida previa, nunca se sobrescribe con nuevos resultados (con la excepción ya comentada de la primera ejecución).
  - Para ver los resultados de pruebas anteriores, podemos elegir de la lista desplegable el fichero *Actual\_yymmddhhmmss.xml* correspondiente, y pulsar en el botón *Refresh*.

Podemos ver las propiedades asociadas a los tests, situándonos sobre un nodo de test de la *Composite Application* y pulsando con el botón derecho, elegimos "Properties". Una de las propiedades es *feature-status*, que acabamos de comentar. Otras propiedades son: descripción del caso de prueba, URL del wsdl a probar, nombre de los ficheros de entrada y de salida de las pruebas, ...

## 6. Ejercicios de Orquestación de servicios BPEL

Vamos a crear desarrollar, desplegar y probar un sencillo servicio Bpel utilizando *Netbeans*. Para ello seguiremos los pasos que hemos indicado en esta sesión. En este caso, se trata de un proceso BPEL síncrono que simplemente recibe un mensaje de entrada y, en función de su contenido, devuelve una respuesta u otra al cliente. Concretamente, el mensaje de entrada consiste en una cadena de caracteres. Si dicha cadena tiene el valor

El cliente, después de invocar al proceso BPEL, permanece bloqueado hasta que el proceso termine y devuelva el resultado. (1,5p)

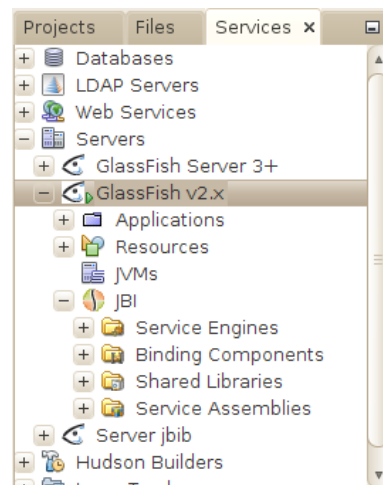
### 6.1. Pasos previos con Netbeans

En esta sesión y en la siguiente, vamos a utilizar una versión de Netbeans 7.1 que incluye los plugins necesarios para poder crear y editar proyectos BPEL, así como proyectos *Composite Application*, necesarios para poder desplegar nuestro proyecto BPEL como un componente JBI en el servidor de aplicaciones. El IDE con la que vamos a trabajar está disponible en la máquina virtual en:

```
/opt/SOA/bin/netbeans
```

Tendréis que instalar el **plugin de JUnit** para que puedan ejecutarse los *tests* de las *Composite Applications*. Para instalar dicho plugin, hay que hacerlo desde *Desde Tools->Plugins-Available Plugins*

En la pestaña "Servers" podéis ver que tenemos disponible un segundo servidor de **glassfish (versión 2.x)**. Ésta es la versión que vamos a utilizar para trabajar con BPEL. El servidor de glassfish 2.x tiene instalado el **runtime de JBI**, necesario para poder desplegar nuestro proyecto BPEL. Lo primero que haremos será poner el marcha dicho servidor. Entonces podremos ver el componente JBI, que contiene las máquinas de servicios (*service engines*), componentes de enlazado (*binding components*) y los ensamblados de servicios (*service assemblies*)



Servidor Glassfish v2.x con en runtime JBI

Podemos abrir la consola de administración del servidor de aplicaciones en la dirección:

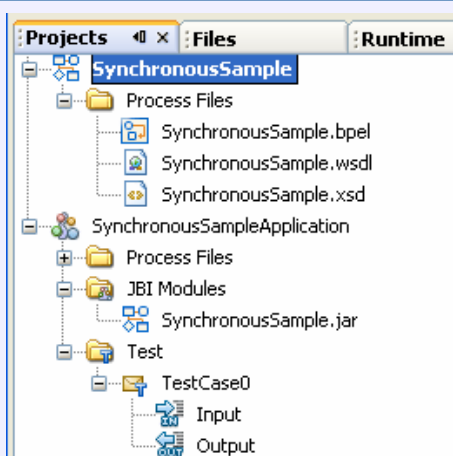
`http://localhost:4848`

Por defecto, los datos para acceder a la consola son:

User Name: admin  
Password: adminadmin

## 6.2. Creamos el proyecto BPEL

La mejor forma de familiarizarnos con la construcción de diagramas BPEL es crear un proyecto ejemplo (*sample project*). *Netbeans* permite generar dos tipos de proyectos ejemplo: síncrono y asíncrono. El IDE genera un proyecto esqueleto, con los ficheros *wsdl* y *xsd* que podremos modificar a nuestra conveniencia.



Plantilla del proyecto BPEL esqueleto síncrono generado

Para crear un nuevo proyecto ejemplo BPEL:

- Elegimos *File->New Project*.
- En la lista *Categories*, expandimos el nodo *Samples* y elegimos *Service Oriented Architecture*.
- En la lista *Projects*, seleccionamos el proyecto *Synchronous BPEL process*.
- Especificamos el nombre y la ubicación del proyecto. Podemos dejar como nombre del proyecto *SynchronousSample*.

Cuando creamos un proyecto BPEL ejemplo, automáticamente se genera el proyecto de aplicaciones compuestas, y automáticamente se añade el proyecto del módulo BPEL como un módulo JBI en el proyecto *Composite Application*.

El proyecto BPEL tiene una serie de propiedades, a las que podemos acceder pulsando con el botón derecho sobre el nodo del proyecto (en la ventana *Projects*), y seleccionando *Properties* del menú emergente. En este caso, nos aparecerá un cuadro en el que podemos acceder a varias páginas:

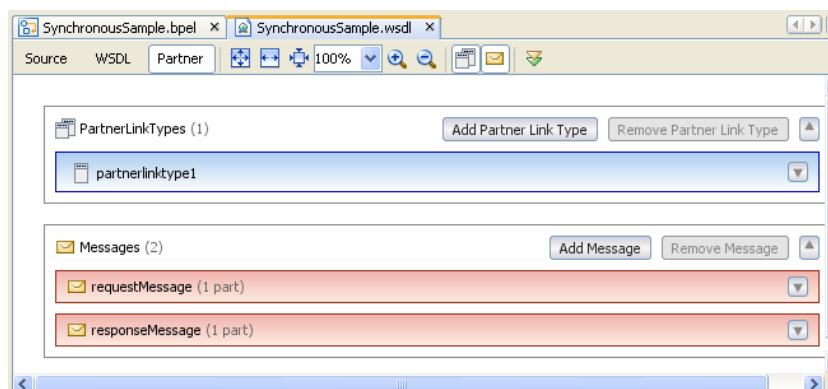
- **General:** muestra la ruta de directorios de los ficheros del proyecto, entre otros elementos
- **Referencias del proyecto:** muestra otros proyectos BPEL referenciados por nuestro proyecto BPEL
- **Catálogo XML:** muestra las entradas del catálogo XML usado en el proyecto BPEL. Los catálogos XML proporcionan información de mapeo entre una entidad externa en un documento XML y la localización real del documento que está siendo referenciado.

### 6.3. WSDL y esquema de nombres del proyecto BPEL

En este caso, al utilizar una plantilla para el proceso BPEL síncrono, *Netbeans* ha creado por nosotros el fichero WSDL y el esquema de nombres.

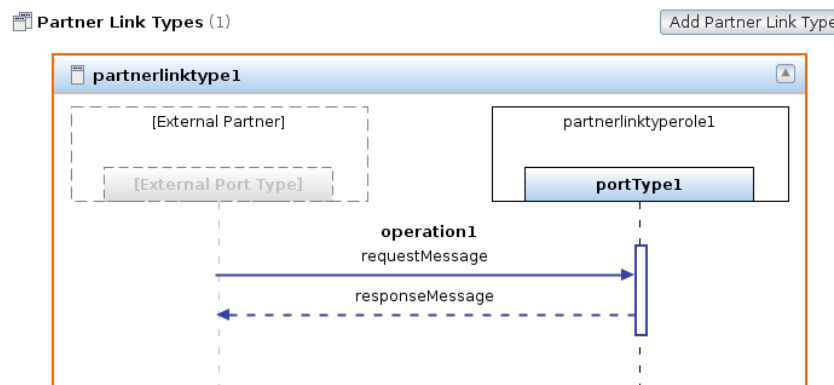
Los ficheros WSDL pueden visualizarse de tres formas: podemos ver el código fuente (*Source view*), podemos verlo en forma de árbol de componentes (*WSDL view*), o de forma gráfica (*Partner view*).

La siguiente figura muestra la vista gráfica del fichero WSDL creado:



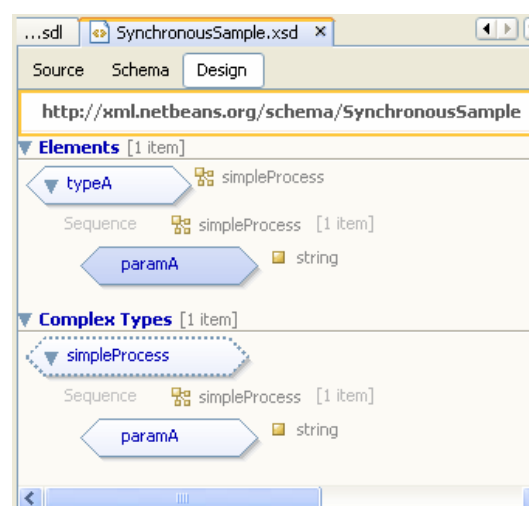
Vista gráfica del fichero WSDL generado

Podemos ver que el fichero WSDL contiene una definición de *partnerLinkType*, que define el rol *partnerlinktyperole1*, asociado al *portType portType1*. También hay definidos dos mensajes: uno de entrada (*requestMessage*), y otro de salida (*responseMessage*), que contienen una cadena de caracteres.



Definición de partnerlinktype en el WSDL

El fichero de definición de esquema es opcional en un proyecto BPEL. En este caso *Netbeans* también lo ha generado por nosotros. Al igual que ocurre con los ficheros WSDL, los ficheros de esquema tienen tres vistas diferentes: código fuente (*Source view*), árbol de componentes (*Schema view*), y vista gráfica (*Design view*). En la siguiente figura se muestra la vista gráfica del fichero *xsd* creado:



Vista gráfica del fichero xsd generado

Podemos ver que se ha definido el tipo *typeA*, que es un elemento del tipo complejo *simpleProcess*, que a su vez es una cadena de caracteres

En caso de que tuviésemos que crear nosotros los ficheros WSDL y de esquema, el proceso que se sigue es similar en ambos casos, tal y como ya hemos visto en sesiones anteriores:

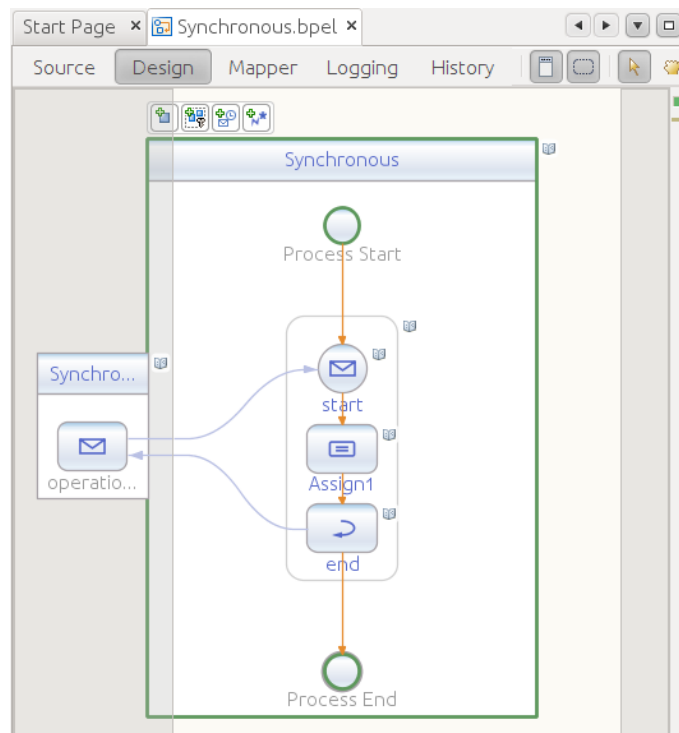
- Crear un nuevo documento: pinchamos con el botón derecho del ratón sobre el nodo *Process Files* y elegimos *New->WSDL Document*, o *New->Other/XML/XML Schema*.
- Desde la vista gráfica, añadimos componentes arrastrándolos desde la ventana *Palette*

- a la vista gráfica.
- Desde la vista de árbol, añadimos componentes y pinchamos con el botón derecho sobre nodo del tipo de componente que queremos añadir, y elegimos el opción *Add ...* correspondiente del menú emergente
- Desde la vista de fuente, escribimos el código en el editor de texto

## 6.4. Lógica del proceso BPEL

*Netbeans* ha creado un proceso BPEL síncrono muy sencillo, que podremos modificar según nos interese. En el caso de que tuviésemos que crearlo nosotros, tendríamos que hacerlo pinchando con el botón derecho del ratón sobre el nodo *Process Files* y después elegir *New->BPEL Process* en el menú emergente.

El proceso BPEL presenta tres vistas: código fuente (*Source*), vista gráfica (*Design*), y vista de mapeado (*Mapper*). Utilizaremos esta última vista más adelante. La **vista de diseño** (o vista gráfica) que se ha generado de forma automática es la siguiente:



Vista de diseño del proceso BPEL síncrono generado

En el caso de que tuviésemos que crear nosotros el proceso BPEL desde cero, tendríamos que ir añadiendo elementos arrastrándolos desde la ventana *Palette* hasta la vista de diseño del proceso BPEL.

Paleta de componentes BPEL

La **paleta de componentes** nos muestra los elementos que podemos insertar en el proceso BPEL, clasificados en tres categorías:

- Servicios Web: en donde encontramos las actividades *Invoke*, *Receive*, *Reply* y el componente *PartnerLink*
- Actividades básicas: *Assign*, *Empty*, *Wait*, *Throw* y *Exit*
- Actividades estructuradas: *If*, *While*, *Sequence*,...

A continuación mostramos el **código fuente** generado por *Netbeans*:

```
<?xml version="1.0" encoding="UTF-8"?>
<process
  name="SynchronousSample"
  targetNamespace="http://enterprise.netbeans.org/bpel/
    SynchronousSample/SynchronousSample_1"
  xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:bpws="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
  xmlns:wsdlNS="http://enterprise.netbeans.org/bpel/
    SynchronousSample/SynchronousSample_1"
  xmlns:ns1="http://localhost/SynchronousSample/SynchronousSample"

  <import location="SynchronousSample.xsd"
    importType="http://www.w3.org/2001/XMLSchema"
    namespace="http://xml.netbeans.org/schema/SynchronousSample" />

  <import namespace=
    "http://localhost/SynchronousSample/SynchronousSample"
    location="SynchronousSample.wsdl"
    importType="http://schemas.xmlsoap.org/wsdl/" />

  <partnerLinks>
    <partnerLink name="SynchronousSample"
      partnerLinkType="ns1:partnerlinktype1"
      myRole="partnerlinktyperole1" />
  </partnerLinks>

  <variables>
    <variable name="outputVar" messageType="ns1:responseMessage" />
    <variable name="inputVar" messageType="ns1:requestMessage" />
  </variables>

  <sequence>
    <receive name="start"
```



```

        partnerLink="SynchronousSample"
        operation="operation1"
        portType="ns1:portType1"
        variable="inputVar" createInstance="yes"/>
    <assign name="Assign1">
        <copy>
            <from>${inputVar.inputType/paramA}</from>
            <to>${outputVar.resultType/paramA}</to>
        </copy>
    </assign>
    <reply name="end"
        partnerLink="SynchronousSample"
        operation="operation1"
        portType="ns1:portType1"
        variable="outputVar"/>
</sequence>
</process>

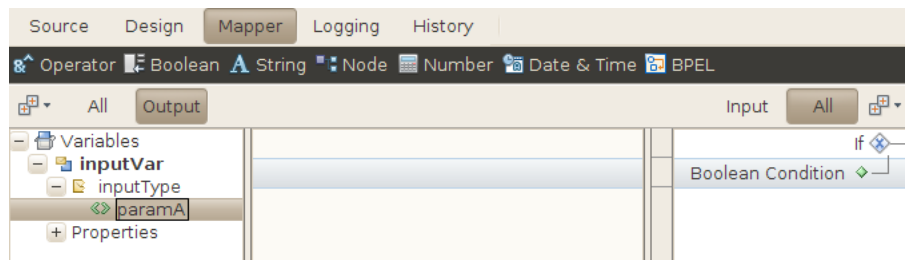
```

Según el código fuente anterior, podemos ver que la lógica del proceso de negocio consiste en esperar a recibir una llamada del *partner link* cliente sobre la operación *operation1*, con el mensaje de entrada *inputVar*. Una vez que el proceso BPEL ha sido invocado por un cliente, asignamos el valor del mensaje de entrada en el mensaje de salida, denominado *outputVar* y se lo enviamos al cliente utilizando la actividad *reply*

### Añadimos una actividad *if*

Vamos a modificar este proceso inicial **añadiendo una actividad *if*** en la vista de diseño. Para ello:

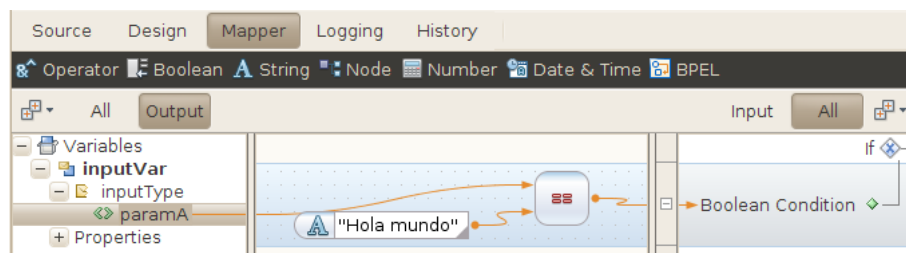
- Elegimos la actividad estructurada *if* de la paleta de componentes y la arrastramos a la vista de diseño del proceso BPEL, situándolo entre las actividades *Start* y *Assign*. Esto nos creará una actividad *if* denominada *if1* en la vista de diseño. (El IDE nos indica dónde podemos situar los elementos que arrastramos mediante dos círculos concéntricos, coloreando el más pequeño).
- Pinchamos dos veces sobre la actividad *if1* y aparecerá la ventana de mapeado (*BPEL Mapper*). Otra forma de acceder a la ventana de mapeado es, teniendo el componente *if1* seleccionado (podemos ver que un componente está seleccionado porque aparece enmarcado en un rectángulo verde con bordes redondeados), simplemente cambiamos a la vista *Mapper*. Usaremos la ventana de mapeado para definir una condición booleana.
- En el panel de la izquierda vemos las variables de nuestro proceso BPEL. En el panel de la derecha nos aparece una condición booleana asociada a la actividad *if*. Seleccionamos la condición booleana con el botón izquierdo del ratón, así como el elemento *paramA* de la variable *inputVar*



Ventana de mapeado de BPEL

- En la barra de menús, pinchamos en *Operator* y seleccionamos la función *EQUAL* en el menú emergente. La función *equal* aparecerá en la parte central de la ventana de mapeado. Dicha función tiene asociados dos conectores de entrada y uno de salida (representados con forma de puntas de flecha).
- En la barra de menús, pinchamos sobre la función *String* y seleccionamos *String Literal* en la lista desplegable. Aparecerá una caja con la etiqueta *String literal* en el panel central.
- Tecleamos 'Hola mundo' en la función *string literal* y pulsamos retorno de carro.
- Pinchamos con el ratón sobre *paramA* y lo arrastramos hasta el segundo conector de entrada de la función *equal*.
- Pinchamos sobre el conector de salida de *String Literal* y lo arrastramos hasta el otro conector de entrada de *Equal*.
- Desde la función *equal*, arrastramos el conector de salida hasta *Boolean Condition* en el panel de la derecha. Si no se muestra el nodo *Boolean Condition*, simplemente haremos doble click sobre la condición *if1*.

La siguiente figura muestra el resultado de realizar las acciones anteriores:

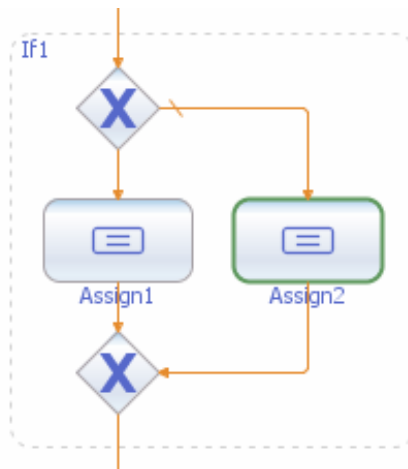


Ventana de mapeado de BPEL a la que hemos añadido una condición

### Añadimos una actividad *Assign*

En la vista de diseño, arrastramos la actividad *Assign1* existente hasta la actividad *if*. Colocamos esta actividad entre rombos con el icono X en el área de la actividad *if1*

A continuación, elegimos la actividad *Assign* de la sección de actividades básicas de la paleta, y la arrastramos a la derecha de la actividad *Assign1* existente.

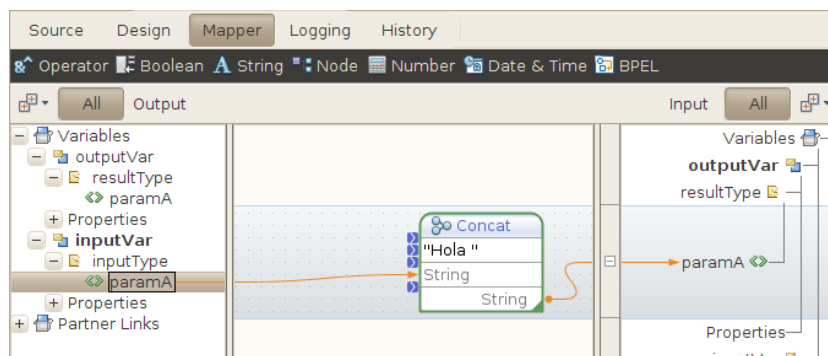


Resultado de añadir una segunda asignación

Seleccionamos la nueva actividad *Assign2* y volvemos a utilizar la ventana de mapeado de la siguiente forma:

- Expandimos los nodos *Variables - inputVar - inputType* y *Variables - outputVar - resultType* (en el panel de la izquierda). Veremos que en el panel de la derecha el nodo *Variables* no se puede expandir (está vacío). Para poder ver los sub-nodos hacemos doble click sobre el nodo *variables*, y nos aparecerá el sub-nodo *outputVar*, si de nuevo hacemos doble click sobre *outputVar*, nos aparecerá el sub-nodo *resultType*, y así sucesivamente hasta que aparezca el nodo *paramA*
- Seleccionamos las variables *paramA* tanto del panel de la derecha como el de la izquierda. Seguidamente seleccionamos el grupo de funciones de *String* y elegimos (pinchamos) sobre la función *concat*. Aparecerá la función *Concat* en el panel central
- Hacemos doble click sobre el primer campo de la función *Concat*, tecleamos 'Hola ' y pulsamos el retorno de carro
- Con el botón izquierdo, arrastramos *paramA* del panel de la izquierda hasta el segundo campo de la función *concat*
- Arrastramos el ratón desde el conector de salida de la función *concat* hasta *paramA* (en el panel de la derecha). Esto concatena la cadena 'Hola ' con la entrada y copia el resultado en la salida

Después de realizar los pasos anteriores, la ventana de mapeado debe presentar el siguiente aspecto:



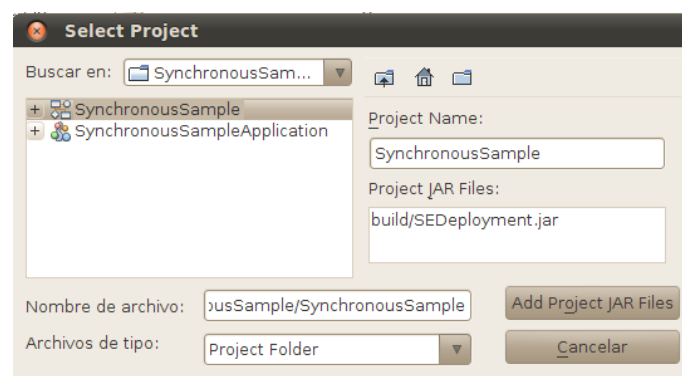
Concatenamos el resultado

## 6.5. Desplegamos el proyecto en el servidor de aplicaciones

Nuestro proyecto Bpel no es directamente desplegable en el servidor de aplicaciones. Para poder hacer el despliegue tenemos que incluir el "jar" de nuestro proyecto como un módulo JBI en una Composite Application (en nuestro caso será el proyecto *SynchronousSampleApplication*) la composite application contendrá un ensamblado de servicios que sí podremos desplegar en el servidor de aplicaciones.

Para desplegar el proyecto realizaremos los siguientes pasos:

- Compilamos nuestro proyecto Bpel (*SynchronousSample*) pinchando con botón derecho y seleccionando *Build*
- Pinchamos con el botón derecho sobre el proyecto *SynchronousSampleApplication* y seleccionamos *Add JBI module...*. Elegimos el proyecto *SynchronousSample* y pinchamos sobre *"Add Project JAR files"*

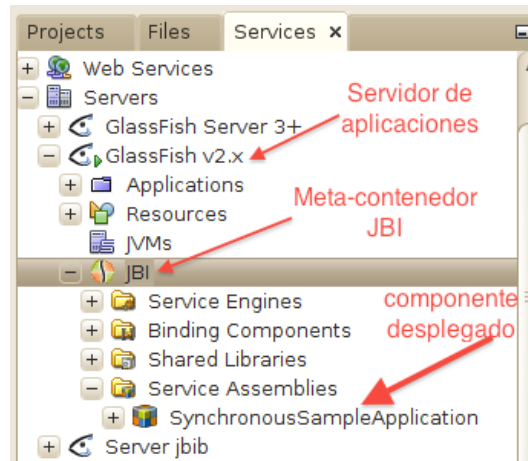


Añadimos el módulo JBI a la Composite Application

- Pinchamos con el botón derecho sobre el proyecto *SynchronousSampleApplication* y elegimos *Deploy Project*, seleccionando *Glassfish v2.x*. Aparecerá el mensaje *BUILD SUCCESSFUL* en la ventana *Output*

Podemos ver el resultado del despliegue en la pestaña *Services*, en el servidor de

aplicaciones *Glassfish v2.x*. El componente *SynchronousSampleApplication* se ha desplegado como un ensamblado de servicios dentro del meta-contenedor JBI.



Componente desplegado como un módulo JBI

## 6.6. Creamos un conductor de pruebas

Vamos a añadir casos de prueba para comprobar el funcionamiento del proceso BPEL que hemos creado. Para ello seguiremos los siguientes pasos:

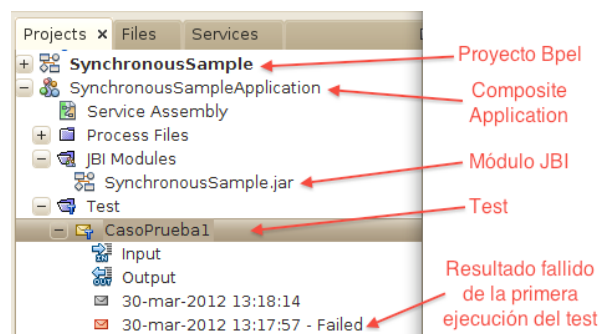
- En la ventana *Projects*, expandimos el nodo del proyecto *SynchronousSampleApplication*, pinchamos con el botón derecho sobre el nodo *Test*, y elegimos *New Test Case* del menú emergente.
- Vamos a darle el nombre *CasoPrueba1* y pinchamos en *Next*.
- Expandimos *SynchronousSample - Source Packages* y seleccionamos *SynchronousSample.wsdl*. Pinchamos en *Next*
- Seleccionamos *operation1* y pinchamos en *Finish*. Ahora, en el árbol del proyecto, bajo el nodo *Test* se ha creado la carpeta *CasoPrueba1*, que contiene dos ficheros: *Input* y *Output*
- Pinchamos dos veces sobre *Input* y modificamos su contenido de la siguiente forma:
  - Localiza la siguiente línea en los contenidos del cuerpo:
 

```
<syn:paramA>?string?<syn:paramA>
```
  - Reemplaza *?string?* con *Colega*
  - Grabamos los cambios desde *File->Save*
- Pinchamos dos veces sobre *Output.xml* para examinar los contenidos. Antes de ejecutar las pruebas, este fichero está vacío. Cada vez que ejecutemos las pruebas, la salida actual se compara con los contenidos de *Output*. La entrada (*input*) se copia sobre *Output* cuando *Output* está vacío.

## 6.7. Ejecutamos las pruebas sobre SynchronousSampleApplication

En la ventana *Projects*, expandimos *SynchronousSampleApplication - Test - CasoPrueba1*. El nodo *CasoPrueba1* contiene dos ficheros XML: *Input* para la entrada, y *Output* para la salida esperada. Como ya hemos indicado, cada vez que se ejecuta el caso de prueba, la salida actual se compara con el contenido de *Output*. Si coinciden, entonces superamos la prueba y veremos en la ventana *Output* el mensaje: *Nombre\_Caso\_Prueba passed*. Si no coinciden, aparecerá el mensaje *Nombre\_Caso\_Prueba FAILED*.

Pinchamos con el botón derecho sobre el nodo *CasoPrueba1*, y elegimos *Run* en el menú emergente. Nos fijamos en la ventana *Output* y veremos que el test se ha completado y que no pasamos el test, ya que aparece el mensaje *CasoPrueba1 FAILED*. En este caso, como el fichero *Output* está vacío, se nos pregunta si queremos sobrescribirlo. Contestamos que sí. Después de la primera ejecución, el fichero *Output.xml* ya no está vacío, por lo que su contenido será preservado y no será sobrescrito por el nuevo resultado. Si ejecutamos de nuevo el test, podremos ver el mensaje: *CasoPrueba1 passed*.



Casos de prueba para nuestro proyecto Bpel

## 6.8. Cambios en la lógica de negocio

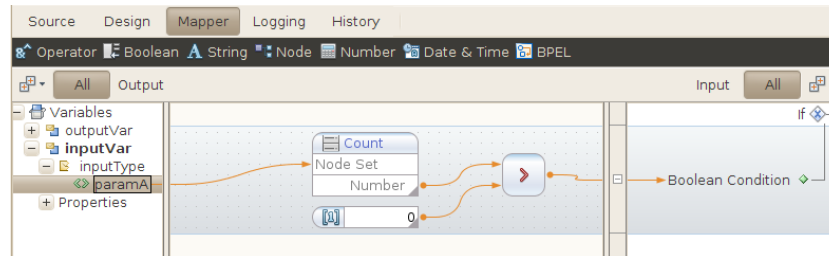
Modifica la lógica del proceso bpel de forma que: si la cadena de entrada es null, devuelva el mensaje de error "ERROR: Valor null". Si la cadena de entrada no es null y es "Hola mundo", entonces debe devolver la misma cadena como salida, y si no es igual a "Hola Mundo" debe devolver "Hola cadena", siendo "cadena" el valor de la cadena de entrada. (0,75p)

Añade dos casos de prueba nuevos, con nombres *TestHolaMundo* y *TestNull*. Para el primer test, el valor de la cadena de entrada debe ser "Hola mundo", para el segundo test, el valor de entrada debe ser "null". (0,75p)

### Notas de ayuda:

- Cuando añadamos el nuevo "if" en la vista de diseño, y pasemos a la vista de mapeado, en el panel de la derecha vemos que únicamente nos aparece el nodo "if". Si hacemos doble click sobre él, nos generará el nuevo sub-elemento "boolean condition"
- Una forma de comprobar si un elemento es "null" es comprobando si el número de

elementos del nodo correspondiente es cero (menú "Node", función "Count"). La siguiente figura ilustra la comprobación de que la cadena de caracteres de entrada no sea "null".



Comprobación de entrada con valor NULL

- Después de hacer las modificaciones en la lógica del proceso bpel, hay que compilar de nuevo el proyecto Bpel, borrar el componente JBI de la Composite Application (*SynchronousSample.jar*) con botón derecho seleccionando "Delete", a continuación añadimos de nuevo el componente JBI, y volvemos a desplegar *SynchronousSampleApplication*
- En el mensaje soap de entrada para el caso de prueba *TestNull*, simplemente "borra" la línea `<syn:paramA>?string?</syn:paramA>` del fichero *input.xml*

## 7. Procesos BPEL síncronos y asíncronos

Ya hemos visto que un proceso BPEL se "expone" como un servicio web. El proceso BPEL puede ser síncrono o asíncrono. Un proceso BPEL síncrono devuelve una respuesta al cliente inmediatamente después de procesar la petición y el cliente permanece bloqueado mientras se ejecuta el proceso BPEL.

Un proceso BPEL asíncrono no bloquea al cliente. Para devolver un resultado al cliente, un proceso asíncrono utiliza una llamada *"callback"*. Sin embargo, no es imprescindible que el proceso BPEL devuelva una respuesta.

Por lo tanto, uno de los requerimientos técnicos importantes a tener en cuenta cuando se diseñan procesos de negocio que implican la ejecución de múltiples servicios Web que se ejecutan durante periodos largos de tiempo, es la capacidad de poder invocar servicios de forma asíncrona.

La invocación de servicios de forma asíncrona es vital para alcanzar la fiabilidad, escalabilidad, y adaptabilidad requeridas hoy en día en entornos IT (*Information Technology*). Mediante un soporte asíncrono, un proceso de negocio puede invocar servicios Web de forma concurrente (en lugar de secuencialmente), de forma que se mejora el rendimiento. Por ejemplo, un sistema de compras podría necesitar interactuar con múltiples servicios Web de proveedores al mismo tiempo, para así buscar el proveedor que le ofrezca el precio más bajo y la fecha de entrega más temprana. Este soporte asíncrono para servicios Web puede conseguirse mediante técnicas de correlación.

En definitiva, la elección del tipo de proceso BPEL es muy importante. La mayoría de procesos del "mundo real" requieren largos periodos de ejecución, de forma que los modelaremos como asíncronos. Sin embargo, también hay muchos procesos que se ejecutan en un periodo relativamente corto de tiempo, o que necesitan que el cliente espere a que se procese la respuesta. En esta sesión explicaremos las diferencias de implementación entre los dos tipos de procesos (síncronos y asíncronos) y veremos algún ejemplo.

### 7.1. Invocación de servicios Web

En la sesión anterior hemos visto que un proceso BPEL se codifica como un documento XML utilizando el elemento raíz `<process>`. Dentro del elemento `<process>`, un proceso BPEL normalmente contendrá el elemento `<sequence>`. Dentro de la secuencia, el proceso primero esperará la llegada de un mensaje para comenzar la ejecución del proceso. Esto se modela con la construcción `<receive>`. Seguidamente el proceso invocará a los servicios Web relacionados, utilizando la construcción `<invoke>`. Tales invocaciones pueden realizarse secuencialmente o en paralelo. Si queremos realizar las llamadas secuencialmente simplemente escribiremos una sentencia `<invoke>` para cada



invocación y los servicios Web serán invocados en dicho orden. Esto se muestra en el siguiente código:

```
<!-- Invocación secuencial de servicios Web-->
<process ...>
  ...
  <sequence>
    <!-- espera a una petición para comenzar el proceso -->
    <receive.../>

    <!-- invoca un conjunto de servicios Web, de forma secuencial -->
    <invoke .../>
    <invoke .../>
    <invoke .../>
    ...
  </sequence>
</process>
```

Para invocar a servicios Web de forma concurrente, podemos utilizar la construcción `<flow>`. En el siguiente ejemplo, las tres operaciones `<invoke>` se ejecutarían concurrentemente.

```
<!-- Invocación concurrente de servicios Web-->
<process ...>
  ...
  <sequence>
    <!-- espera a una petición para comenzar el proceso -->
    <receive.../>

    <!-- invoca un conjunto de servicios Web, concurrentemente -->
    <flow>
      <invoke .../>
      <invoke .../>
      <invoke .../>
    </flow>
    ...
  </sequence>
</process>
```

También podemos combinar y anidar las construcciones `<sequence>` y `<flow>`, lo que nos permite definir varias secuencias que se ejecutarían concurrentemente. En el siguiente ejemplo hemos definido dos secuencias, una formada por tres invocaciones, y otra formada por dos. Ambas secuencias se ejecutarían concurrentemente:

```
<!-- Invocación concurrente de secuencias de servicios Web-->
<process ...>
  ...
  <sequence>
    <!-- espera a una petición para comenzar el proceso -->
    <receive.../>

    <!-- invoca dos secuencias concurrentemente -->
    <flow>
      <!-- ejecución secuencial -->
      <sequence>
        <invoke .../>
```

```

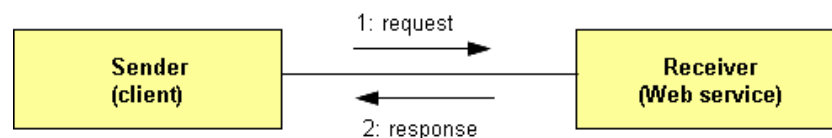
    <invoke .../>
    <invoke .../>
  </sequence>
  <!-- ejecución secuencial -->
  <sequence>
    <invoke .../>
    <invoke .../>
  </sequence>
</flow>
...
</sequence>
</process>

```

## 7.2. Invocación de servicios Web asíncronos

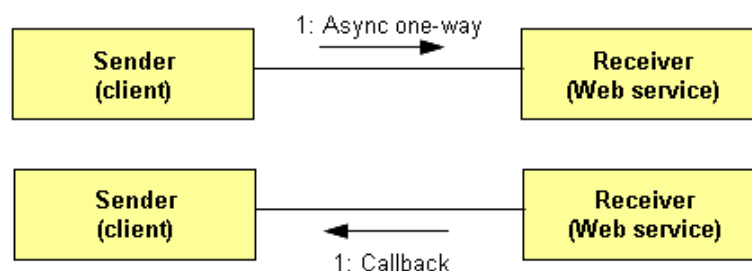
Hay dos tipos operaciones de servicios Web:

- **Operaciones síncronas *request-reply*:** En este caso enviamos una petición y esperamos el resultado. Normalmente las operaciones no necesitan demasiado tiempo de procesamiento, por lo que es razonable esperar hasta que nos llegue la respuesta. Esto se muestra en la siguiente Figura.



Operación síncrona (request/reply).

- **Operaciones asíncronas:** Normalmente, estas operaciones realizan un procesamiento que requiere un largo tiempo hasta su finalización. Por lo tanto, no bloquean al proceso que realizó la llamada. Si dichas operaciones requieren que los resultados se envíen de vuelta al cliente, se realizan *callbacks*, tal y como mostramos a continuación. A este tipo de operaciones se les denomina también operaciones *one-way*



Operación asíncrona (one-way).

Los *callbacks* normalmente necesitarán relacionarse con las peticiones originales, para así poder enviar la respuesta al cliente correcto. A este proceso lo llamaremos *correlación*, y lo explicaremos más adelante.

Utilizando la construcción `<invoke>` podemos invocar ambos tipos de operaciones: síncronas y asíncronas. Si invocamos una **operación síncrona**, el proceso de negocio se espera hasta recibir la respuesta (en este caso, tenemos que incluir el atributo con el mensaje de respuesta). No necesitamos utilizar una construcción especial para recuperar la respuesta.

Ejemplo de invocación síncrona:

```
<!-- invocación SINCRONA sobre el servicio Employee Travel Status -->
<invoke partnerLink="employeeTravelStatus"
        portType="emp:EmployeeTravelStatusPT"
        operation="EmployeeTravelStatus"
        inputVariable="EmployeeTravelStatusRequest"
        outputVariable="EmployeeTravelStatusResponse" />
```

Con las **operaciones asíncronas**, la construcción `<invoke>` solamente realiza la primera parte de la llamada, la que invoca la operación. Para recibir un resultado (si es que se devuelve al cliente), necesitamos utilizar una construcción separada `<receive>`. Con `<receive>`, el proceso de negocio esperará la respuesta. Entre las operaciones `<invoke>` y `<receive>` podríamos realizar cualquier otro procesamiento en lugar de esperar la respuesta, tal y como ocurre en una invocación síncrona. El esquema para una invocación de una operación asíncrona es:

```
<!-- Invocación asíncrona -->
<process ...>
  ...
  <sequence>
    <!-- espera a una petición para comenzar el proceso -->
    <receive.../>

    <!-- invoca una operación asíncrona -->
    <invoke .../>

    <!-- hacer algo -->

    <!-- esperar el Callback -->
    <receive.../>
    ...
  </sequence>
</process>
```

A continuación mostramos un ejemplo concreto de un uso asíncrono de la actividad *invoke*:

```
<!-- invocación ASINCRONA sobre el servicio American Airlines -->
<invoke partnerLink="AmericanAirlines"
        portType="aln:FlightAvailabilityPT"
        operation="FlightAvailability"
        inputVariable="FlightDetails" />

<receive partnerLink="AmericanAirlines"
        portType="aln:FlightCallbackPT"
        operation="FlightTicketCallback"
```

```
variable="FlightResponseAA" />
```

Igual que con las operaciones síncronas, podemos utilizar pares asíncronos `<invoke>/<receive>` dentro de elementos `<flow>` para realizar varias invocaciones concurrentes.

### 7.3. Procesos BPEL síncronos frente a procesos BPEL asíncronos

Los procesos BPEL difieren en el protocolo de intercambio de mensajes. Un **proceso BPEL síncrono** es aquel que devuelve al cliente los resultados de procesamiento de forma inmediata. El cliente se bloquea hasta que los resultados son devueltos. El proceso BPEL tendrá una operación de tipo *request-response*. Este tipo de proceso típicamente sigue la siguiente lógica y sintaxis:

```
<!-- Estructura de un proceso BPEL síncrono -->
<process ...>
  ...
  <sequence>
    <!-- espera a una petición para comenzar el proceso -->
    <receive.../>

    <!-- invoca a un conjunto de servicios relacionados -->
    ...

    <!-- devuelve una respuesta síncrona al cliente -->
    <reply .../>
    ...
  </sequence>
</process>
```

Para que un proceso BPEL sea síncrono, todos los servicios Web a los que invoca tienen que ser también síncronos.

Un proceso BPEL asíncrono no utiliza la cláusula `<reply>`. Si dicho proceso tiene que enviar una respuesta al cliente, utiliza la cláusula `<invoke>` para invocar a la operación *callback* sobre el *port type* del cliente. Recuerda que un proceso BPEL no necesita devolver nada. La estructura de un proceso BPEL asíncrono es:

```
<!-- Estructura de un proceso BPEL asíncrono -->
<process ...>
  ...
  <sequence>
    <!-- espera a una petición para comenzar el proceso -->
    <receive.../>

    <!-- invoca a un conjunto de servicios relacionados -->
    ...

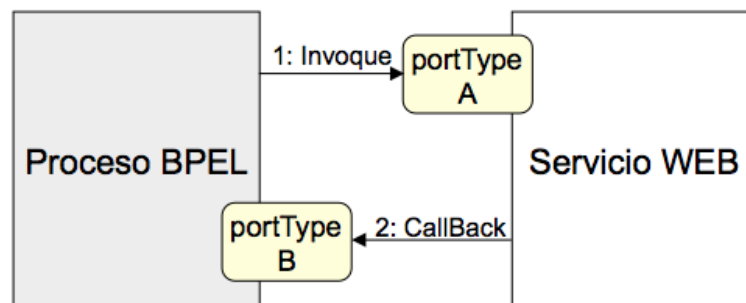
    <!-- invoca a un callback sobre el cliente (si es necesario) -->
    <invoke .../>
    ...
  </sequence>
</process>
```

Podemos utilizar un **proceso BPEL asíncrono** en un escenario en el que el calcular los resultados conlleva mucho tiempo. Cuando utilizamos un proceso BPEL asíncrono, es necesario que el cliente no se quede bloqueado en la llamada. En su lugar, el cliente implementa una interfaz *callback* y una vez que los resultados están disponibles, el proceso BPEL simplemente realiza una invocación *callback* sobre el cliente.

#### 7.4. Partner Link Types en procesos asíncronos

En la sesión anterior ya hemos comentado que los *partner links* (definido en el fichero del proceso BPEL) describen las relaciones con los *partners*, en las que éstos pueden: (a) invocar al proceso BPEL; (b) ser invocados por el proceso BPEL; (c) son invocados por el proceso y a su vez invocan al proceso BPEL.

La situación (c) se da en una invocación asíncrona. En este caso, un servicio Web ofrece un *portTypeA*, a través del cual el proceso BPEL invoca a las operaciones sobre dicho servicio. El proceso BPEL tiene también que proporcionar un *portType* (al que llamaremos *portTypeB*) a través del cual el servicio Web invoca a la operación *callback*. Esto se muestra en la siguiente figura:



Invocación de un servicio asíncrono.

Como acabamos de ver, para describir las situaciones en las que el servicio Web es invocado por el proceso BPEL y viceversa debemos definir una perspectiva concreta. Por ejemplo podemos seleccionar la perspectiva del proceso y describir el proceso como aquél que requiere el *portTypeA* sobre el servicio Web y que proporciona el *portTypeB* al servicio Web. Alternativamente, podemos seleccionar la perspectiva del servicio Web y describir al servicio Web como aquél que oferta el *portTypeA* al proceso BPEL y que requiere el *portTypeB* del proceso.

Los *partner link types* permiten modelar las relaciones entre un servicio Web y un proceso BPEL, declarando cómo interactúan las dos partes y lo que cada parte proporciona. El uso de los *partner link types* no requiere el utilizar una perspectiva concreta; en vez de eso, se definen *roles*. Un *partner link type* debe tener al menos un *rol* y puede tener como mucho dos *roles*. Para cada *rol* debemos especificar el *portType* que se usará para la interacción.

En el siguiente ejemplo, se define un *partnerLinkType* denominado *insuranceLT*. Éste define dos roles, el rol *insuranceService* y el rol *insuranceRequester*. El primero de ellos proporciona el *port type* *ComputeInsurancePremiumPT* desde el espacio de nombres *ins*. El rol *insuranceRequester* proporciona el *port type* *ComputeInsurancePremiumCallbackPT* desde el espacio de nombres *com*. Este último *port type* se utiliza para la operación *callback*. Esta declaración especifica los roles de servicio y *callback*:

```
<!-- partnerLinkType con roles servicio y callback; definido en WSDL-->
<partnerLinkType name="insuranceLT"
  xmlns="http://schemas.xmlsoap.org/ws/2003/05/partner-link/">

  <role name="insuranceService">
    <portType name="ins:ComputeInsurancePremiumPT" />
  </role>

  <role name="insuranceRequester">
    <portType name="com:ComputeInsurancePremiumCallbackPT" />
  </role>
</partnerLinkType>
```

Como ya hemos visto en la sesión anterior, cuando se utilizan operaciones síncronas solamente se utiliza un rol. Recordemos también que la definición del *partnerLinkType* forma parte del fichero WSDL (del proceso BPEL).

Para cada *partner link*, definido en el fichero que implementa el proceso BPEL (fichero con extensión *.bpel*), tenemos que indicar los siguientes atributos:

- **name:** Sirve como referencia para las iteracciones entre *partners*
- **partnerLinkType:** Define el tipo de *partner link*
- **myRole:** Indica el rol del proceso BPEL
- **partnerRole:** Indica el rol del *partner*

Definiremos los dos roles (*myRole* y *partnerRole*) solo si el *partnerLinkType* especifica dos roles. Volvamos al ejemplo anterior, en el que hemos definido el *partnerLinkType* *insuranceLT*. Para definir un *partnerLink* denominado *insurance*, caracterizado por el *partnerLinkType* *insuranceLT*, necesitamos especificar ambos roles, debido a que se trata de una relación asíncrona. El rol del proceso BPEL (*myRole*) se describe como *insuranceRequester*, y el rol del *partner* se describe como *insuranceService*. El código asociado se muestra a continuación:

```
<!-- definición de partnerLink en fichero BPEL-->
...
<partnerLinks>
  <partnerLink name="insurance"
    partnerLinkType="tns:insuranceLT"
    myRole="insuranceRequester"
    partnerRole="insuranceService" />
</partnerLinks>
...
```

## 7.5. Ciclo de vida de los procesos de negocio

---

Cada proceso BPEL tiene un ciclo de vida bien definido. Para comunicarse con los *partners* BPEL utiliza los servicios Web. Los servicios Web proporcionan un modo sin estado para la invocación de operaciones. Esto significa que un servicio Web no almacena la información dependiente del cliente entre invocaciones de operaciones. Por ejemplo, consideremos un carrito de la compra en el que un cliente utiliza una operación para añadir elementos en el carrito. Por supuesto, pueden haber varios clientes simultáneos utilizando el carrito de la compra a través del servicio Web. Sería deseable que cada cliente tuviese su propio carrito. Para conseguir esto utilizando servicios Web, cada cliente tendría que pasar su identidad en cada invocación de la operación *añadir*. Esto es debido a que el modelo de servicios Web es un modelo sin estado (un servicio Web no distingue entre clientes diferentes).

Para procesos de negocio, un modelo sin estado no es adecuado. Consideremos, por ejemplo, un escenario de viajes en el que un cliente envía una petición de viaje, a través de la cual se inicia el proceso de negocio. El proceso de negocio se comunica con varios servicios Web y envía en primer lugar una nota de confirmación del viaje al cliente. Más tarde envía una confirmación del hotel y una factura. El proceso de negocio tiene que recordar cada interacción para poder conocer a quién tiene que devolver los resultados.

A diferencia de los servicios Web, los procesos de negocio BPEL siguen un modelo con estado y soportan interacciones que llevan mucho tiempo con un ciclo de vida bien definido. Para cada interacción con el proceso, se crea una instancia del proceso. Por lo tanto podemos pensar en la definición del proceso BPEL como una plantilla para crear instancias de procesos. Esto es bastante similar a la relación clase-objeto, en donde las clases representan plantillas para crear objetos en tiempo de ejecución.

En BPEL no se crean instancias de forma explícita igual que en los lenguajes de programación (no hay un comando *new*, por ejemplo). En vez de eso, la creación se hace de forma implícita y tiene lugar cuando el proceso recibe el mensaje inicial que inicia el proceso. Esto puede ocurrir con las actividades `<receive>` o `<pick>`. Ambas proporcionan un atributo denominado *createInstance*. Si este atributo se fija a "yes", indica que cuando se ejecuta dicha actividad se crea una nueva instancia del proceso de negocio.

Un proceso de negocio puede terminar normal o anormalmente. Una terminación normal ocurre cuando todas las actividades del proceso se completan. Una terminación anormal tiene lugar cuando, o bien se produce un fallo dentro del ámbito del proceso, o una instancia del mismo termina de forma explícita utilizando la actividad `<terminate>`.

En procesos de negocio complejos, más de una actividad de inicio del proceso podría ser ejecutada de forma concurrente. Dichas actividades de inicio requerirán el uso de conjuntos de correlación.

## 7.6. Correlations

A diferencia de los servicios web, que son un modelo sin estado, los procesos de negocio requieren el uso de un modelo con estado. Cuando un cliente inicia un proceso de negocio, se crea una nueva instancia. Esta instancia "vive" durante la ejecución del proceso de negocio. Los mensajes que se envían a un proceso de negocio (utilizando operaciones sobre los *port types* y *ports*) necesitan ser entregados a la instancia correcta de dicho proceso de negocio.

BPEL proporciona un mecanismo para utilizar datos específicos del negocio para mantener referencias a instancias específicas del proceso de negocio. A esta característica la denomina correlación (*correlation*). En definitiva se trata de hacer corresponder un mensaje de entrada en la máquina BPEL con una instancia específica de un proceso de negocio. Los datos de negocio utilizados para establecer la correlación están contenidos en los mensajes intercambiados entre los *partners*. La localización exacta de la información normalmente difiere entre un mensaje y otro. Por ejemplo, el número de vuelo en el mensaje de un pasajero a la compañía aérea estará indicado en una ubicación diferente (será un campo con un nombre diferente) de la del mensaje de respuesta de la compañía al pasajero. Para especificar qué datos se utilizan para la correlación se utilizan las propiedades de los mensajes.

Por lo tanto, la **correlación** (*correlation*) es el mecanismo que el *runtime* de BPEL utiliza para seguir la pista a las conversaciones entre una instancia particular de un proceso y las correspondientes instancias de sus servicios *partners*. Es decir, la correlación permite que el *runtime* de BPEL conozca qué instancia del proceso está esperando a sus mensajes *callback*. Podemos pensar en la correlación como una clave primaria que utiliza el *runtime* de BPEL para correlacionar los mensajes de entrada y salida y enrutarlos convenientemente.

La correlación puede utilizarse, por ejemplo, para enviar la respuesta de procesos de negocio que requieren mucho tiempo de procesamiento (*long-running business processes*).

La correlación de mensajes puede lograrse mediante *WS-Addressing* (especificación estándar para definir los puntos extremo de cada mensaje), o mediante conjuntos de correlación BPEL. En esta sesión, vamos a ver únicamente esta segunda aproximación.

En muchos sistemas de objetos distribuidos, un componente del enrutado de un mensaje implica examinar el mensaje en busca de un identificador explícito ID de la instancia, que identifique el destino. Si bien el proceso de enrutado es similar, las instancias BPEL se identifican por uno o más campos de datos dentro del mensaje intercambiado. En términos de BPEL, estas colecciones de campos de datos que identifican la instancia de un proceso se conocen con el nombre de **conjuntos de correlación** (*correlation sets*).

Cada conjunto de correlación tiene un nombre asociado, y está formado por unas



propiedades definidas en WSDL. Una **propiedad** es un elemento de datos tipado con nombre, que se define en el documento WSDL. Por ejemplo, un número de chasis puede utilizarse para identificar un motor de un vehículo en un proceso de negocio. Dicho número probablemente aparecerá en varios mensajes y siempre identificará al vehículo. Supongamos que dicha información es de tipo `string`, ya que está codificada como una cadena de caracteres. Si utilizamos el nombre `chasisNumber` para referirnos a la parte del mensaje que hace referencia al número de chasis, estaremos proporcionando un nombre mucho más significativo a nivel de negocio que el tipo de datos `string`. El siguiente código muestra un ejemplo de definición de una propiedad denominada *NumeroDeVuelo*:

```
<!-- definición de una propiedad en el fichero WSDL-->
<definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
...
xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/">
...
  <bpws:property name="NumeroDeVuelo" type="xs:string" />
...
</definitions>
```

#### recuerda

Las propiedades de mensajes tienen un significado global para el proceso de negocio y pueden corresponderse (asociarse) con múltiples mensajes. Las propiedades se definen en el documento WSDL.

Las propiedades hacen referencia a partes de mensajes. Para hacer corresponder una propiedad con un elemento de una parte específica (o incluso un atributo) del mensaje, BPEL proporciona las *property aliases*. Con las *property aliases*, podemos mapear una propiedad con un elemento específico o un atributo de la parte del mensaje seleccionada.

Por lo tanto, en WSDL, una **propertyAlias** define cada uno de dichos mapeados (para hacer corresponder el campo de datos con el elemento del conjunto de correlación). Los mapeados son específicos para cada mensaje, por lo que una única propiedad puede tener múltiples *propertyAliases* asociadas con ella. Las *propertyAliases* especifican cómo extraer los datos de correlación de los mensajes en el atributo *query* (por defecto se utiliza XPath 1.0).

Las *property aliases* se definen en el fichero WSDL. La sintaxis para definir una *property alias* es la siguiente:

```
<!-- definición de una propiedad en el fichero WSDL-->
<wsdl:definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
...
xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/">
...
  <bpws:propertyAlias propertyName="property-name"
                        messageType="message-type-name"
                        part="message-part-name"
```

```

        query="query-string" />
    ...
</wsdl:definitions>

```

Siguiendo con el ejemplo anterior, vamos a definir la *property alias* para la propiedad de número de vuelo. Supongamos que hemos definido el mensaje *MensajeRespuestaViaje* en el WSDL de la línea aérea:

```

...
<message name="MensajeRespuestaViaje">
    <part name="datosConfirmacion" type="tns:TipoConfirmacionVuelo" />
</message>

```

Supongamos que el tipo *TipoConfirmacionVuelo* se ha definido como un tipo complejo, siendo uno de sus elementos el elemento *NumerodeVuelo* de tipo *xs:string*. Para definir el alias, escribimos el siguiente código:

```

<!-- definición de una property alias -->
<definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
    ...
    <bpws:property name="NumerodeVuelo" type="xs:string" />
    ...
    <bpws:propertyAlias propertyName="tns:NumerodeVuelo"
        messageType="tns:TipoConfirmacionVuelo"
        part="datosConfirmacion"
        query="/datosConfirmacion/NumerodeVuelo" />
    ...
</definitions>

```

## Extracción de propiedades

Para extraer los valores de las propiedades a partir de las variables, BPEL define la función *getVariableProperty*, definida en el espacio de nombres estándar de BPEL. La función tiene dos parámetros, el nombre de la variable y el nombre de la propiedad, y devuelve el nodo que representa la propiedad. La sintaxis es la siguiente:

```
getVariableProperty('variableName','propertyName')
```

Por ejemplo, para extraer la propiedad *NumerodeVuelo* de la variable *RespuestaViaje* escribiremos lo siguiente:

```
getVariableProperty('RespuestaViaje','NumerodeVuelo')
```

El uso de propiedades incrementa la flexibilidad en la extracción de datos relevantes de un mensaje, comparado con la función *getVariableData*. Mediante el uso de propiedades, no necesitamos especificar la localización exacta de los datos (tales como el número de vuelo), sino que simplemente utilizamos el nombre de la propiedad. Si cambiamos la localización de los datos, simplemente tendremos que cambiar la definición de la propiedad.

**recuerda**

Una *propiedad* es un concepto abstracto, mientras que una *propertyAlias* es el correspondiente concepto concreto asociado a dicha *propiedad*. La *propertyAlias* asocia la propiedad con un valor definido en el mensaje del servicio web utilizando una *query* mediante *xpath*

### 7.6.1. Uso de conjuntos de correlación

Un conjunto de correlación es un conjunto de propiedades, compartidas por varios mensajes, que se utilizan para la correlación. Cuando los mensajes correlacionados son intercambiados entre los *partners* de negocio, se puede definir dos roles. El *partner* que envía el primer mensaje en una operación de invocación es el "iniciador" y define los valores de las propiedades del conjunto de correlación. Otros *partners* son "seguidores" y obtienen los valores de las propiedades a partir de sus conjuntos de correlación de sus mensaje de entrada. Ambos "iniciador" y "seguidores" deben marcar la primera actividad que enlaza los conjuntos de correlación.

Un conjunto de correlación se utiliza para asociar mensajes con instancias de proceso. Cada conjunto de correlación tiene un nombre. Un mensaje puede correlacionarse con uno o más conjuntos de correlación. El mensaje inicial se utiliza para inicializar los valores de un conjunto de correlación. Los mensajes siguientes relacionados con esta correlación deben tener valores de propiedades idénticos con el conjunto inicial de correlación. Los conjuntos de correlación en BPEL pueden declararse globalmente para el proceso o dentro de ámbitos. La sintaxis se muestra a continuación:

```
<!-- definición de un conjunto de correlación -->
<correlationSets>
  <correlationSet name="correlation-set-name"
    properties="list-of-properties"/>
  <correlationSet name="correlation-set-name"
    properties="list-of-properties"/>
  ...
</correlationSets>
```

Para utilizar un conjunto de correlación, debemos **definir** dicho conjunto enumerando las propiedades que lo forman, y a continuación **referenciarlo** desde una actividad *receive*, *reply*, *invoke*, o desde la parte *onMessage* de la actividad *pick*. Para especificar qué conjuntos de correlación deben utilizarse, utilizaremos la actividad *<correlation>*, anidada en cualquiera de las actividades anteriores. La sintaxis se muestra a continuación:

```
<!-- conjunto de correlación anidado en una actividad de inicio-->
<correlations>
  <correlation set="name="
    initiate="yes|no|join" <!-- opcional
-->
    pattern="in|out|out-in /> <!-- utilizada en
invoke -->
</correlations/>
```

El atributo *initiate* sobre un *correlationSet* se utiliza para indicar si el conjunto de correlación está siendo iniciado. Después de que un *correlationSet* es iniciado, los valores de las propiedades de un *correlationSet* deben ser idénticas para todos los mensajes en todas las operaciones que hacen referencia a dichos *correlationSet*, y tienen validez en el correspondiente ámbito hasta su finalización. Un *correlationSet* puede ser iniciado como mucho una sola vez durante el ciclo de vida de la instancia y dentro del ámbito al que pertenece. Una vez iniciado un *correlationSet*, éste puede considerarse como una identidad de la instancia del proceso de negocio correspondiente. Los valores posibles para el atributo *initiate* son "yes", "join", "no". El valor por defecto del atributo *initiate* es "no".

- Cuando el valor del atributo *initiate* es "yes", la actividad relacionada debe intentar iniciar el *correlationSet*:
  - Si el *correlationSet* ya está iniciado, se lanzará la excepción *bpel:correlationViolation*.
- Cuando el atributo *initiate* tiene el valor "join", la actividad relacionada debe intentar iniciar el *correlationSet*, si el *correlationSet* todavía no está iniciado.
  - Si el *correlationSet* ya está iniciado y la restricción de consistencia de correlaciones se viola, se lanza la excepción *bpel:correlationViolation*.
- Cuando el atributo *initiate* tiene el valor "no" o no se le ha asignado un valor de forma explícita, entonces la actividad relacionada no debe intentar iniciar el *correlationSet*.
  - Si el *correlationSet* no ha sido iniciado previamente, se lanza la excepción *bpel:correlationViolation*.
  - Si el *correlationSet* ya está iniciado y la restricción de consistencia de correlaciones se viola, se lanza la excepción *bpel:correlationViolation*.

El *runtime* de BPEL utiliza la definición del conjunto de correlación y sus referencias para determinar qué elementos de información debe examinar en diferentes momentos de la ejecución del proceso. Cada instancia del proceso tiene una instanciación de cada uno de los conjuntos de correlación que han sido definidos para el proceso. Cada una de estas instanciaciones de los conjuntos de correlación se inicializa exactamente una vez durante la ejecución de la instancia del proceso, y solamente se utiliza durante las comparaciones que implican a mensajes de entrada y de salida. Si se produce un intento de reinicializar un conjunto de correlación, o un intento de uso de un conjunto que no ha sido inicializado, entonces el *runtime* de BPEL genera una *bpws:correlationViolation*. Para extraer los valores de un conjunto de correlación referenciado por una actividad, bien para inicializar el conjunto o para realizar comparaciones, se aplican los *alias* correspondientes al mensaje WSDL particular que está siendo examinado, para cada una de las propiedades que componen el conjunto de correlación referenciado. Para extraer los valores de las propiedades, BPEL define una función denominada *getVariableProperty*. Dicha función tiene dos parámetros: el nombre de la variable, y el nombre de la propiedad, y devuelve el nodo que representa la propiedad.

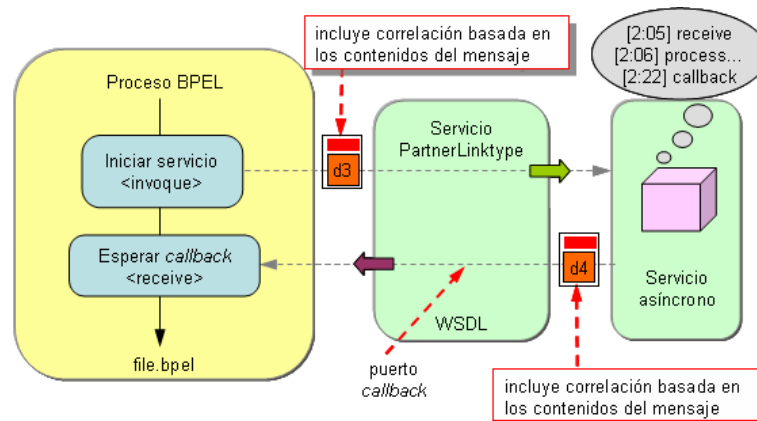
Si un conjunto de correlación se define sobre una actividad *receive* o *pick*, los valores de

las propiedades que comprenden el conjunto de correlación se extraen del mensaje de entrada y se comparan con los valores almacenados para el mismo conjunto de correlación en todas las instancias de dicho proceso, y después el mensaje es enrutado hacia la instancia que tiene los mismos valores.

Los conjuntos de correlación que aparecen en las actividades *invoke*, cuando la operación invocada es una operación de tipo *request/response*, utilizan un atributo adicional: *pattern*. El atributo *pattern* puede tener los valores: 'out', para indicar que el conjunto de correlación tiene que aplicarse al mensaje de salida ("request"), 'in', para el mensaje de entrada ("response"), o 'out-in', que significa que tiene que aplicarse durante las dos fases. El atributo *pattern* no se utiliza cuando se invoca a una operación de tipo *one-way*.

Pueden aparecer varios conjuntos de correlación sobre una única actividad. Algunos de ellos serán inicializados, y otros serán utilizados para hacer comparaciones.

La siguiente Figura muestra el uso de correlación en un proceso BPEL.



Uso de correlación en un proceso BPEL.

A continuación mostramos un ejemplo de uso de correlación en un escenario en el que el proceso BPEL primero comprueba la disponibilidad de un vuelo utilizando un *<invoke>* asíncrono y espera la respuesta *callback*. La respuesta *callback* contiene el número de vuelo *flightNo*, y se utiliza para iniciar el conjunto de correlación. A continuación el billete se confirma utilizando un *<invoke>* síncrono. Aquí, el conjunto de correlación se utiliza con un patrón *out-in*. Finalmente, el resultado se envía al cliente del proceso BPEL utilizando una actividad *callback <invoke>*. En este caso la correlación sigue un patrón *out*.

```
...
<sequence>
  ...
  <!-- Check the flight availability -->
  <invoke partnerLink="AmericanAirlines"
    portType="altn:FlightAvailabilityPT"
    operation="FlightAvailability"
    inputVariable="FlightDetails" />
```

```

<!-- Wait for the callback -->
<receive partnerLink="AmericanAirlines"
  portType="aln:FlightCallbackPT"
  operation="FlightTicketCallback"
  variable="TravelResponse" >

  <!-- The callback includes flight no therefore
    initiate correlation set -->
  <correlations>
    <correlation set="TicketOrder"
      initiate="yes" />
  </correlations>
</receive>
...
<!-- Synchronously confirm the ticket -->
<invoke partnerLink="AmericanAirlines"
  portType="aln:TicketConfirmationPT"
  operation="ConfirmTicket"
  inputVariable="FlightResponseAA"
  outputVariable="Confirmation" >

  <!-- Use the correlation set to confirm the ticket -->
  <correlations>
    <correlation set="TicketOrder" pattern="out-in" />
  </correlations>
</invoke>
...
<!-- Make a callback to the client -->
<invoke partnerLink="client"
  portType="trv:ClientCallbackPT"
  operation="ClientCallback"
  inputVariable="TravelResponse" >
  <!-- Use the correlation set to callback the client -->
  <correlations>
    <correlation set="TicketOrder" pattern="out" />
  </correlations>
</invoke>
</sequence>

```

### 7.6.2. Pasos a seguir para usar correlación en BPEL

Vamos a ordenar los pasos que acabamos de explicar en el apartado anterior para utilizar conjuntos de correlación, e ilustrarlos con ejemplos de código BPEL. Son los siguientes:

1. Definimos una propiedad (mediante un nombre y un tipo de datos) en nuestro fichero WSDL que será utilizado por el conjunto de correlación.

```
<bpws:property name="correlationData" type="xsd:int" />
```

2. Definimos un nombre de propiedad (*propertyAlias*) para cada elemento de los datos de correlación. El nombre de la propiedad puede ser el mismo para diferentes alias.

```

<bpws:propertyAlias messageType="CoreBankingP2P:CallbackType"
  part="accountNumber"
  propertyName="CoreBankingP2P:correlationData" />
<bpws:propertyAlias messageType="CoreBankingP2P:ServiceRequestType"
  part="accountNumber"

```

```
propertyName="CoreBankingP2P:correlationData"/>
```

- Definimos el conjunto de correlación en el documento BPEL relacionado antes de cualquier actividad *sequence* o *flow*.

```
<correlationSets>
  <correlationSet name="CS1"
    properties="CoreBankingP2P:correlationData"/>
</correlationSets>
```

- Referenciamos el conjunto de correlación dentro de la secuencia BPEL. El *runtime* de BPEL creará la instancia del conjunto de correlación para cada conversación (asumiendo que el atributo *createInstance* tenga el valor "yes" en uno de los *receives*).

```
...
<!-- esperamos la llamada del cliente -->
<receive name="receiveRequest" partnerLink="Client"
  portType="CoreBankingP2P:CoreBankingP2PPortType"
  operation="applyForCC" variable="ServiceRequest"
  createInstance="yes">
  <correlations>
    <correlation initiate="yes" set="CS1"/>
  </correlations>
</receive>
...
<!-- esperamos la llamada callback del servicio asíncrono invocado -->
<receive name="P2Pcallback" partnerLink="CoreBankingP2PPLT"
  portType="CoreBankingP2P:CoreBankingCallbackP2PPortType"
  operation="callback" variable="CallbackRequest">
  <correlations>
    <correlation set="CS1"/>
  </correlations>
</receive>
...
```

Cuando se recibe un mensaje, el *runtime* de BPEL examina los datos de correlación. Si se encuentra una coincidencia en el conjunto de datos de la instancia que mantiene la conversación con alguna instancia del proceso BPEL, entonces se servirá el mensaje recibido, en el ejemplo anterior será *P2Pcallback*.

### 7.6.3. Definición y uso de conjuntos de correlación con Netbeans

Como ya hemos expuesto en apartados anteriores, un conjunto de correlación es una colección de propiedades utilizadas por el *runtime* de BPEL para identificar el proceso correcto para recibir un mensaje. Cada propiedad del conjunto de correlación debe ser mapeada con un elemento en uno o más tipos de mensajes a través de los alias de las propiedades.

Para **definir un conjunto de correlación**, los mensajes de los servicios *partner* deben tener definidas propiedades y alias para las propiedades en sus correspondientes ficheros WSDL.

Después de que las propiedades y alias para las propiedades se hayan añadido al fichero WSDL asociado al proceso, podemos definir los conjuntos de correlación para el elemento *Process*, de la siguiente forma:

- En la vista de diseño, pulsar con el botón derecho del ratón sobre el elemento *Process* y elegir *Add > Correlation Set*. Aparecerá un cuadro de diálogo.
- Cambiar el nombre del conjunto de correlación y seleccionar *Add* para añadir propiedades.
- En el cuadro de diálogo *Property Chooser*, seleccionar una propiedad a añadir al conjunto y pulsar sobre *OK*. Por defecto, el cuadro de diálogo *Property Chooser* solamente muestra aquellos ficheros que ya han sido referenciados en el proceso. Sin embargo, el proyecto puede contener otros ficheros *.swdl* y *.xsd* que todavía no hayan sido importados en el proceso. Si seleccionamos una propiedad definida en un fichero no importado, el IDE automáticamente añade los *imports* requeridos en el proceso BPEL.

Después de definir el conjunto de correlación, se añade dicho conjunto de correlación a las actividades *invoke*, *Receive*, o *Reply*, entre otros. Para añadir un conjunto de correlación a un elemento, se debe realizar lo siguiente:

- En la vista de diseño, hacer doble *click* sobre una actividad.
- En el editor de propiedades, seleccionar la pestaña *Correlations* y pinchar sobre *Add*.
- En el cuadro de diálogo para seleccionar un conjunto de correlación, seleccionar el conjunto y pinchar en *OK*.
- Añadir tantos conjuntos de correlación como sean necesarios y pinchar sobre *OK*.



## 8. Ejercicios de Procesos BPEL síncronos y asíncronos

### 8.1. Proceso BPEL síncrono: Servicio de orden de compra.

Como ya hemos indicado, las operaciones síncronas son adecuadas en procesos de negocio que, ante una petición, requieren una respuesta inmediata. La ejecución, en la parte del consumidor del servicio puede continuar solamente después de que la respuesta sea recibida y procesada.

Con este ejercicio utilizaremos un proceso BPEL, denominado *POService* que: (a) proporciona una operación síncrona a un cliente externo; (b) consume una operación síncrona suministrada por un servicio Web *partner* (1,5 puntos).

Recordemos que cuando se diseña un proceso de negocio que incorpora interacciones con servicios Web síncronos, se necesitan las siguientes construcciones:

- `partnerLinks` que representan servicios Web *partner*
- `variables` que almacenan los datos intercambiados entre los servicios Web
- Una actividad `invoke` para **consumir** un servicio
- Un par de actividades `receive-reply` para **proporcionar** un servicio
- Mapeados de datos de entrada y salida para conseguir la lógica de negocio requerida

#### 8.1.1. Lógica de negocio de los servicios

Los dos servicios Web que vamos a implementar son: (a) un servicio de orden de compra, *POService*, que es consumido por un cliente externo mediante SOAP sobre HTTP; (b) un servicio de comprobación de inventariado, *InventoryService*, consumido por el proceso BPEL que proporciona el servicio de orden de compra. Ambos servicios, *POService* e *InventoryService* se implementan como servicios BPEL.

Cuando el suministrador del servicio de orden de compra (*POService*) recibe una petición de un cliente, tienen lugar los siguientes eventos:

- El suministrador del servicio de orden de compra:
  - Asigna el precio de la petición
  - Llama al servicio de inventario (*InventoryService*) para comprobar el estado del inventario
- La lógica de negocio del suministrador del servicio de inventario comprueba la disponibilidad de un *item*. Si el valor de *orderDescription* comienza con *OrderVal*, entonces el estado de la orden en el inventario será la de "disponible"
- Según el resultado del servicio de inventario, el suministrador del servicio de orden de compra responde con lo siguiente:
  - La orden de compra cumplimentada
  - Un error (en forma de mensaje) indicando que la orden no puede ser completada

La Figura 1 ilustra el diseño de la solución que vamos a plantear:

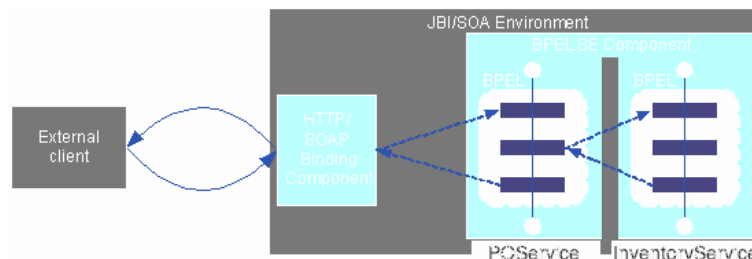


Figura 1. Interacción con un servicio Web Sincrónico.

### 8.1.2. Creación del proyecto BPEL

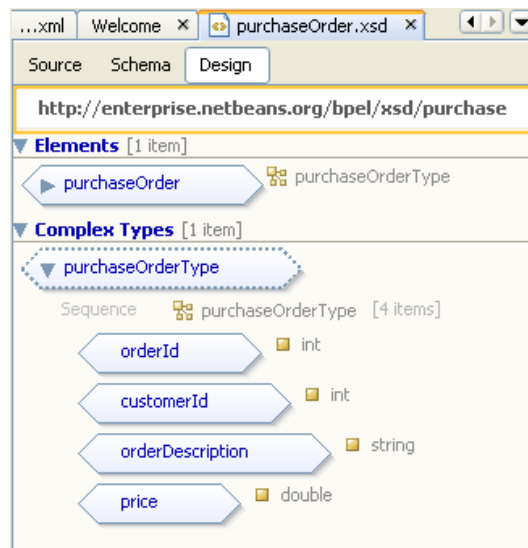
Para este ejercicio proporcionamos un proyecto BPEL denominado *OrdenCompraSincrona* con los WSDL y los ficheros de esquema que vamos a utilizar (fichero *ordenCompra.zip*). Descomprimos el fichero *ordenCompra.zip*, y veremos que se crea la carpeta *OrdenCompra*. A continuación desde el menú principal de Netbeans elegimos *File > Open Project*, y seleccionamos el proyecto *OrdenCompraSincrona* (dentro de la carpeta *OrdenCompra*).

### 8.1.3. Creación del esquema XML

En el proyecto *OrdenCompraSincrona*, encontraremos los ficheros de esquema *.xsd* ya creados (en el directorio *Process Files*). Concretamente se trata de *purchaseOrder.xsd*, e *inventory.xsd*, que contienen la definición de los tipos de datos que utilizan los servicios BPEL que vamos a implementar.

En el fichero *purchaseOrder.xsd* hemos definido el tipo complejo *purchaseOrderType*, que representa una orden de compra. El tipo *purchaseOrderType* está formado por los elementos locales: *orderId*, *customerId*, *orderDescription*, y *price*, que representan el identificador de la orden de compra, identificador del cliente, la descripción de la orden de compra y el precio de la compra, respectivamente.

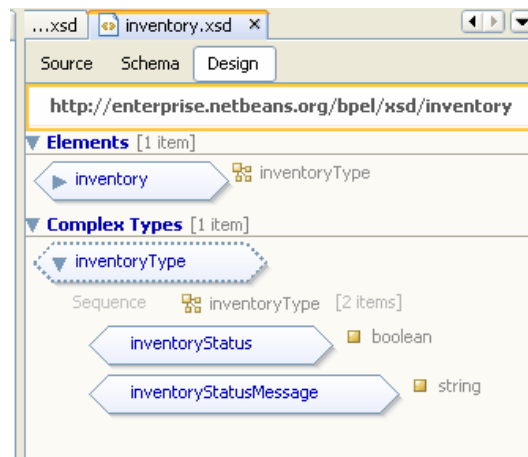
También definimos el elemento global *purchaseOrder*, del tipo *purchaseOrderType*.



Esquema purchaseOrder.xsd

El fichero de esquema *inventory.xsd* contiene la definición del tipo complejo *inventoryType*, que almacena información sobre el estado del inventario. El tipo *inventoryType* está formado por los elementos locales: *inventoryStatus* e *inventoryStatusMessage*.

También definimos el elemento global *inventory*, del tipo *inventoryType*.



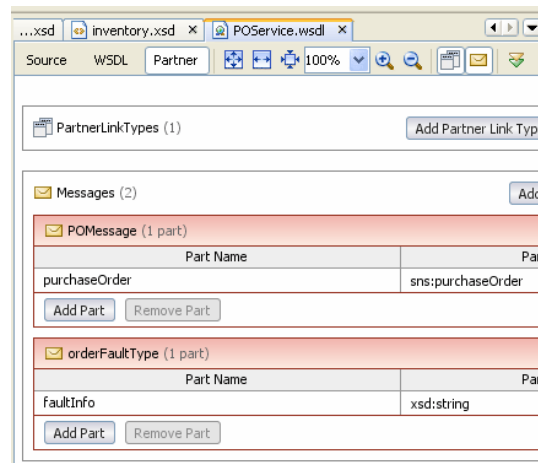
Esquema inventory.xsd

#### 8.1.4. Creación del documento WSDL

En el directorio *Process Files* del proyecto *OrdenCompraSincrona* podemos ver los ficheros *POService.wsdl* e *InventoryService.wsdl*.

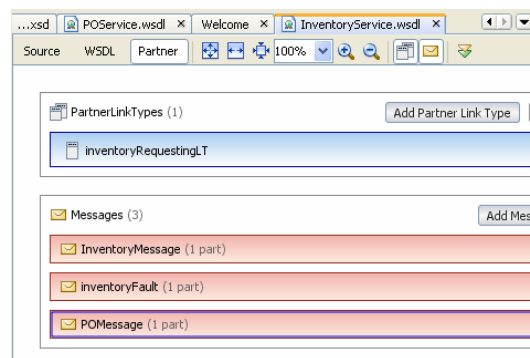
El fichero *POService.wsdl* contiene la definición de dos mensajes: *POMessage*, que

representa la orden de compra, y *orderFaultType*, que hace referencia a la información sobre el error producido (en caso de que en el inventario no queden existencias para servir la orden de compra).



Fichero POService.wsdl

El fichero *inventoryService.wsdl* contiene la definición de los mensajes: (a) *InventoryMessage*, formado por un elemento de tipo *inventoryType*, (b) *InventoryFault*, que contiene información sobre el error producido, y (c) *POMessage*, que contiene la orden de compra.



Fichero inventoryService.wsdl

## PARTNER LINK TYPES

En cada fichero *wsdl* hemos definido un *partnerLinkType*. Concretamente, en *POService.wsdl* se define un *partnerLinkType* (denominado *purchasingLT*) con el rol *purchaseService*. Dicho rol hace referencia a la operación WSDL *sendPurchaseOrder* a través del *portType* denominado *purchaseOrderPT*.

```
<!-- POService.wsdl-->
<plink:partnerLinkType name="purchasingLT">
  <plink:role name="purchaseService">
```

```

        portType="tns:purchaseOrderPT">
    </plink:role>
</plink:partnerLinkType>

```

La operación *sendPurchaseOrder* define una entrada que debe enviarse al suministrador del servicio Web correspondiente, y espera una respuesta o bien un error.

```

<!-- POService.wsdl-->
<portType name="purchaseOrderPT">
  <operation name="sendPurchaseOrder">
    <input name="sendPurchaseOrderRequest"
      message="tns:POMessage"></input>
    <output name="sendPurchaseOrderReply"
      message="tns:POMessage"></output>
    <fault name="cannotCompleteOrder"
      message="tns:orderFaultType"></fault>
  </operation>
</portType>

```

En el fichero *InventoryService.wsdl* se define un *partnerLinkType* (denominado *inventoryRequestingLT*) con el rol *inventoryService*. Este rol hace referencia a la operación *inventoryService* a través del *portType* denominado *inventoryPortType*.

```

<!-- InventoryService.wsdl-->
<plink:partnerLinkType name="inventoryRequestingLT">
  <plink:role name="inventoryService"
    portType="tns:inventoryPortType">
  </plink:role>
</plink:partnerLinkType>

```

La operación *inventoryService* espera un mensaje *purchaseOrder*, y responde con un *inventoryStatus* o *inventoryFaultType*.

```

<!-- InventoryService.wsdl-->
<portType name="inventoryPortType">
  <operation name="inventoryService">
    <input name="purchaseOrder"
      message="tns:POMessage"></input>
    <output name="inventoryStatus"
      message="tns:InventoryMessage"></output>
    <fault name="inventoryFaultType"
      message="tns:inventoryFault"></fault>
  </operation>
</portType>

```

#### NOTA

Si el fichero WSDL de un servicio Web existente no contiene una definición para un *partnerLinkType*, podemos crear un fichero WSDL *wrapper* para importar el fichero WSDL original, y añadirle la definición de los *partnerLinkTypes*. Posteriormente, podemos hacer referencia a dicho *wrapper* desde nuestro proceso BPEL.

### 8.1.5. Definimos el proceso BPEL

En nuestro caso tendremos que crear dos documentos *bpel*, correspondientes a los dos servicios BPEL de la solución propuesta, con los nombres *POService.bpel*, e *InventoryService.bpel*.

#### Creamos *POService.bpel*

Nos situamos en el nodo *Process Files* del proyecto (*OrdenCompraSincrona*) y con el botón derecho seleccionamos *New > BPEL Process*. En el campo *File Name* de la ventana que aparece, ponemos el nombre del fichero: *POService* y pulsamos el botón *Finish*. A continuación se abrirá el fichero *POService.bpel* creado, con la vista de Diseño abierta.

El nombre del proceso BPEL por defecto es "\_PROCESS\_NAME\_", para cambiarlo tenemos que pinchar con el botón derecho sobre el rectángulo que contiene el nombre y elegimos la opción "Properties". Nos aparecerá la lista de propiedades del proceso (etiqueta <process> del proceso BPEL, ver la vista de fuentes). Si pinchamos sobre el recuadro con puntos suspensivos que hay a la derecha del valor actual de la propiedad "Name", podremos editar dicho valor. Vamos a cambiarlo a "POService". Podemos comprobar en la vista de fuentes que se ha cambiado el atributo "name" de la etiqueta "process".

#### Añadimos los *Partner Links* del proceso

Recordemos que los elementos *partnerLink* especifican los *partners* con los que interactúa el proceso BPEL. Cada *partnerLink* se corresponde con un *partnerLinkType* específico definido en el correspondiente fichero WSDL.

Un elemento *partnerLink* puede contener uno o dos roles:

- *myRole*: especifica el rol del proceso BPEL. Si definimos solamente *myRole* en un *partnerLink*, hacemos posible que cualquier *partner* o cliente pueda interactuar con el proceso BPEL sin ningún otro requerimiento adicional sobre los *partners*.
- *partnerRole*: especifica el rol del *partner*. Si solamente definimos *partnerRole* en el *partnerLink*, permitimos interacciones con un *partner* sin imponer restricciones sobre el servicio que realice la llamada.

En el proceso *bpel POService* vamos a definir un *partnerLink* que denominaremos *Cliente*, y que mostrará la interacción del cliente con el proceso *POService*. Dicho *partner* es del tipo *purchasingLT*, en el que el rol del proceso BPEL es *purchaseService*. Para ello realizaremos las siguientes acciones:

- Seleccionamos el fichero *POService.wsdl* y lo arrastramos a la vista de diseño de *POService.bpel*, (a la izquierda del rectángulo que representa del proceso *bpel*. Aparecerá un punto naranja indicando dónde debemos "soltar" dicho *wsdl*).
- En la ventana de propiedades del componente *partnerLink* que acabamos de añadir,

modificamos el nombre por defecto "PartnerLink1" por el de *Cliente*. Comprobamos que los valores las propiedades "wsdl file", "partner Link type" y "my role" son */POService.wsdl*, *purchasingLT*, y *purchaseService*, respectivamente.

Como resultado, si pasamos a la vista de código, podemos comprobar que hemos generado el siguiente código BPEL:

```
<!-- POService.bpel-->
<partnerLink name="Cliente"
  partnerLinkType="tns:purchasingLT"
  myRole="purchaseService"/>
```

Puesto que *POService* consume el servicio *InventoryService*, definimos otro *partnerLink* al que llamaremos *requestInventoryPLink*. En este caso el fichero wsdl asociado es: *InventoryService.wsdl*. Procedemos igual que en el caso anterior, pero arrastramos el fichero wsdl a la derecha del proceso bpel. El nombre del *partnerLinkType* ya existente es: *inventoryRequestingLT*. Y finalmente definimos un *partnerRole* con el nombre *inventoryService*

```
<!-- POService.bpel-->
<partnerLink name="requestInventoryPLink"
  partnerLinkType="tns:inventoryRequestingLT"
  partnerRole="inventoryService"/>
```

### Definir variables globales del proceso

Para añadir variables, seleccionamos el proceso bpel, y nos aparecerán cuatro iconos en la parte superior izquierda de dicho proceso. Pulsamos el primero de ellos (el situado más hacia la izquierda), y nos aparecerá un editor de variables; a cada una de ellas vamos a asignarle un tipo (que seleccionaremos de los tipos que nos muestra el editor). Concretamente, las variables a crear y sus tipos son:

- Nombre: *purchaseOrderRequest*; tipo: *POMessage* (desde *POService.wsdl*)
- Nombre: *purchaseOrderReply*; tipo: *POMessage* (desde *POService.wsdl*)
- Nombre: *purchaseOrderFault*; tipo: *orderFaultType* (desde *POService.wsdl*)
- Nombre: *inventoryServiceRequest*; tipo: *POMessage* (desde *InventoryService.wsdl*)
- Nombre: *inventoryServiceReply*; tipo: *InventoryMessage* (desde *InventoryService.wsdl*)
- Nombre: *inventoryServiceFault*; tipo: *inventoryFault* (desde *InventoryService.wsdl*)

Las tres primeras variables las utilizaremos para almacenar los mensajes de petición y respuesta entre el cliente y el servicio *POService*. Las otras tres almacenarán los mensajes de petición y respuesta entre el servicio *POService* y el servicio *InventoryService*.

El código BPEL generado es el siguiente:

```
<!-- POService.bpel-->
<variables>
```

```

<variable name="purchaseOrderRequest"
  messageType="ns0:POMessage"></variable>
<variable name="purchaseOrderReply"
  messageType="ns0:POMessage"></variable>
<variable name="purchaseOrderFault"
  messageType="ns0:orderFaultType"></variable>
<variable name="inventoryServiceRequest"
  messageType="ns1:POMessage"></variable>
<variable name="inventoryServiceReply"
  messageType="ns1:InventoryMessage"></variable>
<variable name="inventoryServiceFault"
  messageType="ns1:inventoryFault"></variable>
</variables>

```

### Añadimos una actividad *receive*

En la sección *Web Service* de la paleta de elementos, seleccionamos el icono *Receive* y lo arrastramos al área de diseño entre las actividades *Start* y *End* del proceso. El IDE de *Netbeans* muestra mediante marcas visuales dónde podemos "soltar" el elemento que estamos arrastrando.

Una vez que "depositemos" la actividad *Receive* en el lugar deseado, pulsamos con el botón izquierdo sobre el icono que aparece en la parte superior del rectángulo que rodea al componente que acabamos de insertar y se abrirá el editor de propiedades, a las que daremos los siguientes valores:

- Name: *receivePOFromClient*.
- Partner Link: *Cliente*.
- Operation: *sendPurchaseOrder*.
- Input variable: *purchaseOrderRequest*.
- La casilla: *Create Instance* estará marcada.

Como ya hemos creado todas las variables que necesitamos, la variable de entrada la elegiremos pulsando sobre el botón "Browse". De forma alternativa, podríamos haber creado dicha variable después de insertar el componente, en cuyo caso tendríamos que elegir el botón "Create"

El código BPEL resultante es:

```

<!-- POService.bpel-->
<receive name="receivePOFromClient"
  partnerLink="Cliente"
  portType="ns1:purchaseOrderPT"
  operation="sendPurchaseOrder"
  variable="purchaseOrderRequest"
  createInstance="yes">
</receive>

```

### Añadimos una actividad *assign*

En el siguiente paso vamos a añadir una asignación a la que llamaremos *Assign 1*. Se trata de asignar un valor a la parte *price* de la variable (*purchaseOrderRequest*). El valor será:



49.98.

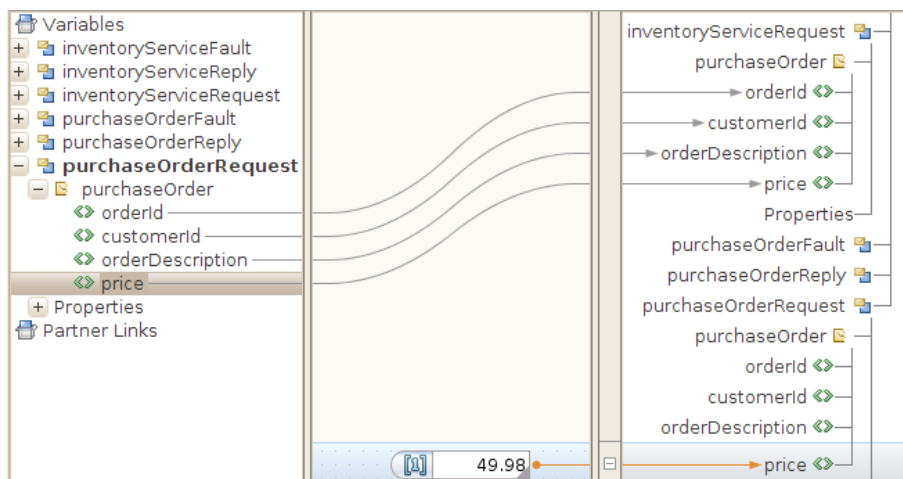
Para ello tendremos que arrastrar desde la paleta la actividad *Assign* en la sección de *Basic Activities*, hasta colocarla a continuación de la actividad *receive*. Para hacer la asignación pasaremos a la vista "Mapper", para lo cual seleccionamos la actividad "Assign1" y pinchamos sobre el botón "Mapper" en la parte superior del editor.

En nuestro caso particular, queremos asignar un precio a la orden de compra. Concretamente queremos asignar a la variable *purchaseOrderRequest.purchaseOrder/price* el valor 49.98 ("Number Literal"). Para ello, en la lista de variables de la parte derecha, desplegamos la variable *purchaseOrderRequest > purchaseOrder >* y dejamos visible la parte *price* y realizamos la asignación.

El código BPEL generado es el siguiente:

```
<!-- POService.bpel-->
<copy>
  <from>49.98</from>
  <to>$purchaseOrderRequest.purchaseOrder/ns0:price</to>
</copy>
```

A continuación asignamos todos los valores del mensaje *purchaseOrderRequest.purchaseOrder* en la parte izquierda del Mapper, a los elementos correspondientes del mensaje *inventoryServiceRequest.purchaseOrder* (en la parte derecha del Mapper). El resultado se muestra a continuación.



BPEL Mapper: asignaciones correspondientes a la actividad assign1

### Añadimos la actividad *invoke*

La actividad *invoke* tiene el efecto de consumir un servicio Web. Añadiremos dicha actividad a continuación de la actividad *Assign1*. El proceso es similar al realizado para la actividad *receive*. En este caso, los valores de las propiedades para la actividad *invoke*

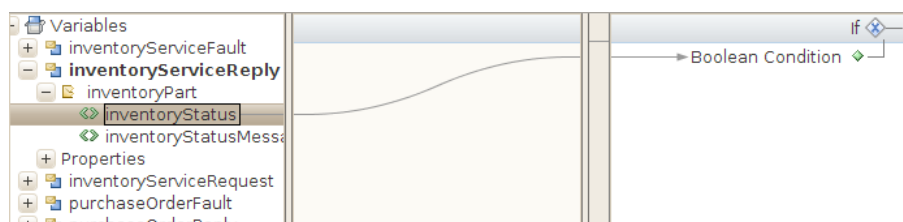
serán los siguientes:

- Name: *CallInventoryService*
- PartnerLink: *requestInventoryPLink*
- Operation: *inventoryService*
- Input variable: *inventoryServiceRequest*
- Output variable: *inventoryServiceReply*

### Añadimos la actividad *if*

La actividad *if* tiene el efecto de bifurcar la lógica del proceso BPEL en función de una condición cuyo resultado será cierto o falso. La añadiremos a continuación de la actividad *invoke*. Los valores de las propiedades de la actividad *if* serán:

- Name: *CheckAvailability*
- Condition: *\$inventoryServiceReply.inventoryPart/ns3:inventoryStatus*. A continuación mostramos el mapeado a realizar en la ventana *BPEL Mapper*, asignando el valor de la parte *inventoryStatus* (de tipo *boolean*) a la condición booleana de la actividad *if*.



BPEL Mapper: asignación correspondiente a la actividad *CheckAvailability*

En la parte cierta de la actividad *if* (parte "then", queda justo debajo del símbolo que representa el *if*), añadimos una actividad *sequence* (con nombre "AvailTrue"), en la que anidaremos las actividades *assign* y *reply*.

Para la actividad *assign* a la que llamaremos *assign 2*, haremos corresponder la variable *purchaseOrderRequest* con la variable *PurchaseOrderReply*.

Para la actividad *reply*, los valores de sus propiedades serán (pulsamos con botón derecho y elegimos la opción "Properties", y a continuación pinchamos con botón izquierdo del ratón sobre el botón con puntos suspensivos a la derecha del campo "Property Editor"; o bien, de forma alternativa pinchamos con botón izquierdo sobre el icono de edición en la parte superior de la actividad *reply*):

- Name: *ReplyPO*
- Partner Link: *Cliente*
- Operation: *sendPurchaseOrder*
- Normal response, Output variable: *purchaseOrderReply*

Procederemos de la misma forma para la parte "else" de la actividad *if*, es decir, añadimos una actividad *sequence* (con nombre "AvailFalse") a la que anidamos las actividades

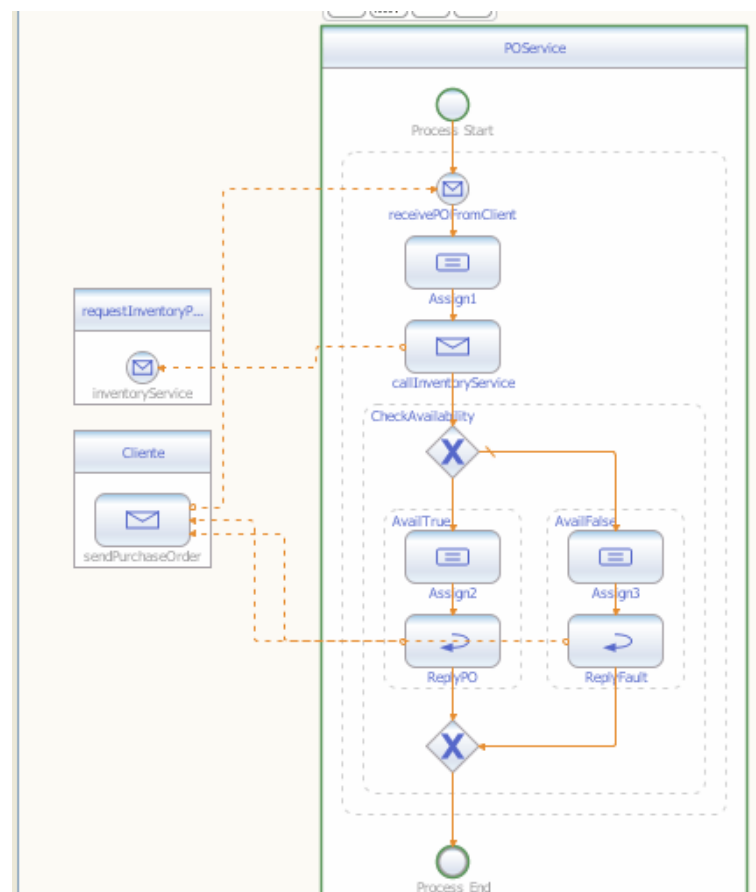
*assign* y *reply*.

Para la actividad *assign* a la que llamaremos *assign 3*, haremos corresponder la variable *inventoryServiceReply > inventoryPart > inventoryStatusMessage* con la variable *purchaseOrderFault > faultInfo*.

Para la actividad *reply*, los valores de sus propiedades serán:

- Name: *ReplyFault*
- Partner Link: *Cliente*
- Operation: *sendPurchaseOrder*
- Fault response, Fault Name: *ns0:cannotCompleteOrder*
- Fault response, Fault Variable: *purchaseOrderFault*

Mostramos el resultado final de la vista de diseño del proceso *POService.bpel*.



Vista de diseño del proceso *POService.bpel*

Guardamos todos los cambios con *File > Save All* (si es que no hemos ido guardando antes los cambios con *File > Save*).

De forma similar, crearemos el proceso *InventoryService.bpel*. En el fichero *inventory.zip*

que se proporciona, encontraréis el fuente *InventoryService.bpel* que podéis incorporar en vuestro proyecto. Para ello no tendréis más que, desde el sistema operativo, copiar dicho fichero en el directorio *src* del proyecto.

#### 8.1.6. Compilamos el proyecto

---

Para compilar el proyecto BPEL, simplemente nos situamos en el nodo del proyecto *OrdenCompraSincrona* y con el botón derecho del ratón seleccionamos la opción *Build*.

#### 8.1.7. Creamos la Composite Application

---

Crearemos una nueva *Composite Application* para que podamos desplegar nuestro proyecto BPEL en el servidor de aplicaciones. Para ello seguiremos los siguientes pasos:

- Seleccionamos *File > New Project > Service Oriented Architecture > Composite Application*
- Asignamos el nombre *OrdenCompraSincronaApplication*. Vamos a crear la aplicación en el mismo directorio que el proyecto "OrdenCompraSincrona" (es decir, dentro de la carpeta "OrdenCompra").
- Una vez hecho esto, nos aparecerá un nuevo proyecto con el nombre *OrdenCompraSincronaApplication*. Nos situamos en dicho nodo, y con el botón derecho del ratón seleccionamos *Add JBI Module*
- Seleccionamos el proyecto *OrdenCompraSincrona* y pulsamos el botón: *Add Project JAR Files*. Si desplegamos el nodo *JB Modules* veremos que se ha añadido el nodo *OrdenCompraSincrona.jar*

A continuación debemos asegurarnos de que el servidor está en marcha para poder desplegar la *Composite Application* que hemos creado. Recuerda que para trabajar con BPEL necesitamos arrancar el servidor **Glassfish v2.x**

Para desplegar el proceso BPEL en el servidor de aplicaciones nos situaremos en el nodo *OrdenCompraSincronaApplication* y elegiremos *Deploy*. Podemos ver el componente desplegado "OrdenCompraSincronaApplication" como un componente JBI, dentro de "Service Assemblies".

#### 8.1.8. Probamos el proceso BPEL

---

Para probar nuestro proceso BPEL vamos a añadir un caso de prueba. Nos situamos en el nodo *Test* del proyecto *OrdenCompraSincronaApplication* y elegimos *New Test case* al que llamaremos *TestExito*.

A continuación elegimos el fichero WSDL del proceso a probar: *POService.wsdl*. La operación a probar será *sendPurchaseOrder*. Vemos que se ha creado el fichero *Input.xml*, que contiene la estructura del mensaje de entrada. Como datos de entrada podemos utilizar los datos del fichero *Input.xml* que se proporciona. Para ello podemos pegar el contenido del fichero *Input.xml* proporcionado en el nuevo fichero *Input.xml*

reemplazado así su contenido (O simplemente cambiar los valores por defecto de los elementos "orderId", "customerId", "orderDescription" y "price", copiando los del fichero Input.xml). Guardamos los cambios.

Para ejecutar el proceso con dicha entrada, nos situamos en el nodo *TestExito* y elegimos *Run*.

Recordad que la primera vez obtendremos un informe por pantalla indicándonos que no se ha "pasado la prueba" debido a que el fichero *Output.xml* estará vacío inicialmente.

Al ejecutar la prueba por segunda vez (y sucesivas) ya debemos obtener un resultado correcto.

## 8.2. Proceso BPEL asíncrono. Uso de correlación: Hola Mundo.

Cada proceso de negocio BPEL es un servicio Web, por lo que puede parecer que solamente necesitamos conocer el puerto de destino para poder enviar un mensaje a un proceso BPEL. Sin embargo, puesto que los procesos de negocio son procesos con estado, se instancian basándose en su estado. En este ejercicio, utilizaremos los conjuntos de correlación de BPEL para dar soporte a una colaboración entre servicios Web con estado. (1,5 puntos)

Comenzaremos por crear un proyecto bpel síncrono, a partir de una plantilla: New Project->Samples->SOA-> Synchronous BPEL Process. Dejaremos el nombre por defecto *Synchronous* como nombre de proyecto.

### Cuidado

En la sesión anterior hemos utilizado también la misma plantilla, ten cuidado de no poner el mismo nombre de proyecto que utilizaste en la sesión anterior. Si has seguido las instrucciones del ejercicio de la sesión anterior no habrá problema, porque utilizamos un nombre diferente, pero si dejaste el nombre por defecto estarás sobrescribiendo el proyecto de la sesión anterior.

Si echamos un vistazo al wsdl del proyecto creado, en la vista WSDL, vemos que nuestro servicio ofrece una operación síncrona denominada ***operation1*** (ver nodo Port Types->portType1).

Podemos ver también el fichero de esquema, en la vista de Diseño, en el que hay definido un elemento global denominado ***typeA*** de tipo *simpleProcess*.

### 8.2.1. Modificamos el fichero de esquema

Vamos a modificar el fichero de esquema para incluir un elemento de tipo cadena de caracteres, al que denominaremos ***id***, y que utilizaremos como información para correlacionar los mensajes que recibe y devuelve nuestro proceso bpel.

Primero añadiremos al tipo complejo *simpleProcess* un nuevo elemento con nombre ***id***,

de tipo *string*.

A continuación vamos a añadir un nuevo elemento global de tipo *simpleProcess*, con el nombre *typeB*. Este nuevo elemento lo utilizaremos cuando añadamos una nueva operación a nuestro proceso *bpel*

Como resultado tendremos dos elementos globales: *typeA*, y *typeB*, de tipo *simpleProcess*. El tipo *simpleProcess* tiene dos elementos: *paramA*, e *id*.

### 8.2.2. Modificamos el wsdl del proceso *bpel*

Vamos a añadir un nuevo mensaje, para ello abrimos el fichero *wsdl*, y nos situamos en la vista *WSDL*. Para añadir un nuevo mensaje, pulsamos con botón derecho sobre el nodo *Messages*, y seleccionamos "Add Message". Veremos que hemos añadido un nuevo nodo con el nombre "message1". Vamos a cambiar el nombre. De nuevo con botón derecho sobre dicho nodo, esta vez elegimos "Properties", y cambiamos el nombre por el de *setPrefixRequest*. Vamos editar también el nodo "part1", y en el campo "Element or Type" vamos a seleccionar *typeB*. En definitiva, hemos añadido el mensaje *setPrefixRequest*, que contiene el subnodo "part1" con el elemento asociado *typeB*.

A continuación añadimos una operación al *portType1*. Para ello, con botón derecho seleccionamos "Add->Operation". Podemos utilizar los siguientes valores para los campos:

```
OperationName= setPrefix
OperationType= one-way operation
Input= tns:setPrefixRequest
```

Con esto estamos indicando que nuestro servicio proporcionará dos operaciones: *operation1* (que es una operación de tipo *request-response*) y *setPrefix* (que es una operación de tipo *one-way*).

A continuación tenemos que añadir la operación en el nodo "binding1", que define la parte concreta de las operaciones que se ofertan. Para ello, nos situamos sobre *binding1*, y con botón derecho seleccionamos "Add->Binding operation". Al hacer esto automáticamente se nos añade la operación *setPrefix*. Finalmente tendremos que modificar "a mano" el código fuente. Nos vamos a la vista de fuentes, y localizamos el código que se ha añadido automáticamente al realizar la operación anterior. Veremos:

```
<!-- extracto de fichero Synchronous.wsdl -->
<binding name="binding1" ...
...
  <operation name="setPrefix">
    <input name="input2"/>
  </operation>
```

Editamos las líneas anteriores para convertirlas en:

```
<!-- extracto de fichero Synchronous.wsdl -->
<binding name="binding1" ...
```

```

...
<operation name="setPrefix">
  <input name="input2">
    <soap:body use="literal"/>
  </input>
</operation>

```

Ahora nuestro servicio ofrece las dos operaciones, y tiene definido el tipo de transporte y el mensaje soap que espera recibir.

### 8.2.3. Añadimos un estado al proceso de negocio

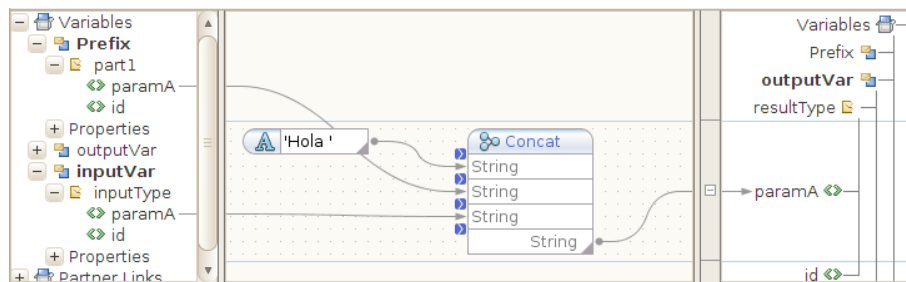
Para convertir al proceso *Synchronous* en un proceso de negocio con estado, lo que haremos será añadir otra actividad *Receive* a dicho proceso. Esta actividad añade una comunicación asíncrona con el proceso y éste se convierte en un proceso con estado. El proceso recibe un mensaje de entrada, que será extendido con un prefijo, que es proporcionado también por el cliente en un segundo mensaje.

Para añadir el estado al proceso *Synchronous* realizaremos los siguientes pasos:

- Expandimos el nodo *Synchronous* > *Process Files* y abrimos el fichero *Synchronous.bpel*.
- Arrastramos el icono *Receive* desde la paleta hasta situarla entre las actividades *start* y *Assign1*. Esta acción añade una actividad *Receive1* en el diagrama.
- Vamos a cambiar las propiedades de *Receive1* de la siguiente forma:
  - Name: *SetPrefix*
  - Partner Link: *Synchronous*
  - Operation: *setPrefix*
  - Creamos una variable de entrada utilizando el botón *Create* con el nombre *Prefix* (de tipo *setPrefixRequest*) y cerramos el editor de propiedades
- Añadimos un prefijo al nombre de usuario en la cadena de salida de la siguiente forma:
  - Abrimos el *BPEL mapper* pinchando sobre el botón *Mapper*, estando seleccionada la actividad *Assign1* y borramos el enlace existente entre las variables.
  - Seleccionamos el nodo *outputVar->resultType->paramA*, en el panel de la derecha de *Mapper* y añadimos un *String->Literal*, con el valor 'Hola'. A continuación añadimos un *String-Concat*, y concatenamos 'Hola', *Prefix->part1->paramA*, *inputVar->part1->paramA* (estos dos últimos nodos del panel de la izquierda). El resultado lo asignaremos a *outputVar->resultType->paramA* en la parte izquierda del *mapper* y la enlazamos como segundo parámetro de la función *Concat*.
  - Pulsamos *Ctrl-S* para guardar los cambios.

El resultado de la nueva asignación en la vista de *Mapper* es la siguiente





Añadimos un prefijo al nombre de usuario

Con los pasos anteriores hemos añadido una comunicación asíncrona en el proceso. Ahora, después de que se reciba el primer mensaje y se inicialice el proceso, el proceso BPEL necesita esperar a otro mensaje en la actividad *SetPrefix*.

Imaginemos una situación en la que se crean varias instancias del proceso y todas ellas están esperando un mensaje en la segunda actividad *Receive*.

Como ya hemos visto, la máquina BPEL utiliza conjuntos de correlación para decidir a qué instancia proceso se le ha enviado el mensaje. Como resultado de los cambios que hemos realizado, el proceso devuelve la cadena de saludo extendida con un prefijo que pasaremos como entrada al proceso.

#### 8.2.4. Definimos las propiedades de correlación y los alias de las propiedades

Las propiedades se utilizan típicamente para almacenar elementos para la correlación de instancias de servicios con mensajes. Usaremos *property aliases* para especificar qué parte de los datos tiene que extraerse de los mensajes y con qué propiedad tienen que asociarse los datos extraídos. Una propiedad es un concepto abstracto, mientras que la *propertyAlias* es el aspecto concreto correspondiente. Las *property aliases* enlazan las propiedades con valores definidos en el mensaje del servicio Web utilizando una *query xpath*.

Para **crear una propiedad**, seguiremos los siguientes pasos:

- Desde el menú principal, elegimos *Window > Navigator*, que debe mostrarse en el panel inferior izquierda de la pantalla.
- En la vista de diseño, seleccionamos el proceso *Synchronous*. La ventana *Navigator* muestra la vista lógica de BPEL, es decir, una vista estructurada del proceso de negocio.
- En la ventana *Navigator* expandimos *Imports*. (Vamos a utilizar *Synchronous.wsdl* para añadir las propiedades y alias de las propiedades).
- Pinchamos con el botón derecho sobre *Synchronous.wsdl* y elegimos **Add Property** del menú emergente.
- Como nombre de la propiedad pondremos *MyProperty*, y elegimos *string* como tipo de la propiedad en el árbol *Built-in Types*. Finalmente pulsamos OK.



Ahora necesitamos crear una *property alias* para especificar cómo se extraen los datos de correlación a partir de los mensajes. Como tenemos dos actividades *Receive* que reciben mensajes de tipos diferentes, necesitamos definir dos *property aliases*.

Creamos la primera *property alias*, con los siguientes pasos:

- En la ventana *Navigator* expandimos *Imports*.
- Pinchamos con el botón derecho sobre *Synchronous.wsdl* y elegimos **Add Property Alias** del menú emergente. Recuerda que necesitamos especificar la propiedad, parte del mensaje, y la *query* para crear una *property alias*. La **propiedad** especificada se usa para almacenar un *token* de correlación extraído. La **parte** del mensaje ayuda a establecer una correspondencia entre la *property alias* con algún mensaje específico y su parte. La *query* se utiliza para especificar qué datos en particular necesitan ser extraídos.
- En el cuadro de diálogo, pulsamos sobre *Browse*.
- Expandimos el nodo *Synchronous.wsdl*, seleccionamos *MyProperty* y pulsamos OK.
- En el cuadro de diálogo, expandimos el nodo *Synchronous.wsdl* y *requestMessage*.
- Seleccionamos *inputType typeA* como una parte del mensaje.
- Especificamos */ns:typeA/ns:id* en el campo de texto *Query* y pulsamos OK.

Para crear la segunda *property alias*, repetimos los pasos anteriores, teniendo en cuenta que elegiremos *Synchronous.wsdl > setPrefixRequest > part1* como parte del mensaje. Para la *Query* especificaremos */ns:typeB/ns:id*.

### 8.2.5. Creamos y añadimos los Correlation sets

---

Como ya hemos visto, un conjunto de correlación es una colección de propiedades que especifican qué datos deberían extraerse de los mensajes de entrada. Estos datos se utilizan posteriormente para identificar la instancia del proceso destinataria del mensaje.

Para **crear un conjunto de correlación** seguimos los siguientes pasos:

- Seleccionamos el proceso *Synchronous* en la vista de diseño, pulsamos el botón derecho del ratón y elegimos **Add > Correlation Set** en el menú emergente.
- En el cuadro de diálogo correspondiente elegimos como nombre: *MyCorrelationSet*, y pulsamos el botón *Add*.
- Aparecerá el cuadro de diálogo *Property Chooser*, expandimos el nodo *Synchronous.wsdl*, y seleccionamos *MyProperty*. Pulsamos OK para añadir la propiedad, y OK de nuevo en el cuadro de diálogo *Add Correlation Set* para crear el conjunto de correlación.

Después de crear el conjunto de correlación, necesitamos añadirlo a las actividades que reciben/envían mensajes.

Para **añadir el conjunto de correlación a la actividad *start*** tenemos que:

- Seleccionamos la actividad *start* en la vista de diseño, y pulsamos sobre el icono *Edit*

de la actividad.

- En la pestaña *Correlations*, pulsamos en el botón *Add*, y elegimos *MyCorrelationSet* en la lista desplegable.
- A continuación fijamos el valor de *MyCorrelationSet* en la columna *Initiate* a *yes*. Finalmente pusamos sobre *OK*.

Ahora necesitamos mapear los datos de correlación con la actividad *setPrefix*. Esto permitirá a la máquina BPEL asociar datos de entrada con instancias del proceso.

Para **añadir el conjunto de correlación a la actividad *SetPrefix*** tenemos que:

- Seleccionamos la actividad *SetPrefix* en la vista de diseño
- Pulsamos sobre el icono *Edit* de la actividad.
- En la pestaña *Correlations* pulsamos sobre *Add*, y elegimos *MyCorrelationSet*. Finalmente pulsamos *OK*. Tenemos que asegurarnos de que el valor de *MyCorrelationSet* en la columna *Inititate* tiene el valor *no*.

Finalmente grabamos todos los cambios con *File > Save All*.

### 8.2.6. Compilación y despliegue del proceso bpel

Para compilar y desplegar nuestro proceso tenemos que seguir los siguientes pasos:

- Seleccionamos la opción "Clean and build", en el menú emergente del proecto *Synchronous*. Esto generará un nuevo jar del proyecto.
- Borramos el componente JBI del proyecto *SynchronousApplication* (dentro de "JBI Modules")
- Añandimos de nuevo el componente JBI del proyecto *Synchronous*
- Desplegamos la *composite application* en el servidor de aplicaciones "Glassfish v2.x"

#### Cuidado

Puede que nos aparezca un error al desplegar la aplicación si previamente hemos desplegado la aplicación "SynchronousSample" del ejercicio de la sesión anterior. Ello es debido a que ambas comparten la misma dirección física de despliegue (recordad que hemos utilizado la misma plantilla). Para solucionar el problema, primero habrá que hacer "undeploy" del ensamblado de servicios del *runtime JBI* del ejercicio de la sesión anterior, y a continuación ya podremos desplegar sin problemas este ejercicio.

### 8.2.7. Pruebas y debugging del proceso BPEL

Vamos a comprobar que el proceso que hemos diseñado funciona bien. Monitorizaremos la ejecución del proceso para verificar que se devuelve la salida esperada. Para ello vamos a seguir los siguientes pasos:

- Añadimos un Test con el nombre "SendUserName", a partir del wsdl del proceso, y de la operación "operation1". Por ejemplo podemos poner como valor para *paramA* **Pepito**, y como valor de *id* **88**

- Añadimos un Test con el nombre "SendPrefixName", a partir del wsdl del proceso, y de la operación "setPrefix". Por ejemplo podemos poner como valor para *paramA* **Mister.**, y como valor de *id* **88**
- Pulsamos con el botón derecho del ratón sobre el proyecto *HelloWorldSampleCompApp* y elegimos *Debug Project (BPEL)* del menú emergente para comenzar a monitorizar la ejecución del proyecto.
- Esperamos hasta que el mensaje que indica que la sesión de *debug* ha comenzado en la ventana de la consola del *debugger* de BPEL.
- En la vista de diseño, seleccionamos la actividad *start*. Pulsamos sobre ella con el botón derecho del ratón y elegimos *Toggle Breakpoint*. Aparecerá un cuadrado rojo sobre la actividad con un *breakpoint*.
- Pulsamos con el botón derecho del ratón sobre la actividad *Assign1* y añadimos otro *breakpoint*.
- En la ventana de proyectos, expandimos *HelloWorldSampleCompApp* > *Test*.
- Pulsamos con el botón derecho sobre el test *SendUserName* y elegimos *Run* en el menú emergente. Las actividades por las que va pasando el programa aparecen marcadas con un indicador verde.
- Tan pronto como el proceso se detiene en su primer *breakpoint* pulsamos sobre un icono verde con una triángulo blanco en su interior, situado en la barra de herramientas (o alternativamente sobre *Continue* en el menú "Debug" de la barra de herramientas).
- El proceso continúa y se detiene en la actividad *SetPrefix*.
- En la ventana de proyectos, pulsamos con el botón derecho del ratón sobre el test *SendPrefix* y seleccionamos *Run*.
- El proceso continúa y se detiene en el segundo *breakpoint* de la actividad *AddHello*.
- Volvemos a pulsar sobre *Continue* en la barra de herramientas del *debugging* para continuar el proceso. (Cuando JUnit nos pregunte si queremos sobrescribir la salida responderemos afirmativamente).
- Al final del proceso, el test *SendPrefix* se ejecuta con éxito.
- En la ventana del proyectos, hacemos doble *click* sobre el nodo con la salida del test *SendUserName*. La salida del test *SendUserName* contiene la cadena esperada: *Hola Mister.Pepito*. Lo cual significa que el proceso ha funcionado bien y que la máquina BPEL ha manejado correctamente la comunicación asíncrona utilizando el conjunto de correlación *MyCorrelationSet*
- Seleccionamos *Run* > *Finish Debugger Session* desde el menú principal.

Podemos comprobar que si deshabilitamos el conjunto de correlación *MyCorrelationSet*, el proceso nunca alcanza la actividad *AddHello* ya que no hay datos de correlación que puedan ayudar a la máquina de servicios BPEL para enrutar el mensaje a la instancia correcta del proceso.

También podemos comprobar que la correlación funciona añadiendo un tercer Test, por ejemplo con nombre *SendPrefixFail*, en el que indiquemos un valor para "id" diferente, por ejemplo **55**. Cuando ejecutemos el test *SendUserName* se producirá una excepción.



