Arquitectura Orientada a Servicios

Servicios Web

Interfaz de Invocación Dinámica

Edwin Salvador

26 de junio de 2015

Sesión 12

Contenido I

1. La anotación @WebServiceRef
2. Interfaz de Invocación Dinámica
   - DII con un WSDL
   - Sin un WSDL
3. Creación de Servicios Web
   - Servicios web desde la vista del servidor
   - Implementación del servicio JAX-WS con el modelo de servlets
   - Implementación del servicio Web con el modelo EJB
   - Empaquetado y despliegue de un WS
   - Creación de WS con Netbeans
   - Creación de un WS a partir de un EJB
4. Ejercicio
5. Deber

La anotación @WebServiceRef

- Permite inyectar una referencia a un servicio web remoto (objeto `Service`).

La anotación @WebServiceRef

- Permite inyectar una referencia a un servicio web remoto (objeto `Service`).
- Gracias a esto podemos obtener una instancia del proxy de WS

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

La anotación @WebServiceRef

- Permite inyectar una referencia a un servicio web remoto (objeto `Service`).
- Gracias a esto podemos obtener una instancia del proxy de WS

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

- En este caso `MyService` es la clase que herede de `javax.xml.ws.Service` generada por `wsimport`.

La anotación @WebServiceRef

- Permite inyectar una referencia a un servicio web remoto (objeto `Service`).
- Gracias a esto podemos obtener una instancia del proxy de WS

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

- En este caso `MyService` es la clase que herede de `javax.xml.ws.Service` generada por `wsimport`.
- Solo podemos utilizar `@WebServiceRef` cuando nuestro cliente sea un servlet o un EJB pero no cuando sea una clase Java plana como un JSP. En ese caso debemos instanciar directamente a la clase:

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

Contenido I

1. La anotación @WebServiceRef
2. Interfaz de Invocación Dinámica
   - DII con un WSDL
   - Sin un WSDL
3. Creación de Servicios Web
   - Servicios web desde la vista del servidor
   - Implementación del servicio JAX-WS con el modelo de servlets
   - Implementación del servicio Web con el modelo EJB
   - Empaquetado y despliegue de un WS
   - Creación de WS con Netbeans
   - Creación de un WS a partir de un EJB
4. Ejercicio
5. Deber

Interfaz de Invocación Dinámica

- Con la Interfaz de Invocación Dinámica (DDI) no necesitamos crear el *stub* para invocar los métodos del servicio.

Interfaz de Invocación Dinámica

- Con la Interfaz de Invocación Dinámica (DDI) no necesitamos crear el *stub* para invocar los métodos del servicio.
- Podemos invocarlos dinámicamente para indicando el nombre del método como una cadena de texto y los parámetros como un array.

Interfaz de Invocación Dinámica

- Con la Interfaz de Invocación Dinámica (DDI) no necesitamos crear el *stub* para invocar los métodos del servicio.
- Podemos invocarlos dinámicamente para indicando el nombre del método como una cadena de texto y los parámetros como un array.
- Así se pueden implementar los *broker* de servicios que intentan localizar el servicio más apropiado para la tarea que deseamos realizar.

Interfaz de Invocación Dinámica

- Con la Interfaz de Invocación Dinámica (DDI) no necesitamos crear el *stub* para invocar los métodos del servicio.
- Podemos invocarlos dinámicamente para indicando el nombre del método como una cadena de texto y los parámetros como un array.
- Así se pueden implementar los *broker* de servicios que intentan localizar el servicio más apropiado para la tarea que deseamos realizar.
- El *broker* es completamente transparente al usuario. Una vez que localiza el servicio este nos devuelve los resultados.

Interfaz de Invocación Dinámica

- Con la Interfaz de Invocación Dinámica (DDI) no necesitamos crear el *stub* para invocar los métodos del servicio.
- Podemos invocarlos dinámicamente para indicando el nombre del método como una cadena de texto y los parámetros como un array.
- Así se pueden implementar los *broker* de servicios que intentan localizar el servicio más apropiado para la tarea que deseamos realizar.
- El *broker* es completamente transparente al usuario. Una vez que localiza el servicio este nos devuelve los resultados.
- Podemos utilizar la DII tanto si contamos con un WSDL o no.

Interfaz de Invocación Dinámica

- Con la Interfaz de Invocación Dinámica (DDI) no necesitamos crear el *stub* para invocar los métodos del servicio.
- Podemos invocarlos dinámicamente para indicando el nombre del método como una cadena de texto y los parámetros como un array.
- Así se pueden implementar los *broker* de servicios que intentan localizar el servicio más apropiado para la tarea que deseamos realizar.
- El *broker* es completamente transparente al usuario. Una vez que localiza el servicio este nos devuelve los resultados.
- Podemos utilizar la DII tanto si contamos con un WSDL o no.
- Se pierde la transparencia debido a que debemos especificar directamente los métodos y parámetros.

Contenido I

1. La anotación @WebServiceRef
2. Interfaz de Invocación Dinámica
   - DII con un WSDL
   - Sin un WSDL
3. Creación de Servicios Web
   - Servicios web desde la vista del servidor
   - Implementación del servicio JAX-WS con el modelo de servlets
   - Implementación del servicio Web con el modelo EJB
   - Empaquetado y despliegue de un WS
   - Creación de WS con Netbeans
   - Creación de un WS a partir de un EJB
4. Ejercicio
5. Deber

DII con un WSDL

- En el caso que se cuente con un WSDL se debe primero conseguir el objeto `Service`.

DII con un WSDL

- En el caso que se cuente con un WSDL se debe primero conseguir el objeto `Service`.
- `javax.xml.ws.Service` es la clase abstracta qeu representa el WSDL.

DII con un WSDL

- En el caso que se cuente con un WSDL se debe primero conseguir el objeto `Service`.
- `javax.xml.ws.Service` es la clase abstracta qeu representa el WSDL.
- La clase `Service` nos sirve para llamadas de forma estático (generando el *stub*) y para llamadas remotas de manera dinámica.

Ejemplo de uso dinámico de Service

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

// Utilizamos Call para las llamadas dinamicas a los metodos
    , indicando un puerto y operacion determinados.
Call call = serv.createCall(new QName("http://localhost
    :8080/HolaMundo", "holaPort"),new QName("http://
    localhost:8080/HolaMundo", "hello"));

// Invocamos a la llamada que hemos creado
String result = (String) call.invoke(
new Object[] { "Miguel" }); // array de objetos

// Para utilizar el valor devuelto debemos hacer un cast ya
    que nos devuelve un  Object
```

Contenido I

1. La anotación @WebServiceRef
2. Interfaz de Invocación Dinámica
   - DII con un WSDL
   - Sin un WSDL
3. Creación de Servicios Web
   - Servicios web desde la vista del servidor
   - Implementación del servicio JAX-WS con el modelo de servlets
   - Implementación del servicio Web con el modelo EJB
   - Empaquetado y despliegue de un WS
   - Creación de WS con Netbeans
   - Creación de un WS a partir de un EJB
4. Ejercicio
5. Deber

Sin un WSDL

```
// Creamos un objeto Service y proporcionamos solamente el nombre del servicio
ServiceFactory sf = ServiceFactory.newInstance();
Service serv = sf.createService(new QName("http://jtech.ua.es", "hola"));

// Con este objeto obtenemos el objeto Call para realizar la llamada al servicio
Call call = serv.createCall( new QName("http://localhost:8080/HolaMundo", "holaPort"),
     new QName("http://localhost:8080/HolaMundo", "hello"));

// El objeto Call no tendra info sobre el servicio (sin el WSDL) debemos indicarle
     explicitamente
call.setTargetEndpointAddress(endpoint);

// Indicar el tipo de datos que nos devuelve la operacion que vamos a invocar
QName t_string = new QName("http://www.w3.org/2001/XMLSchema", "string");
call.setReturnType(t_string);

// Indicamos los parametros de entrada que toma la operacion y sus tipos
QName t_string = new QName("http://www.w3.org/2001/XMLSchema", "string");
call.addParameter("string_1", t_string, ParameterMode.IN);

// Ahora si podemos invocar la operacion
String result = (String) call.invoke(
new Object[] { "Miguel" });
```

Arquitectura Orientada a Servicios

Creación de Servicios Web

Edwin Salvador

26 de junio de 2015

Sesión 12

Contenido I

1. La anotación @WebServiceRef
2. Interfaz de Invocación Dinámica
   - DII con un WSDL
   - Sin un WSDL
3. Creación de Servicios Web
   - Servicios web desde la vista del servidor
   - Implementación del servicio JAX-WS con el modelo de servlets
   - Implementación del servicio Web con el modelo EJB
   - Empaquetado y despliegue de un WS
   - Creación de WS con Netbeans
   - Creación de un WS a partir de un EJB
4. Ejercicio
5. Deber

Creación de Servicios Web

- Crearemos nuestros servicios web que podrán ser llamados desde cualquier parte de Internet.

Creación de Servicios Web

- Crearemos nuestros servicios web que podrán ser llamados desde cualquier parte de Internet.
- Debemos interpretar los mensajes SOAP entrantes como petición de un método. Invocar al método y construir un mensaje SOAP de respuesta para enviarlo al cliente.

Creación de Servicios Web

- Crearemos nuestros servicios web que podrán ser llamados desde cualquier parte de Internet.
- Debemos interpretar los mensajes SOAP entrantes como petición de un método. Invocar al método y construir un mensaje SOAP de respuesta para enviarlo al cliente.
- En lugar de interpretar y construir los mensajes SOAP manualmente (que sería muy costoso y propenso a errores) podemos utilizar la API JAX-WS

Creación de Servicios Web

- Crearemos nuestros servicios web que podrán ser llamados desde cualquier parte de Internet.
- Debemos interpretar los mensajes SOAP entrantes como petición de un método. Invocar al método y construir un mensaje SOAP de respuesta para enviarlo al cliente.
- En lugar de interpretar y construir los mensajes SOAP manualmente (que sería muy costoso y propenso a errores) podemos utilizar la API JAX-WS
- Además utilizaremos herramientas que nos permitan generar las clases que permitan leer el mensaje SOAP, invocar el método y construir el mensaje SOAP de respuesta.

Creación de Servicios Web

- Crearemos nuestros servicios web que podrán ser llamados desde cualquier parte de Internet.
- Debemos interpretar los mensajes SOAP entrantes como petición de un método. Invocar al método y construir un mensaje SOAP de respuesta para enviarlo al cliente.
- En lugar de interpretar y construir los mensajes SOAP manualmente (que sería muy costoso y propenso a errores) podemos utilizar la API JAX-WS
- Además utilizaremos herramientas que nos permitan generar las clases que permitan leer el mensaje SOAP, invocar el método y construir el mensaje SOAP de respuesta.
- Esto ayuda a centrarse en la funcionalidad del servicio y no en la invocación de estos.

Creación de Servicios Web

- Crearemos nuestros servicios web que podrán ser llamados desde cualquier parte de Internet.
- Debemos interpretar los mensajes SOAP entrantes como petición de un método. Invocar al método y construir un mensaje SOAP de respuesta para enviarlo al cliente.
- En lugar de interpretar y construir los mensajes SOAP manualmente (que sería muy costoso y propenso a errores) podemos utilizar la API JAX-WS
- Además utilizaremos herramientas que nos permitan generar las clases que permitan leer el mensaje SOAP, invocar el método y construir el mensaje SOAP de respuesta.
- Esto ayuda a centrarse en la funcionalidad del servicio y no en la invocación de estos.
- Utilizaremos Netbeans y Glassfish.

Contenido I

1. La anotación @WebServiceRef
2. Interfaz de Invocación Dinámica
   - DII con un WSDL
   - Sin un WSDL
3. Creación de Servicios Web
   - Servicios web desde la vista del servidor
   - Implementación del servicio JAX-WS con el modelo de servlets
   - Implementación del servicio Web con el modelo EJB
   - Empaquetado y despliegue de un WS
   - Creación de WS con Netbeans
   - Creación de un WS a partir de un EJB
4. Ejercicio
5. Deber

Servicios web desde la vista del servidor

- Un WSDL define la interoperabilidad de los servicios web (métodos, parámetros y formatos).

Servicios web desde la vista del servidor

- Un WSDL define la interoperabilidad de los servicios web (métodos, parámetros y formatos).
- Un WSDL no impone un modelo de programación del cliente o servidor.

Servicios web desde la vista del servidor

- Un WSDL define la interoperabilidad de los servicios web (métodos, parámetros y formatos).
- Un WSDL no impone un modelo de programación del cliente o servidor.
- La especificación de SW para Java EE define tres formas de implementar la lógica de negocio de SW:

Servicios web desde la vista del servidor

- Un WSDL define la interoperabilidad de los servicios web (métodos, parámetros y formatos).
- Un WSDL no impone un modelo de programación del cliente o servidor.
- La especificación de SW para Java EE define tres formas de implementar la lógica de negocio de SW:
  - Como un Bean de sesión sin estado: El port se crea como un bean se sesión sin estado, que implementa los métodos de SEI (Service Endpoint Interface)

Servicios web desde la vista del servidor

- Un WSDL define la interoperabilidad de los servicios web (métodos, parámetros y formatos).
- Un WSDL no impone un modelo de programación del cliente o servidor.
- La especificación de SW para Java EE define tres formas de implementar la lógica de negocio de SW:
  - Como un Bean de sesión sin estado: El port se crea como un bean se sesión sin estado, que implementa los métodos de SEI (Service Endpoint Interface)
  - Como una clase Java: El port se implementa como un Servlet JAX-WS.

Servicios web desde la vista del servidor

- Un WSDL define la interoperabilidad de los servicios web (métodos, parámetros y formatos).
- Un WSDL no impone un modelo de programación del cliente o servidor.
- La especificación de SW para Java EE define tres formas de implementar la lógica de negocio de SW:
  - Como un Bean de sesión sin estado: El port se crea como un bean se sesión sin estado, que implementa los métodos de SEI (Service Endpoint Interface)
  - Como una clase Java: El port se implementa como un Servlet JAX-WS.
  - Como un Singleton Session Bean: el port se crea como un singleton session bean que implementa los métodos del SEI.

El componente **Port**

- Define la vista del servidor de un WS.

El componente **Port**

- Define la vista del servidor de un WS.
- Proporciona un servicio en una URL particular definida en el WSDL.

El componente **Port**

- Define la vista del servidor de un WS.
- Proporciona un servicio en una URL particular definida en el WSDL.
- Sirve una petición de operacion definida en el `<portType>`.

El componente **Port**

- Define la vista del servidor de un WS.
- Proporciona un servicio en una URL particular definida en el WSDL.
- Sirve una petición de operacion definida en el `<portType>`.
- El *Service Implementation Bean* depende del contenedor del `port`, generalmente es una clase Java que implementa los métodos definidos en el SEI.

El componente **Port**

- Define la vista del servidor de un WS.
- Proporciona un servicio en una URL particular definida en el WSDL.
- Sirve una petición de operacion definida en el `<portType>`.
- El *Service Implementation Bean* depende del contenedor del `port`, generalmente es una clase Java que implementa los métodos definidos en el SEI.
- El SEI es un mapeo del `<portType>` y el `<binding>` asociado a un `<port>`.

El componente **Port**

- Define la vista del servidor de un WS.
- Proporciona un servicio en una URL particular definida en el WSDL.
- Sirve una petición de operacion definida en el `<portType>`.
- El *Service Implementation Bean* depende del contenedor del `port`, generalmente es una clase Java que implementa los métodos definidos en el SEI.
- El SEI es un mapeo del `<portType>` y el `<binding>` asociado a un `<port>`.
- Un WS es un conjunto de `ports` que difieren en su URL mapeados en `ports` separados.

El componente **Port**

- Define la vista del servidor de un WS.
- Proporciona un servicio en una URL particular definida en el WSDL.
- Sirve una petición de operacion definida en el `<portType>`.
- El *Service Implementation Bean* depende del contenedor del `port`, generalmente es una clase Java que implementa los métodos definidos en el SEI.
- El SEI es un mapeo del `<portType>` y el `<binding>` asociado a un `<port>`.
- Un WS es un conjunto de `ports` que difieren en su URL mapeados en `ports` separados.
- Un `port` es creado e inicializado por el contenedor antes de que la primera llamada recibida sea sea servida.

El componente **Port**

- Define la vista del servidor de un WS.
- Proporciona un servicio en una URL particular definida en el WSDL.
- Sirve una petición de operacion definida en el `<portType>`.
- El *Service Implementation Bean* depende del contenedor del `port`, generalmente es una clase Java que implementa los métodos definidos en el SEI.
- El SEI es un mapeo del `<portType>` y el `<binding>` asociado a un `<port>`.
- Un WS es un conjunto de `ports` que difieren en su URL mapeados en `ports` separados.
- Un `port` es creado e inicializado por el contenedor antes de que la primera llamada recibida sea sea servida.
- El contenedor se encarga de destruir el `port` cuando sea necesario.

El componente **Port**

- Define la vista del servidor de un WS.
- Proporciona un servicio en una URL particular definida en el WSDL.
- Sirve una petición de operacion definida en el `<portType>`.
- El *Service Implementation Bean* depende del contenedor del `port`, generalmente es una clase Java que implementa los métodos definidos en el SEI.
- El SEI es un mapeo del `<portType>` y el `<binding>` asociado a un `<port>`.
- Un WS es un conjunto de `ports` que difieren en su URL mapeados en `ports` separados.
- Un `port` es creado e inicializado por el contenedor antes de que la primera llamada recibida sea sea servida.
- El contenedor se encarga de destruir el `port` cuando sea necesario.
- Un contenedor Web puede desplegarse en un servidor web o servidor de aplicaciones, un contenedor EJB solo en un servidor de aplicaciones.

El componente **Port**

- Asocia un URL con la implementación del servicio.

El componente **Port**

- Asocia un URL con la implementación del servicio.
- Pospone la definición de los requerimientos del contenedor del servicio a la fase de despliegue.

El componente **Port**

- Asocia un URL con la implementación del servicio.
- Pospone la definición de los requerimientos del contenedor del servicio a la fase de despliegue.
- Un contenedor porporciona un *listener* en la dirección del puerto y un mecanismo para para enviar la petición a la implementación del WS.

El componente **Port**

- Asocia un URL con la implementación del servicio.
- Pospone la definición de los requerimientos del contenedor del servicio a la fase de despliegue.
- Un contenedor porporciona un *listener* en la dirección del puerto y un mecanismo para para enviar la petición a la implementación del WS.
- Un contenedor también proporciona servicios en tiempo de ejecución como restricciones de seguridad y mapeos de referencia lógicas a referencias físicas de objetos distribuidos y recursos.

El componente **Port**

- Asocia un URL con la implementación del servicio.
- Pospone la definición de los requerimientos del contenedor del servicio a la fase de despliegue.
- Un contenedor porporciona un *listener* en la dirección del puerto y un mecanismo para para enviar la petición a la implementación del WS.
- Un contenedor también proporciona servicios en tiempo de ejecución como restricciones de seguridad y mapeos de referencia lógicas a referencias físicas de objetos distribuidos y recursos.
- El empaquetado de un WS en un módulo Java EE se lo hace en un EJB-JAR o un WAR que contiene las clases Java del SEI y los WSDL del WS.

Implementación de un WS

- Podemos elegir entre dos puntos de partida: desde una clase Java o desde un WSDL.

Implementación de un WS

- Podemos elegir entre dos puntos de partida: desde una clase Java o desde un WSDL.
- **Si empezamos desde la clase Java**

Implementación de un WS

- Podemos elegir entre dos puntos de partida: desde una clase Java o desde un WSDL.
- **Si empezamos desde la clase Java**
  - Utilizaremos herramientas para generar los artefactos como el WSDL.

Implementación de un WS

- Podemos elegir entre dos puntos de partida: desde una clase Java o desde un WSDL.
- **Si empezamos desde la clase Java**
  - Utilizaremos herramientas para generar los artefactos como el WSDL.
  - Tendremos la seguridad de que la clase que implementa el servicio tiene los tipos de Java adecuados pero el desarrollador tiene menos control sobre el XML generado.

Implementación de un WS

- Podemos elegir entre dos puntos de partida: desde una clase Java o desde un WSDL.
- **Si empezamos desde la clase Java**
  - Utilizaremos herramientas para generar los artefactos como el WSDL.
  - Tendremos la seguridad de que la clase que implementa el servicio tiene los tipos de Java adecuados pero el desarrollador tiene menos control sobre el XML generado.
- **Si empezamos desde el WSDL (y el XSD)**

Implementación de un WS

- Podemos elegir entre dos puntos de partida: desde una clase Java o desde un WSDL.
- **Si empezamos desde la clase Java**
  - Utilizaremos herramientas para generar los artefactos como el WSDL.
  - Tendremos la seguridad de que la clase que implementa el servicio tiene los tipos de Java adecuados pero el desarrollador tiene menos control sobre el XML generado.
- **Si empezamos desde el WSDL (y el XSD)**
  - Utilizaremos las herramientas para generar el SEI.

Implementación de un WS

- Podemos elegir entre dos puntos de partida: desde una clase Java o desde un WSDL.
- **Si empezamos desde la clase Java**
  - Utilizaremos herramientas para generar los artefactos como el WSDL.
  - Tendremos la seguridad de que la clase que implementa el servicio tiene los tipos de Java adecuados pero el desarrollador tiene menos control sobre el XML generado.
- **Si empezamos desde el WSDL (y el XSD)**
  - Utilizaremos las herramientas para generar el SEI.
  - Se tiene el control total sobre que esquema se utiliza pero menos control sobre el *endpoint* del servicio generado y las clases que utiliza.

Implementación de un WS desde una clase Java

- Tenemos que guiarnos por ciertas restricciones:

Implementación de un WS desde una clase Java

- Tenemos que guiarnos por ciertas restricciones:
  - La clase debe estar anotada con `javax.jws.WebService`.

Implementación de un WS desde una clase Java

- Tenemos que guiarnos por ciertas restricciones:
  - La clase debe estar anotada con `javax.jws.WebService`.
  - Podemos anotar los métodos con `javax.jws.WebMethod`.

Implementación de un WS desde una clase Java

- Tenemos que guiarnos por ciertas restricciones:
  - La clase debe estar anotada con `javax.jws.WebService`.
  - Podemos anotar los métodos con `javax.jws.WebMethod`.
  - Los métodos pueden lanzar la excepción `java.rmi.RemoteException`.

Implementación de un WS desde una clase Java

- Tenemos que guiarnos por ciertas restricciones:
  - La clase debe estar anotada con `javax.jws.WebService`.
  - Podemos anotar los métodos con `javax.jws.WebMethod`.
  - Los métodos pueden lanzar la excepción `java.rmi.RemoteException`.
  - Los parámetros de los métodos y tipos de retorno deben ser compatibles con JAXB (reglas para mapear tipos Java con tipos de XSD)

Implementación de un WS desde una clase Java

- Tenemos que guiarnos por ciertas restricciones:
  - La clase debe estar anotada con `javax.jws.WebService`.
  - Podemos anotar los métodos con `javax.jws.WebMethod`.
  - Los métodos pueden lanzar la excepción `java.rmi.RemoteException`.
  - Los parámetros de los métodos y tipos de retorno deben ser compatibles con JAXB (reglas para mapear tipos Java con tipos de XSD)
  - Ningún parámetro o tipo de retorno pueden implementar la interfaz `java.rmi.Remote` ni directa ni indirectamente.

Implementación de un WS desde una clase Java

- Tenemos que guiarnos por ciertas restricciones:
  - La clase debe estar anotada con `javax.jws.WebService`.
  - Podemos anotar los métodos con `javax.jws.WebMethod`.
  - Los métodos pueden lanzar la excepción `java.rmi.RemoteException`.
  - Los parámetros de los métodos y tipos de retorno deben ser compatibles con JAXB (reglas para mapear tipos Java con tipos de XSD)
  - Ningún parámetro 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 manera **implícita** entonces no necesitamos proporcionar una interfaz.

Implementación de un WS desde una clase Java

- Tenemos que guiarnos por ciertas restricciones:
  - La clase debe estar anotada con `javax.jws.WebService`.
  - Podemos anotar los métodos con `javax.jws.WebMethod`.
  - Los métodos pueden lanzar la excepción `java.rmi.RemoteException`.
  - Los parámetros de los métodos y tipos de retorno deben ser compatibles con JAXB (reglas para mapear tipos Java con tipos de XSD)
  - Ningún parámetro 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 manera **implícita** entonces no necesitamos proporcionar una interfaz.
- Si queremos proporcionar la interfaz de manera **explícita** debemos añadir el atributo `endpointInterface` a la anotación `@WebService` y debemos proporcionar la interfaz que indique los métodos públicos disponibles que implementa el servicio.

Contenido I

1. La anotación @WebServiceRef
2. Interfaz de Invocación Dinámica
   - DII con un WSDL
   - Sin un WSDL
3. Creación de Servicios Web
   - Servicios web desde la vista del servidor
   - Implementación del servicio JAX-WS con el modelo de servlets
   - Implementación del servicio Web con el modelo EJB
   - Empaquetado y despliegue de un WS
   - Creación de WS con Netbeans
   - Creación de un WS a partir de un EJB
4. Ejercicio
5. Deber

Implementación del servicio JAX-WS con el modelo de servlets

Tienen las siguientes restricciones:

- La clase debe estar anotada (`javax.jws.WebService`)

Implementación del servicio JAX-WS con el modelo de servlets

Tienen las siguientes restricciones:

- La clase debe estar anotada (`javax.jws.WebService`)
- Si empezamos con clase Java: La anotación @WebService define una interfaz (SEI) implícitamente. Los métodos de negocio deben ser **públicos** y NO **final** o **static**. Solo los métodos con `@WebMethod` son expuestos al cliente.

Implementación del servicio JAX-WS con el modelo de servlets

Tienen las siguientes restricciones:

- La clase debe estar anotada (`javax.jws.WebService`)
- Si empezamos con clase Java: La anotación @WebService define una interfaz (SEI) implícitamente. Los métodos de negocio deben ser **públicos** y NO **final** o **static**. Solo los métodos con `@WebMethod` son expuestos al cliente.
- Si empezamos con el WSDL: el SEI generado desde el WSDL y la implementación del servicio deben estar anotados con `javax.jws.WebService`. Se deben implementar todos los métodos indicados en el SEI y métodos adicionales. Los métodos de negocio deben ser **públicos** y NO **final** o **static**.

Implementación del servicio JAX-WS con el modelo de servlets

Tienen las siguientes restricciones:

- La clase debe estar anotada (`javax.jws.WebService`)
- Si empezamos con clase Java: La anotación @WebService define una interfaz (SEI) implícitamente. Los métodos de negocio deben ser **públicos** y NO **final** o **static**. Solo los métodos con `@WebMethod` son expuestos al cliente.
- Si empezamos con el WSDL: el SEI generado desde el WSDL y la implementación del servicio deben estar anotados con `javax.jws.WebService`. Se deben implementar todos los métodos indicados en el SEI y métodos adicionales. Los métodos de negocio deben ser **públicos** y NO **final** o **static**.
- La implementación de servicio debe tener un constructor público por defecto.

Implementación del servicio JAX-WS con el modelo de servlets

Tienen las siguientes restricciones:

- La clase debe estar anotada (`javax.jws.WebService`)
- Si empezamos con clase Java: La anotación @WebService define una interfaz (SEI) implícitamente. Los métodos de negocio deben ser **públicos** y NO **final** o **static**. Solo los métodos con `@WebMethod` son expuestos al cliente.
- Si empezamos con el WSDL: el SEI generado desde el WSDL y la implementación del servicio deben estar anotados con `javax.jws.WebService`. Se deben implementar todos los métodos indicados en el SEI y métodos adicionales. Los métodos de negocio deben ser **públicos** y NO **final** o **static**.
- La implementación de servicio debe tener un constructor público por defecto.
- La implementación de servicio no debe guardar el estado.

Implementación del servicio JAX-WS con el modelo de servlets

Tienen las siguientes restricciones:

- La clase debe estar anotada (`javax.jws.WebService`)
- Si empezamos con clase Java: La anotación @WebService define una interfaz (SEI) implícitamente. Los métodos de negocio deben ser **públicos** y NO **final** o **static**. Solo los métodos con `@WebMethod` son expuestos al cliente.
- Si empezamos con el WSDL: el SEI generado desde el WSDL y la implementación del servicio deben estar anotados con `javax.jws.WebService`. Se deben implementar todos los métodos indicados en el SEI y métodos adicionales. Los métodos de negocio deben ser **públicos** y NO **final** o **static**.
- La implementación de servicio debe tener un constructor público por defecto.
- La implementación de servicio no debe guardar el estado.
- La clase debe ser pública NO final ni abstracta.

Implementación del servicio JAX-WS con el modelo de servlets

Tienen las siguientes restricciones:

- La clase debe estar anotada (`javax.jws.WebService`)
- Si empezamos con clase Java: La anotación @WebService define una interfaz (SEI) implícitamente. Los métodos de negocio deben ser **públicos** y NO **final** o **static**. Solo los métodos con `@WebMethod` son expuestos al cliente.
- Si empezamos con el WSDL: el SEI generado desde el WSDL y la implementación del servicio deben estar anotados con `javax.jws.WebService`. Se deben implementar todos los métodos indicados en el SEI y métodos adicionales. Los métodos de negocio deben ser **públicos** y NO **final** o **static**.
- La implementación de servicio debe tener un constructor público por defecto.
- La implementación de servicio no debe guardar el estado.
- La clase debe ser pública NO final ni abstracta.
- La clase no debe implementar el método `finalize()`.

Pasos para crear un WS con JAX-WS

- Codificar la clase que implementa el servicio

Pasos para crear un WS con JAX-WS

- Codificar la clase que implementa el servicio
- Compilar la clase que implementa el servicio

Pasos para crear un WS con JAX-WS

- Codificar la clase que implementa el servicio
- Compilar la clase que implementa el servicio
- Empaquetar los ficheros en un war

Pasos para crear un WS con JAX-WS

- Codificar la clase que implementa el servicio
- Compilar la clase que implementa el servicio
- Empaquetar los ficheros en un war
- Desplegar el war. Los artefactos del WS necesarios para comunicarse con los clientes serán generados por Glassfish durante el despliegue.

Ejemplo de un WS

```
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 + ".";
  }
}
```

- Esto sería suficiente para implementar un WS sin tener conocimiento de librerías adicionales.

Ejemplo de un WS

```java
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 + ".";
  }
}
```

- Esto sería suficiente para implementar un WS sin tener conocimiento de librerías adicionales.
- Ya que es un componente web podríamos acceder a métodos de los *servlets* como `HttpServletRequest` y `HttpSession`, etc

Anotaciones JAX-WS

- Como dijimos anteriormente la clase que implementa el servicio debe hacer uso de las anotaciones JAX-WS. Estas anotaciones son:

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

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

Anotaciones JAX-WS

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

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

Anotaciones JAX-WS

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

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

Anotaciones JAX-WS

`@WebFault`	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 *SOAP Fault* que nos indique el error producido. En el lado del cliente la clase con dicha excepción se habrá generado en el *stub* para el acceso al servicio, y al recibir el mensaje SOAP con el error el *stub* lanzará la excepción correspondiente. Es decir, para el desarrollador será como si la excepción saltase directamente desde el servicio hasta el cliente.

```
@WebFault
public class ConversionFaultException extends
Exception {
    public ConversionFaultException(String msg)
{
        super(msg);
    }
}
```

Anotación `@SOAPBinding`

- Con esta anotación podemos cambiar el estilo y la codificacion del servicio. Estos estilos son:

Anotación `@SOAPBinding`

- Con esta anotación podemos cambiar el estilo y la codificacion del servicio. Estos estilos son:
  - `SOAPBinding.Style.RPC`: Se utilizan mensajes SOAP RPC en los que se codifican en XML las llamadas a métodos remotos.

Anotación `@SOAPBinding`

- Con esta anotación podemos cambiar el estilo y la codificacion del servicio. Estos estilos son:
  - `SOAPBinding.Style.RPC`: Se utilizan mensajes SOAP RPC en los que se codifican en XML las llamadas a métodos remotos.
  - `SOAPBinding.style.DOCUMENT` Se utilizan mensajes SOAP orientados al documento. También se pueden utilizar par invocar operaciones de servicios.

Anotación `@SOAPBinding`

- Con esta anotación podemos cambiar el estilo y la codificacion del servicio. Estos estilos son:
  - `SOAPBinding.Style.RPC`: Se utilizan mensajes SOAP RPC en los que se codifican en XML las llamadas a métodos remotos.
  - `SOAPBinding.style.DOCUMENT` Se utilizan mensajes SOAP orientados al documento. También se pueden utilizar par invocar operaciones de servicios.
- Podemos especificar la codificación:

Anotación `@SOAPBinding`

- Con esta anotación podemos cambiar el estilo y la codificacion del servicio. Estos estilos son:
  - `SOAPBinding.Style.RPC`: Se utilizan mensajes SOAP RPC en los que se codifican en XML las llamadas a métodos remotos.
  - `SOAPBinding.style.DOCUMENT` Se utilizan mensajes SOAP orientados al documento. También se pueden utilizar par invocar operaciones de servicios.
- Podemos especificar la codificación:
  - `SOAPBinding.Use.LITERAL`: Aceptada pen el WS-I Basic Profile que da soporte a servicio del tipo *document/literal* y *RPC/literal*

Anotación `@SOAPBinding`

- Con esta anotación podemos cambiar el estilo y la codificacion del servicio. Estos estilos son:
  - `SOAPBinding.Style.RPC`: Se utilizan mensajes SOAP RPC en los que se codifican en XML las llamadas a métodos remotos.
  - `SOAPBinding.style.DOCUMENT` Se utilizan mensajes SOAP orientados al documento. También se pueden utilizar par invocar operaciones de servicios.
- Podemos especificar la codificación:
  - `SOAPBinding.Use.LITERAL`: Aceptada pen el WS-I Basic Profile que da soporte a servicio del tipo *document/literal* y *RPC/literal*
  - `SOAPBinding.Use.ENCODED`: permite representar mayor variedad de estructuras de datos que la anterior. No aceptada por incompatibilidades entre servicios.

Anotación `@SOAPBinding`

- Con esta anotación podemos cambiar el estilo y la codificacion del servicio. Estos estilos son:
  - `SOAPBinding.Style.RPC`: Se utilizan mensajes SOAP RPC en los que se codifican en XML las llamadas a métodos remotos.
  - `SOAPBinding.style.DOCUMENT` Se utilizan mensajes SOAP orientados al documento. También se pueden utilizar par invocar operaciones de servicios.
- Podemos especificar la codificación:
  - `SOAPBinding.Use.LITERAL`: Aceptada pen el WS-I Basic Profile que da soporte a servicio del tipo *document/literal* y *RPC/literal*
  - `SOAPBinding.Use.ENCODED`: permite representar mayor variedad de estructuras de datos que la anterior. No aceptada por incompatibilidades entre servicios.
- Con los servicios tipo *document/literal* podemos indicar la forma en la que se presentan los datos:

Anotación `@SOAPBinding`

- Con esta anotación podemos cambiar el estilo y la codificacion del servicio. Estos estilos son:
  - `SOAPBinding.Style.RPC`: Se utilizan mensajes SOAP RPC en los que se codifican en XML las llamadas a métodos remotos.
  - `SOAPBinding.style.DOCUMENT` Se utilizan mensajes SOAP orientados al documento. También se pueden utilizar par invocar operaciones de servicios.
- Podemos especificar la codificación:
  - `SOAPBinding.Use.LITERAL`: Aceptada pen el WS-I Basic Profile que da soporte a servicio del tipo *document/literal* y *RPC/literal*
  - `SOAPBinding.Use.ENCODED`: permite representar mayor variedad de estructuras de datos que la anterior. No aceptada por incompatibilidades entre servicios.
- Con los servicios tipo *document/literal* podemos indicar la forma en la que se presentan los datos:
  - `SOAPBinding.ParameterStyle.BARE`: se pasan directamente.

Anotación `@SOAPBinding`

- Con esta anotación podemos cambiar el estilo y la codificacion del servicio. Estos estilos son:
  - `SOAPBinding.Style.RPC`: Se utilizan mensajes SOAP RPC en los que se codifican en XML las llamadas a métodos remotos.
  - `SOAPBinding.style.DOCUMENT` Se utilizan mensajes SOAP orientados al documento. También se pueden utilizar par invocar operaciones de servicios.
- Podemos especificar la codificación:
  - `SOAPBinding.Use.LITERAL`: Aceptada pen el WS-I Basic Profile que da soporte a servicio del tipo *document/literal* y *RPC/literal*
  - `SOAPBinding.Use.ENCODED`: permite representar mayor variedad de estructuras de datos que la anterior. No aceptada por incompatibilidades entre servicios.
- Con los servicios tipo *document/literal* podemos indicar la forma en la que se presentan los datos:
  - `SOAPBinding.ParameterStyle.BARE`: se pasan directamente.
  - `SOAPBinding.ParameterStyle.WRAPPED`: se pasan envueltos en tipos de datos complejos. Esta es la opción por defecto.

Tipos de datos compatibles

- Podemos utilizar los tipos soportados por JAXB: básicos y wrappers.

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

Tipos de datos compatibles

- Otros tipos de datos:

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

Tipos de datos compatibles

- Otros tipos de datos:

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

- Colecciones y genéricos

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

Ejemplo de WS (endpoint) JAX-WS

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

import javax.jws.WebService;

@WebService
public class ConversionSW {
  public ConversionSW() { }

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

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

Ejemplo usando anotaciones

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

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

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

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

Contenido I

1. La anotación @WebServiceRef
2. Interfaz de Invocación Dinámica
   - DII con un WSDL
   - Sin un WSDL
3. Creación de Servicios Web
   - Servicios web desde la vista del servidor
   - Implementación del servicio JAX-WS con el modelo de servlets
   - Implementación del servicio Web con el modelo EJB
   - Empaquetado y despliegue de un WS
   - Creación de WS con Netbeans
   - Creación de un WS a partir de un EJB
4. Ejercicio
5. Deber

Implementación del servicio Web con el modelo EJB

- Podemos utilizar un *Stateless Session Bean* para implementar el servicio. El componente `port residirá en un contenedor EJB`

Implementación del servicio Web con el modelo EJB

- Podemos utilizar un *Stateless Session Bean* para implementar el servicio. El componente `port residirá en un contenedor EJB`
- Se utilizan las mismas anotaciones, normas y requisitos que hemos mencionado en la sección anterior.

Implementación del servicio Web con el modelo EJB

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

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

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

Contenido I

1. La anotación @WebServiceRef
2. Interfaz de Invocación Dinámica
   - DII con un WSDL
   - Sin un WSDL
3. Creación de Servicios Web
   - Servicios web desde la vista del servidor
   - Implementación del servicio JAX-WS con el modelo de servlets
   - Implementación del servicio Web con el modelo EJB
   - Empaquetado y despliegue de un WS
   - Creación de WS con Netbeans
   - Creación de un WS a partir de un EJB
4. Ejercicio
5. Deber

Empaquetado y despliegue de un WS

- Si utilizamos el componente web (Servlets), el WS debe ser empaquetado en un WAR, si usamos la aplicación EJB debe ser empaquetado en un ejb-jar.

Empaquetado y despliegue de un WS

- Si utilizamos el componente web (Servlets), el WS debe ser empaquetado en un WAR, si usamos la aplicación EJB debe ser empaquetado en un ejb-jar.
- El desarrollador debe empaquetar los elementos:

Empaquetado y despliegue de un WS

- Si utilizamos el componente web (Servlets), el WS debe ser empaquetado en un WAR, si usamos la aplicación EJB debe ser empaquetado en un ejb-jar.
- El desarrollador debe empaquetar los elementos:
  - WSDL (opcional con anotaciones JAX-WS)

Empaquetado y despliegue de un WS

- Si utilizamos el componente web (Servlets), el WS debe ser empaquetado en un WAR, si usamos la aplicación EJB debe ser empaquetado en un ejb-jar.
- El desarrollador debe empaquetar los elementos:
  - WSDL (opcional con anotaciones JAX-WS)
  - Clase SEI (opcional con JAX-WS)

Empaquetado y despliegue de un WS

- Si utilizamos el componente web (Servlets), el WS debe ser empaquetado en un WAR, si usamos la aplicación EJB debe ser empaquetado en un ejb-jar.
- El desarrollador debe empaquetar los elementos:
  - WSDL (opcional con anotaciones JAX-WS)
  - Clase SEI (opcional con JAX-WS)
  - Clase que implementa el servicio y clases dependientes

Empaquetado y despliegue de un WS

- Si utilizamos el componente web (Servlets), el WS debe ser empaquetado en un WAR, si usamos la aplicación EJB debe ser empaquetado en un ejb-jar.
- El desarrollador debe empaquetar los elementos:
  - WSDL (opcional con anotaciones JAX-WS)
  - Clase SEI (opcional con JAX-WS)
  - Clase que implementa el servicio y clases dependientes
  - Los artefactos portables generados por JAX-WS (ayudan al marshal/unmarshal de invocaciones y repuestas del WS)

Empaquetado y despliegue de un WS

- Si utilizamos el componente web (Servlets), el WS debe ser empaquetado en un WAR, si usamos la aplicación EJB debe ser empaquetado en un ejb-jar.
- El desarrollador debe empaquetar los elementos:
  - WSDL (opcional con anotaciones JAX-WS)
  - Clase SEI (opcional con JAX-WS)
  - Clase que implementa el servicio y clases dependientes
  - Los artefactos portables generados por JAX-WS (ayudan al marshal/unmarshal de invocaciones y repuestas del WS)
  - Descriptor del despliegue en un módulo Java EE (opcional con anotaciones JAX-WS)

Empaquetado y despliegue de un WS

- Si utilizamos el componente web (Servlets), el WS debe ser empaquetado en un WAR, si usamos la aplicación EJB debe ser empaquetado en un ejb-jar.
- El desarrollador debe empaquetar los elementos:
  - WSDL (opcional con anotaciones JAX-WS)
  - Clase SEI (opcional con JAX-WS)
  - Clase que implementa el servicio y clases dependientes
  - Los artefactos portables generados por JAX-WS (ayudan al marshal/unmarshal de invocaciones y repuestas del WS)
  - Descriptor del despliegue en un módulo Java EE (opcional con anotaciones JAX-WS)
- El descriptor `webservices.xml` y el `wsdl` se almacena en el directorio `META-INF` en el caso de un EJB y en el directorio `WEB-INF` en el caso de un componente web.

Descriptores

- Glassfish 3.1 ya contiene todas las librerias y clases necesarias para desarrollar y desplegar WS entonces no es necesario incluir ningún descriptor para e despliegue.

Descriptores

- Glassfish 3.1 ya contiene todas las librerias y clases necesarias para desarrollar y desplegar WS entonces no es necesario incluir ningún descriptor para e despliegue.
- Sin embargo los descriptores son necesarios si queremos que nuestro WS sea portable, por lo tanto deberíamos incluir las librerías JAX-WS necesarias en el `war`.

Descriptores

- Glassfish 3.1 ya contiene todas las librerias y clases necesarias para desarrollar y desplegar WS entonces no es necesario incluir ningún descriptor para e despliegue.
- Sin embargo los descriptores son necesarios si queremos que nuestro WS sea portable, por lo tanto deberíamos incluir las librerías JAX-WS necesarias en el `war`.
- Los descriptores también son necesarios par que nuestro WS sea descubierto automáticamente por el contenedor, por lo tanto debemos incluir algunos descriptores para especificar como queremos que se desplieguen nuestros WS.

Despliegue de un WS

- JAX-WS soporta dos modelos de despliegue:

Despliegue de un WS

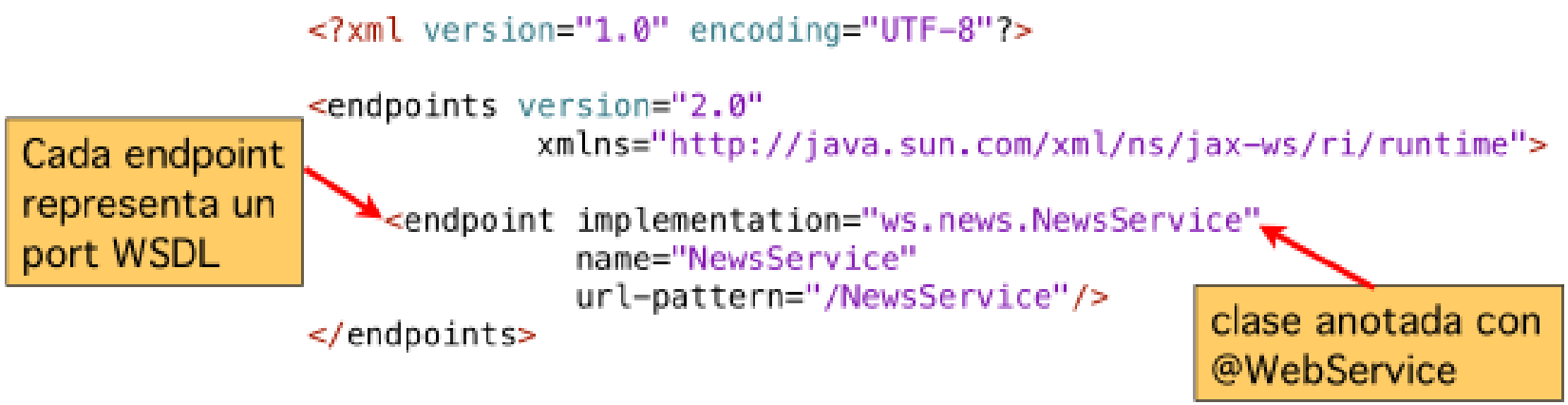
- JAX-WS soporta dos modelos de despliegue:
  - Uno utiliza el fichero `webservices.xml`

Despliegue de un WS

- JAX-WS soporta dos modelos de despliegue:
  - Uno utiliza el fichero `webservices.xml`
  - El otro utiliza los ficheros `web.xml` y `sun-jaxws.xml`

Despliegue de un WS

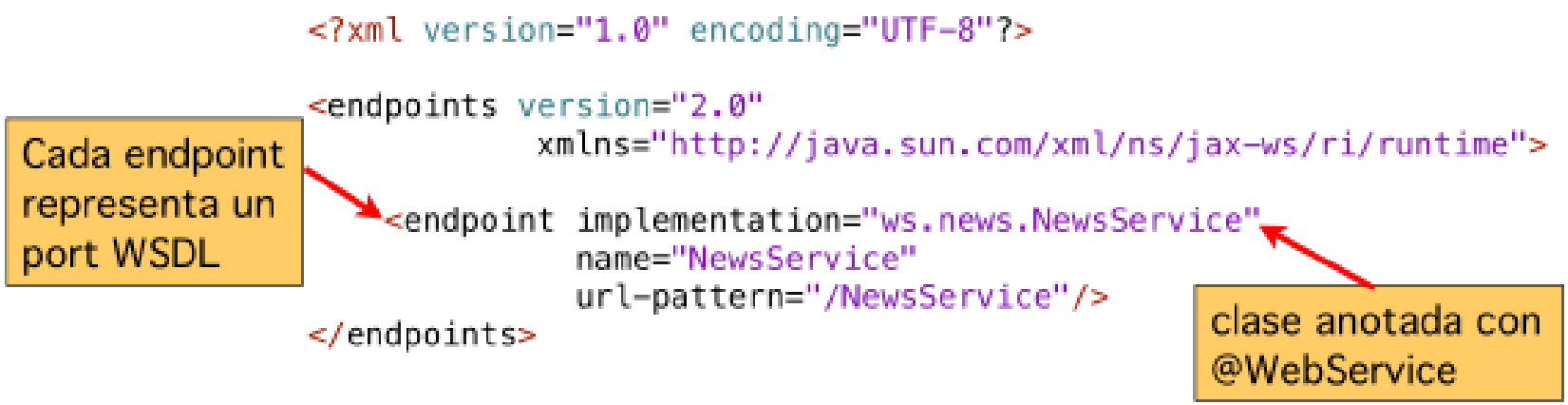
- JAX-WS soporta dos modelos de despliegue:
  - Uno utiliza el fichero `webservices.xml`
  - El otro utiliza los ficheros `web.xml` y `sun-jaxws.xml`
- El fichero `sun-jaxws.xml`:




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

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

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

Despliegue de un WS

- JAX-WS soporta dos modelos de despliegue:
  - Uno utiliza el fichero `webservices.xml`
  - El otro utiliza los ficheros `web.xml` y `sun-jaxws.xml`
- El fichero `sun-jaxws.xml`:

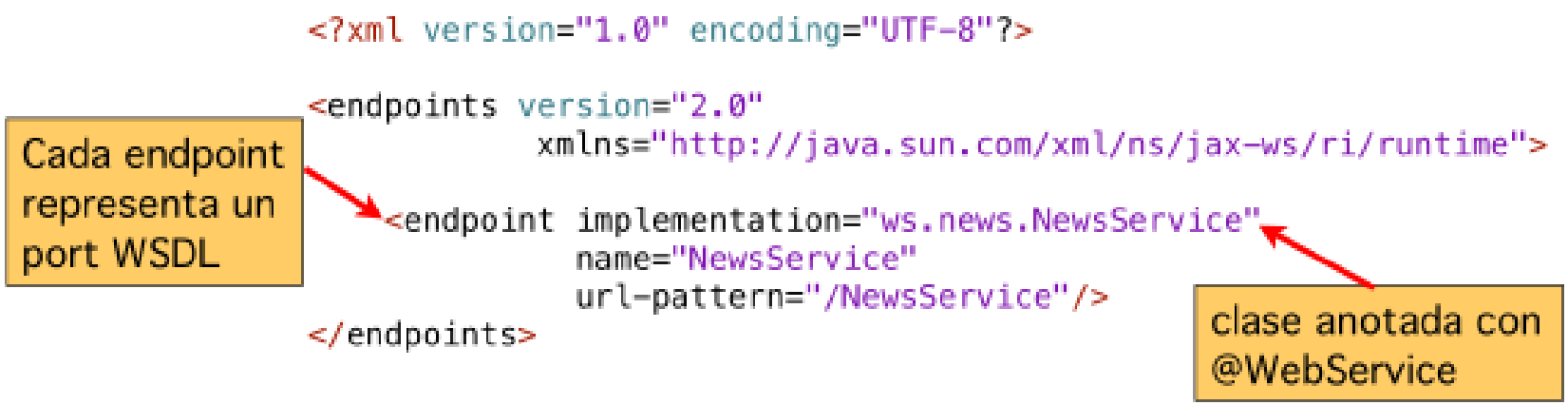



```
<?xml version="1.0" encoding="UTF-8"?>

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

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

  - Contiene la definición de la implementación del *endpoint*.

Despliegue de un WS

- JAX-WS soporta dos modelos de despliegue:
  - Uno utiliza el fichero `webservices.xml`
  - El otro utiliza los ficheros `web.xml` y `sun-jaxws.xml`
- El fichero `sun-jaxws.xml`:




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

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

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

  - Contiene la definición de la implementación del *endpoint*.
  - Cada *endpoint* representa un `port` WSDL contiene la clase que implementa el servicio (`ws.news.NewsService`), url-pattern del servlet, ubicación del WSDL.

Contenido de web.xml

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

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

servlet que sirve las peticiones realizadas al servicio

Responsabilidades del contenedor

- El contenedor en el que resida nuestro servicio web debe proporcionar el runtime de JAX-WS, para soportar peticiones de invocación sobre los componentes port desplegados en dicho contenedor.

Responsabilidades del contenedor

- El contenedor en el que resida nuestro servicio web debe proporcionar el runtime de JAX-WS, para soportar peticiones de invocación sobre los componentes port desplegados en dicho contenedor.
- El runtime se encarga de convertir las llamadas SOAP entrantes en llamadas al API java y viceversa. El contenedor es responsable de:

Responsabilidades del contenedor

- El contenedor en el que resida nuestro servicio web debe proporcionar el runtime de JAX-WS, para soportar peticiones de invocación sobre los componentes port desplegados en dicho contenedor.
- El runtime se encarga de convertir las llamadas SOAP entrantes en llamadas al API java y viceversa. El contenedor es responsable de:
  - "Escuchar" a la espera de peticiones SOA/HTTP

Responsabilidades del contenedor

- El contenedor en el que resida nuestro servicio web debe proporcionar el runtime de JAX-WS, para soportar peticiones de invocación sobre los componentes port desplegados en dicho contenedor.
- El runtime se encarga de convertir las llamadas SOAP entrantes en llamadas al API java y viceversa. El contenedor es responsable de:
  - "Escuchar" a la espera de peticiones SOA/HTTP
  - "Parsear" el mensaje de entrada según el tipo de binding

Responsabilidades del contenedor

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

Responsabilidades del contenedor

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

Responsabilidades del contenedor

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

Responsabilidades del contenedor

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

Contenido I

1. La anotación @WebServiceRef
2. Interfaz de Invocación Dinámica
   - DII con un WSDL
   - Sin un WSDL
3. Creación de Servicios Web
   - Servicios web desde la vista del servidor
   - Implementación del servicio JAX-WS con el modelo de servlets
   - Implementación del servicio Web con el modelo EJB
   - Empaquetado y despliegue de un WS
   - Creación de WS con Netbeans
   - Creación de un WS a partir de un EJB
4. Ejercicio
5. Deber

Creación de WS con Netbeans

- Primero necesitamos un contenedor en el que crear nuestros servicios. Nuevo proyecto -> Java Web -> Aplicación Web. Nombre: ConversionWeb.

Creación de WS con Netbeans

- Primero necesitamos un contenedor en el que crear nuestros servicios. Nuevo proyecto -> Java Web -> Aplicación Web. Nombre: ConversionWeb.
- Nuevo -> Servicio Web. Nombre: ConversionWS, Package -> esfot.soa, Crear WS desde cero. O **Crear WS desde Session Bean**.

Creación de WS con Netbeans

- Primero necesitamos un contenedor en el que crear nuestros servicios. Nuevo proyecto -> Java Web -> Aplicación Web. Nombre: ConversionWeb.
- Nuevo -> Servicio Web. Nombre: ConversionWS, Package -> esfot.soa, Crear WS desde cero. O **Crear WS desde Session Bean**.
- Finalizar. Veremos el código creado de nuestro WS en la clase ConversionWS.java

Creación de WS con Netbeans

- Primero necesitamos un contenedor en el que crear nuestros servicios. Nuevo proyecto -> Java Web -> Aplicación Web. Nombre: ConversionWeb.
- Nuevo -> Servicio Web. Nombre: ConversionWS, Package -> esfot.soa, Crear WS desde cero. O **Crear WS desde Session Bean**.
- Finalizar. Veremos el código creado de nuestro WS en la clase ConversionWS.java
- Añadir una operación a nuestro WS. Clic derecho sobre el servicio creado -> Add Operation.

Creación de WS con Netbeans

- Primero necesitamos un contenedor en el que crear nuestros servicios. Nuevo proyecto -> Java Web -> Aplicación Web. Nombre: ConversionWeb.
- Nuevo -> Servicio Web. Nombre: ConversionWS, Package -> esfot.soa, Crear WS desde cero. O **Crear WS desde Session Bean**.
- Finalizar. Veremos el código creado de nuestro WS en la clase ConversionWS.java
- Añadir una operación a nuestro WS. Clic derecho sobre el servicio creado -> Add Operation.
- Nuestro WS convertirá valores de dolar a euros, entonces añadimos la operación `dolar2euro`. Un parámetro `dolar` de tipo `double` y devuelve un valor tipo `double`.

Creación de WS con Netbeans

- Primero necesitamos un contenedor en el que crear nuestros servicios. Nuevo proyecto -> Java Web -> Aplicación Web. Nombre: ConversionWeb.
- Nuevo -> Servicio Web. Nombre: ConversionWS, Package -> esfot.soa, Crear WS desde cero. O **Crear WS desde Session Bean**.
- Finalizar. Veremos el código creado de nuestro WS en la clase ConversionWS.java
- Añadir una operación a nuestro WS. Clic derecho sobre el servicio creado -> Add Operation.
- Nuestro WS convertirá valores de dolar a euros, entonces añadimos la operación `dolar2euro`. Un parámetro `dolar` de tipo `double` y devuelve un valor tipo `double`.
- Se genera el código esqueleto de nuestra operación, con las anotaciones necesarias.

Operación dolar2euro

```
@WebMethod(operationName = "dolar2euro")
public Double dolar2euro(@WebParam(name = "dolar") double
    dolar) {
     return (dolar * 0.896);
}
```

- Ya tenemos creada nuestra operación y el servicio está listo para ser desplegado. Clic derecho en el proyecto -> Run

Operación dolar2euro

```
@WebMethod(operationName = "dolar2euro")
public Double dolar2euro(@WebParam(name = "dolar") double
    dolar) {
    return (dolar * 0.896);
}
```

- Ya tenemos creada nuestra operación y el servicio está listo para ser desplegado. Clic derecho en el proyecto -> Run
- Vamos al navegador `http://localhost:4848/` y verificamos que nuestro WS esta desplegado.

Operación dolar2euro

```
@WebMethod(operationName = "dolar2euro")
public Double dolar2euro(@WebParam(name = "dolar") double
    dolar) {
    return (dolar * 0.896);
}
```

- Ya tenemos creada nuestra operación y el servicio está listo para ser desplegado. Clic derecho en el proyecto -> Run
- Vamos al navegador `http://localhost:4848/` y verificamos que nuestro WS esta desplegado.
- Podemos ir al `Tester` de nuestro WS desde el navegador o directamente desde Netbeans. Clic derecho en el servicio -> Test Web Service

Operación dolar2euro

```
@WebMethod(operationName = "dolar2euro")
public Double dolar2euro(@WebParam(name = "dolar") double
    dolar) {
    return (dolar * 0.896);
}
```

- Ya tenemos creada nuestra operación y el servicio está listo para ser desplegado. Clic derecho en el proyecto -> Run
- Vamos al navegador `http://localhost:4848/` y verificamos que nuestro WS esta desplegado.
- Podemos ir al `Tester` de nuestro WS desde el navegador o directamente desde Netbeans. Clic derecho en el servicio -> Test Web Service
- Podemos comprobar que nuestra operación funciona correctamente.

Operación dolar2euro

```
@WebMethod(operationName = "dolar2euro")
public Double dolar2euro(@WebParam(name = "dolar") double
    dolar) {
    return (dolar * 0.896);
}
```

- Ya tenemos creada nuestra operación y el servicio está listo para ser desplegado. Clic derecho en el proyecto -> Run
- Vamos al navegador `http://localhost:4848/` y verificamos que nuestro WS esta desplegado.
- Podemos ir al `Tester` de nuestro WS desde el navegador o directamente desde Netbeans. Clic derecho en el servicio -> Test Web Service
- Podemos comprobar que nuestra operación funciona correctamente.
- Tenemos un enlace al WSDL que define nuestro servicio que lo necesitaremos cuando implementemos un cliente.

Operación dolar2euro

```
@WebMethod(operationName = "dolar2euro")
public Double dolar2euro(@WebParam(name = "dolar") double
    dolar) {
    return (dolar * 0.896);
}
```

- Ya tenemos creada nuestra operación y el servicio está listo para ser desplegado. Clic derecho en el proyecto -> Run
- Vamos al navegador `http://localhost:4848/` y verificamos que nuestro WS esta desplegado.
- Podemos ir al `Tester` de nuestro WS desde el navegador o directamente desde Netbeans. Clic derecho en el servicio -> Test Web Service
- Podemos comprobar que nuestra operación funciona correctamente.
- Tenemos un enlace al WSDL que define nuestro servicio que lo necesitaremos cuando implementemos un cliente.
- Tendremos un cuadro de texto para cada operación del WS.

Operación dolar2euro

```
@WebMethod(operationName = "dolar2euro")
public Double dolar2euro(@WebParam(name = "dolar") double
    dolar) {
    return (dolar * 0.896);
}
```

- Ya tenemos creada nuestra operación y el servicio está listo para ser desplegado. Clic derecho en el proyecto -> Run
- Vamos al navegador `http://localhost:4848/` y verificamos que nuestro WS esta desplegado.
- Podemos ir al `Tester` de nuestro WS desde el navegador o directamente desde Netbeans. Clic derecho en el servicio -> Test Web Service
- Podemos comprobar que nuestra operación funciona correctamente.
- Tenemos un enlace al WSDL que define nuestro servicio que lo necesitaremos cuando implementemos un cliente.
- Tendremos un cuadro de texto para cada operación del WS.
- En el resultado de la invocación tenemos también los mensajes SOAP que se utilizan para la invocación.

Contenido I

1. La anotación @WebServiceRef
2. Interfaz de Invocación Dinámica
   - DII con un WSDL
   - Sin un WSDL
3. Creación de Servicios Web
   - Servicios web desde la vista del servidor
   - Implementación del servicio JAX-WS con el modelo de servlets
   - Implementación del servicio Web con el modelo EJB
   - Empaquetado y despliegue de un WS
   - Creación de WS con Netbeans
   - Creación de un WS a partir de un EJB
4. Ejercicio
5. Deber

Creación de un WS a partir de un EJB

- Utilicemos un ejemplo de un EJB de Netbeans. Nuevo proyecto -> Samples -> JavaEE -> WAR-based EJB.

Creación de un WS a partir de un EJB

- Utilicemos un ejemplo de un EJB de Netbeans. Nuevo proyecto -> Samples -> JavaEE -> WAR-based EJB.
- Clic derecho sobre el proyecto -> New -> Web Service -> Create Web Service from Existing Session Bean -> Browse -> Enterprise Beans del proyecto -> HelloBean -> Finalizar.

Creación de un WS a partir de un EJB

- Utilicemos un ejemplo de un EJB de Netbeans. Nuevo proyecto -> Samples -> JavaEE -> WAR-based EJB.
- Clic derecho sobre el proyecto -> New -> Web Service -> Create Web Service from Existing Session Bean -> Browse -> Enterprise Beans del proyecto -> HelloBean -> Finalizar.
- Comentar las operaciones del servicio que den conflictos por ser privadas.

Creación de un WS a partir de un EJB

- Utilicemos un ejemplo de un EJB de Netbeans. Nuevo proyecto -> Samples -> JavaEE -> WAR-based EJB.
- Clic derecho sobre el proyecto -> New -> Web Service -> Create Web Service from Existing Session Bean -> Browse -> Enterprise Beans del proyecto -> HelloBean -> Finalizar.
- Comentar las operaciones del servicio que den conflictos por ser privadas.
- Clic derecho en el proyecto -> Run. Verificar su despliegue en Glassfish.

Creación de un WS a partir de un EJB

- Utilicemos un ejemplo de un EJB de Netbeans. Nuevo proyecto -> Samples -> JavaEE -> WAR-based EJB.
- Clic derecho sobre el proyecto -> New -> Web Service -> Create Web Service from Existing Session Bean -> Browse -> Enterprise Beans del proyecto -> HelloBean -> Finalizar.
- Comentar las operaciones del servicio que den conflictos por ser privadas.
- Clic derecho en el proyecto -> Run. Verificar su despliegue en Glassfish.
- Probar el Web Service.

Contenido I

1. La anotación @WebServiceRef
2. Interfaz de Invocación Dinámica
   - DII con un WSDL
   - Sin un WSDL
3. Creación de Servicios Web
   - Servicios web desde la vista del servidor
   - Implementación del servicio JAX-WS con el modelo de servlets
   - Implementación del servicio Web con el modelo EJB
   - Empaquetado y despliegue de un WS
   - Creación de WS con Netbeans
   - Creación de un WS a partir de un EJB
4. Ejercicio
5. Deber

Ejercicio

- Crear un WS básico JAX-WS “HolaMundo”

Ejercicio

- Crear un WS básico JAX-WS “HolaMundo”
- La clase que implementará el servicio web será un `HolaMundo`.

Ejercicio

- Crear un WS básico JAX-WS “HolaMundo”
- La clase que implementará el servicio web será un `HolaMundo`.
- Tendrá solo una operación `String saluda(nombre)` que nos devolverá un mensaje de saludo incluyendo el nombre proporcionado.

Ejercicio

- Crear un WS básico JAX-WS “HolaMundo”
- La clase que implementará el servicio web será un `HolaMundo`.
- Tendrá solo una operación `String saluda(nombre)` que nos devolverá un mensaje de saludo incluyendo el nombre proporcionado.
- Ejemplo: El parámetro de entrada es `nombre` y le damos un valor `Carlos` la salida será `Hola Carlos, cómo estás?`

Ejercicio

- Crear un WS básico JAX-WS "HolaMundo"
- La clase que implementará el servicio web será un `HolaMundo`.
- Tendrá solo una operación `String saluda(nombre)` que nos devolverá un mensaje de saludo incluyendo el nombre proporcionado.
- Ejemplo: El parámetro de entrada es `nombre` y le damos un valor `Carlos` la salida será `Hola Carlos, cómo estás?`
- El nombre del servicio será `Hola` (calor del atributo `serviceName` de la anotación `@WebService`).

Ejercicio

- Crear un WS básico JAX-WS "HolaMundo"
- La clase que implementará el servicio web será un `HolaMundo`.
- Tendrá solo una operación `String saluda(nombre)` que nos devolverá un mensaje de saludo incluyendo el nombre proporcionado.
- Ejemplo: El parámetro de entrada es `nombre` y le damos un valor `Carlos` la salida será `Hola Carlos, cómo estás?`
- El nombre del servicio será `Hola` (calor del atributo `serviceName` de la anotación `@WebService`).
- Crear un cliente web con Netbeans para probar el WS.

Contenido I

1. La anotación @WebServiceRef
2. Interfaz de Invocación Dinámica
   - DII con un WSDL
   - Sin un WSDL
3. Creación de Servicios Web
   - Servicios web desde la vista del servidor
   - Implementación del servicio JAX-WS con el modelo de servlets
   - Implementación del servicio Web con el modelo EJB
   - Empaquetado y despliegue de un WS
   - Creación de WS con Netbeans
   - Creación de un WS a partir de un EJB
4. Ejercicio
5. Deber

Deber

- Implementar un WS que valide un número de cédula. Nombre: `ValidaCIWS`.

Deber

- Implementar un WS que valide un número de cédula. Nombre: `ValidaCIWS`.
- Debe implementar la operación `boolean validarCI(int ci)`: tomará un número de cédula y nos dirá si es válido (`true`) y no (`false`)

Deber

- Implementar un WS que valide un número de cédula. Nombre: `ValidaCIWS`.
- Debe implementar la operación `boolean validarCI(int ci)`: tomará un número de cédula y nos dirá si es válido (`true`) y no (`false`)
  - Algoritmo ejemplo: `http://telesjimenez.blogspot.com/2011/05/algoritmo-de-verificacion-de-cedula.html`

Deber

- Implementar un WS que valide un número de cédula. Nombre: `ValidaCIWS`.
- Debe implementar la operación `boolean validarCI(int ci)`: tomará un número de cédula y nos dirá si es válido (`true`) y no (`false`)
  - Algoritmo ejemplo: `http://telesjimenez.blogspot.com/2011/05/algoritmo-de-verificacion-de-cedula.html`
- Debe implementar la operación `obtenerProvincia(int ci) throws CIFault` tomará como entrada una cédula y nos dirá a que provincia corresponde. Para esto debe primero verificar la cédula utilizando el método anterior. Si no es correcta lanzará una excepción de tipo `CIFault` (que debe ser creada con anterioridad). La excepción mostrará el mensaje "La cédula no es válida".

Deber

- Implementar un WS que valide un número de cédula. Nombre: `ValidaCIWS`.
- Debe implementar la operación `boolean validarCI(int ci)`: tomará un número de cédula y nos dirá si es válido (`true`) y no (`false`)
  - Algoritmo ejemplo: `http://telesjimenez.blogspot.com/2011/05/algoritmo-de-verificacion-de-cedula.html`
- Debe implementar la operación `obtenerProvincia(int ci) throws CIFault` tomará como entrada una cédula y nos dirá a que provincia corresponde. Para esto debe primero verificar la cédula utilizando el método anterior. Si no es correcta lanzará una excepción de tipo `CIFault` (que debe ser creada con anterioridad). La excepción mostrará el mensaje "La cédula no es válida".
- Implementar un **cliente web** que pruebe el WS. El `Tester` no será capaz de captar correctamente las excepciones por lo que será necesario hacerlo en el cliente.