Servicios Web

Sesión 2: Creación de servicios

Puntos a tratar

- Arquitectura de los Servicios Web RPC
- Tipos de datos compatibles
- Creación del fichero JWS
- Servicios web avanzados
- Implementar servicios mediante JDK 1.6
- Implementar servicios mediante Weblogic
- Handlers de mensajes

Introducción

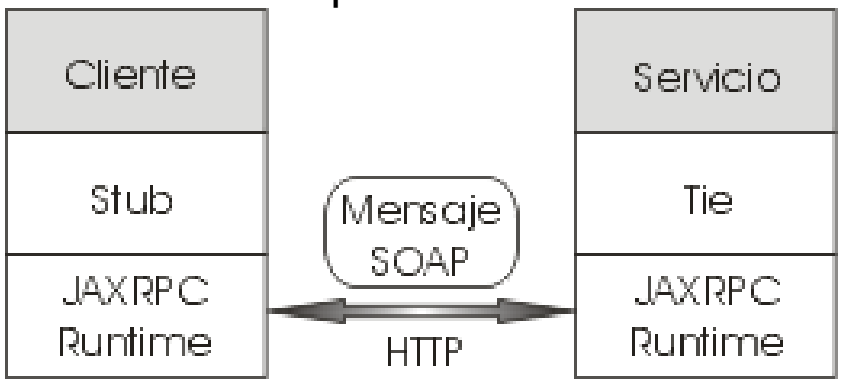
- Los Servicios Web que creemos deberán ofrecer una serie de operaciones que se invocarán mediante SOAP. Por lo tanto:
  - Debe recibir y analizar el mensaje SOAP de petición
  - Ejecutará la operación y obtendrá un resultado
  - Deberá componer un mensaje SOAP de respuesta con este resultado y devolverlo al cliente del servicio
- Si tuviésemos que implementar todo esto nosotros
  - Desarrollar Servicios Web sería muy costoso
  - Se podría fácilmente cometer errores, no respetar al 100% los estándares y perder interoperabilidad

Librerías y herramientas

- Para facilitarnos la tarea contamos con:
  - Librerías (JAX-RPC/WS)
    - Nos permitirá leer y componer mensajes SOAP de forma sencilla
    - Estos mensajes respetarán el estándar
  - Herramientas
    - Generarán de forma automática el código para
      - Leer e interpretar el mensaje SOAP de entrada
      - Invocar la operación correspondiente
      - Componer la respuesta con el resultado obtenido
      - Devolver la respuesta al cliente
- Sólo necesitamos implementar la lógica del servicio
  - La infraestructura necesaria para poderlo invocar mediante SOAP se creará automáticamente

Capas del servicio

- Las capas Stub y Tie
  - Se encargan de componer e interpretar los mensajes SOAP que se intercambian
  - Utilizan la librería JAX-RPC/WS
  - Se generan automáticamente
- El cliente y el servicio
  - No necesitan utilizar JAX-RPC/WS, este trabajo lo hacen las capas anteriores
  - Los escribimos nosotros
  - Para ellos es transparente el método de invocación subyacente
  - El servicio es un componente que implementa la lógica (clase Java)
  - El cliente accede al servicio a través del stub, como si se tratase de un objeto Java local que tiene lo métodos que ofrece el servicio



Tipos de datos básicos

- Tipos de datos básicos y *wrappers* de estos tipos

`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`

Otros tipos de datos y estructuras

- Otros tipos de datos

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

- Colecciones

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

Otras clases

- Podremos utilizar objetos de clases propias, siempre que estas clases cumplan
  - Deben tener un constructor `void` público
  - No deben implementar `javax.rmi.Remote`
  - Todos sus campos deben

    Ser tipos de datos soportados por JAX-RPC/WS

    Los campos públicos no deben ser ni `final` ni `transient`

    Los campos no públicos deben tener sus correspondientes métodos `get`/`set`.
  - Si no cumplen esto deberemos construir serializadores
- También podemos utilizar *arrays* de cualquiera de los tipos de datos anteriores

Fichero JWS

- Forma estándar de definir Servicios Web en Java
  - Clase Java cuyos métodos se ofrecerán como operaciones de un Servicio Web
- Utiliza anotaciones para definir el servicio
  - *Web Services Metadata for the Java Platform* (JSR-181)
- El fichero JWS contendrá:
  - Al menos la anotación `@WebService`
  - Un constructor sin parámetros
  - Por defecto todos sus métodos públicos serán las operaciones del servicio
- Las herramientas utilizadas para generar el servicio dependerán de la plataforma

Ejemplo de fichero JWS

```
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;
  }
}
```

Anotaciones

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

@WebService(name=“MiServicioPortType",
            serviceName=“MiServicio",
            targetNamespace="http://jtech.ua.es")
public class MiServicio {

  @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) { ... }
}
```

Estilo y codificación

- Utilizamos la anotación:

```
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
             use=SOAPBinding.Use.LITERAL,
             parameterStyle=
                    SOAPBinding.ParameterStyle.WRAPPED)
```

- Estilo
  - `SOAPBinding.Style.RPC`
  - `SOAPBinding.Style.DOCUMENT`
- Codificación
  - `SOAPBinding.Use.LITERAL`
  - `SOAPBinding.Use.ENCODED` (*RPC/Encoded*) Desaprobado
- Estilo de los parámetros (para *Document/Literal*)
  - `SOAPBinding.ParameterStyle.BARE`
  - `SOAPBinding.ParameterStyle.WRAPPED`

Tratamiento de errores

- Utilizamos `SOAPFaultExcepcion`
  - Enviará una respuesta *SOAP Fault* al cliente

```
@WebMethod
public double ptas2euro(int ptas) {
  if(ptas<0) {
    lanzarExcepcion("La cantidad de ptas debe ser positiva");
  return ((double) ptas) / 166.386;
}

private void lanzarExcepcion(String mensaje) {
  Detail detail = null;
  try {
    SOAPFactory soapFactory = SOAPFactory.newInstance();
    detail = soapFactory.createDetail();
  } catch (SOAPException e) { }
  QName faultCode = null;
  String faultString = mensaje;
  String faultActor = "Servicio Conversion";
  throw new SOAPFaultException(faultCode, faultString, faultActor, detail);
}
```

Operaciones asíncronas

- Muchas operaciones no devuelven un resultado inmediato
  - P.ej. la aprobación de un crédito necesita ser supervisada por una persona
  - Pueden devolvernos la llamada (*callback*)
- Dos elementos
  - Método que inicia la operación

    Se declara como `@Oneway`
  - Callback que devuelve el resultado
- Deben ser invocados desde servidores

Servicios web conversacionales

- Necesitamos mantener un estado
  - P.ej. para implementar un carrito de la compra
  - Podemos suministrar un ID al cliente
  - Los servicios web conversacionales mantienen un estado para cada cliente de forma transparente
- Tres tipos de operaciones:
  - `START`
  - `CONTINUE`
  - `FINISH`
- Los campos de la clase JWS son privados de cada conversación

Generar el servicio con JDK 1.6

- Contamos con la herramienta wsgen
  - Genera los artefactos necesarios
  - Debemos compilar previamente el fichero JWS

```
wsgen -cp bin -s src -d bin
      es.ua.jtech.servcweb.conversion.ConversionSW
```

- También disponible como tarea de Ant

```
<wsgen classpath="${bin.home}"
       sei="${service.class.name}"
       sourcedestdir="${src.home}"
       destdir="${bin.home}" />
```

Publicar servicios con JDK 1.6

- Podemos publicar sin servidor de aplicaciones

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

Generar el servicio con Weblogic

- Para compilar y crear las capas del servicio usaremos la tarea de *ant* `jwsc`

```
<jwsc srcdir="src"
      destdir="build/ear">
  <jws file=" es/ua/jtech/servcweb/conversion/Conversion.java" />
</jwsc>
```

- En el directorio de salida se generará el EAR que podremos desplegar en Weblogic
  - En el directorio `APP-INF/classes` podremos copiar las clases auxiliares que necesite el servicio

Partir de un documento WSDL

- Nos asegura la máxima compatibilidad con una especificación
- Utilizaremos la tarea `wsdlc`

```
<wsdlc srcWsdl="${wsdl}"
       destJwsDir="${lib.home}"
       destImplDir="${src.home}"
       packageName="${package.name}" />
```

- Esta tarea genera el esqueleto del fichero JWS de nuestro servicio y los tipos de datos necesarios
- Deberemos
  - Rellenar el código del fichero JWS
  - Aplicar `jwsc` para generar el servicio

Despliegue del servicio

- La tarea anterior habrá generado un fichero EAR con el servicio
- Podremos desplegar este servicio utilizando la consola de Weblogic o la tarea `wldeploy`
- Una vez desplegado podremos acceder al WSDL

  `http://localhost:7001/<servicio>/<servicio>?WSDL`

- Podremos probarlo con el cliente de prueba de Weblogic (*Weblogic Test Client*)

  `http://localhost:7001/wls_utc`

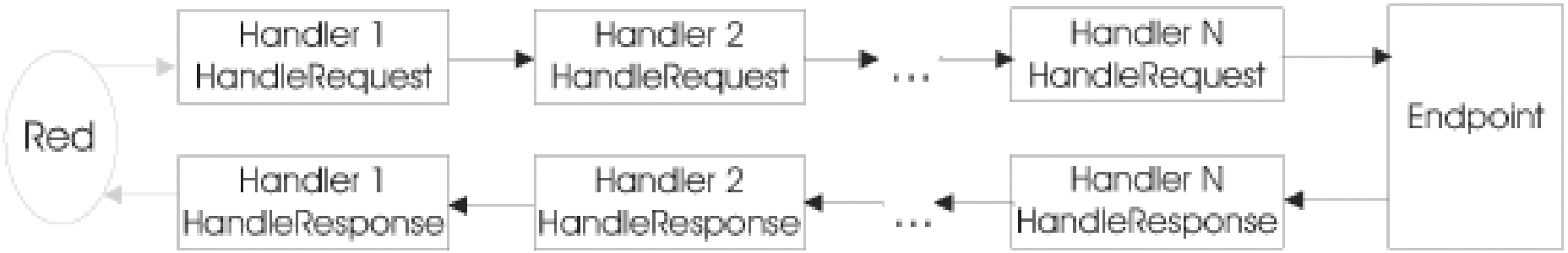
- Deberemos especificar la URL del WSDL

Creación de servicios con Workshop

- En BEA Workshop podemos crear servicios web de forma visual
  - En un proyecto web dinámico o de servicios web
- Puede acceder a otros componentes mediante controles:
  - EJB (local o remoto)
  - JDBC
  - Otros servicios web (remotos)
- Nos permiten exponer funcionalidades de la aplicación en forma de servicios web

¿Qué es un *handler?*

- Similar al concepto de filtro en la API de servlets
  - Intercepta mensajes SOAP de petición y respuesta
- Se pueden instalar en el cliente o en el servidor
  - Sin ellos no podríamos acceder al contenido del mensaje SOAP
- Nos permiten:
  - Encriptar mensajes
  - Restringir acceso
  - Inspeccionar mensajes
  - Registrar mensajes
  - Etc...
- Los handlers se organizan en forma de cadena (`HandlerChain`):



Creación de *handlers*

- Crear una clase que implemente la interfaz `Handler`
- Implementar los métodos:

```
boolean handleRequest(MessageContext context)
boolean handleResponse(MessageContext context)
boolean handleFault(MessageContext context)
init(HandlerInfo info)
destroy()
```

- El valor *booleano* devuelto indica si se debe seguir procesando la cadena
- Podemos acceder al mensaje SOAP interceptado mediante el objecto `MessageContext` proporcionado

```
SOAPMessageContext smc = (SOAPMessageContext)context;
SOAPMessage msg = smc.getMessage();
```

Registro de handlers en el servidor

- Registraremos los *handlers* mediante anotaciones en el fichero JWS
  - Utilizamos `@SOAPMessageHandlers` para definir la cadena de *handlers* que interceptará las llamadas al servicio
  - Utilizamos `@SOAPMessageHandler` para especificar cada *handler* de la cadena
  - De cada *handler* indicaremos la clase Java en la que está implementado
  - Declarar cadena de *handlers*

```
@SOAPMessageHandlers ( {
  @SOAPMessageHandler (
    className="utils.MiHandler"),
  @SOAPMessageHandler (
    className="utils.MiSegundoHandler")
} )
```

Registro de handlers en el cliente

- Podemos registrar *handlers* en el cliente
  - Para interceptar peticiones y respuestas SOAP

- Se registran en la aplicación cliente
  - A través del objeto `Service` que nos da acceso al servicio
  - En caso de utilizar *stub* será el objeto con sufijo `_Impl`

```
HandlerRegistry hr = serv.getHandlerRegistry();
List chain = hr.getHandlerChain(
  new QName("http://jtech.ua.es", "ConversionSoapPort"));
HandlerInfo hi = new
  HandlerInfo(HandlerEspia.class,null,null);
chain.add(hi);
```

¿Preguntas...?