Práctica 1: Arquitectura de JAVA EE (Primera parte)

Índice

1. Objetivos

El objetivo fundamental de esta práctica es adentrarse en la arquitectura de JAVA EE desde el punto de vista del arquitecto de software. Se asume que el alumno ya conoce a través de otras asignaturas el desarrollo de aplicaciones en JAVA EE.

Dadas las dimensiones de JAVA EE como arquitectura de middleware, hemos dividido la práctica en **dos partes** con **entregas separadas**.

En esta **primera parte** se pretenden alcanzar los siguientes **subobjetivos**:

- Experimentar con un sistema multicapa (multitier) de varios niveles: interfaz cliente, aplicación servidora y base de datos. La aplicación servidora la subdividermos en varios niveles según el proyecto adquiera más complejidad.
- Introducir la aplicación de ejemplo que emplearemos a lo largo de todas las prácticas: Aplicación VISA para venta electrónica. Esta aplicación hace uso de JSP, Servlets y JavaBeans.
- Conocer JDBC como API de acceso a base de datos.
- Conocer y experimentar con la tecnología de publicación de Servicios Web o ***Web Services***,
- Automatización de tareas de construcción y despliegue con la herramienta *ant.*

Como se ha comentado anteriormente se asume que el alumno ya conoce y maneja Java EE desde el punto de vista del desarrollador. Si no fuera así, puede consultar los apéndices de esta práctica para ampliar información.

La dedicación estimada para esta práctica es de 4 horas presenciales y 4 horas no presenciales por estudiante.

2. Material entregado

El material entregado en esta práctica se puede descargar de la página de Moodle de la asignatura. Consta de los siguientes archivos:

- *P1-base.tar.gz: aplicación web de ejemplo, en tres capas: páginas JSP, controladores servlets y clases de acceso directo a base de datos.*
- *P1-ws.tar.gz: archivos adicionales necesarios para convertir P1-base en una aplicación con servicio web*.

3. Entorno

El entorno de realización de la práctica es el indicado en la práctica 0. Para realizar las prácticas en el laboratorio se podrá utilizar sólo la partición de Linux. En el ordenador de personal el entorno de debe estár ya instalado y configurado (Práctica 0)

Importante: Si la variable de entorno J2EE_HOME no está definida o no contiene el valor del directorio de instalación de Glassfish (*<install-dir-j2ee>*), es necesario exportarla con el comando: .

```
export J2EE_HOME=<install-dir-j2ee>
```

En los laboratorios, este directorio es:

```
/opt/glassfish4/glassfish
```

En la máquina virtual, este directorio es:

```
/opt/glassfish4.1.2/glassfish
```

4. Introducción a la arquitectura Java EE

El servidor Glassfish integra la ejecución de *servlet*, JSP y EJBs en la misma plataforma. Así mismo, incluye un gestor de colas MQ lo que permite no depender de un proveedor de mensajería externo para poder intercomunicar aplicaciones a través de JMS.

En la **Figura 1**, se puede ver la arquitectura de una instancia del servidor Glassfish. En esta práctica van a entrar en juego los elementos que se muestran en la figura enmarcados en color rojo: los *web clients* (clientes web), que incluye un navegador o cualquier aplicación capaz de comunicarse por HTTP, el *http server* (Servidor web) y el *Web Container* (Contenedores web). El *Http server* es la parte que se encarga de implementar el protocolo HTTP. Ésta es transparente al programador, y si se requiere acceder a las peticiones recibidas se hará a través de métodos de la API de *HttpServlet*. El *Web Container* (Contenedor web), es quien contiene las instancias de los objetos que se encargan de ejecutar cada *servlet/JSP* de nuestra aplicación. El Java Message Service es el componente middleware encargado de intermediar con los proveedores de mensajería, como el gestor de colas incluido en el propio Glassfish, para la intercomunicación de aplicaciones JAVA EE.

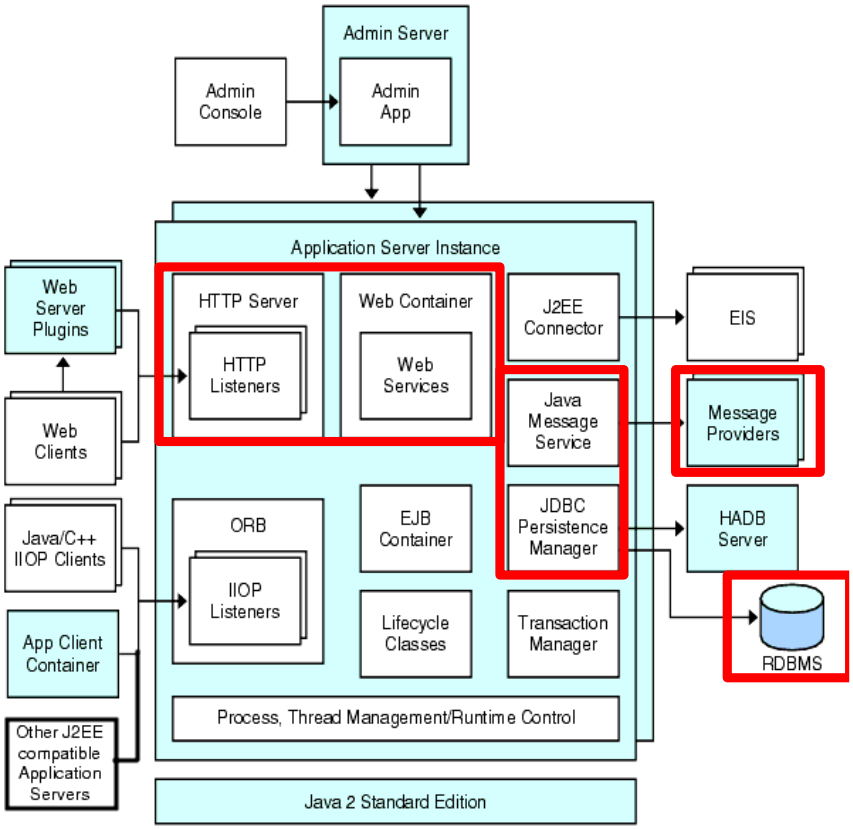


Figura 1. Módulos de la arquitectura de Glassfish relacionados con el contenido de la práctica

En aplicaciones que emplean el paradigma MVC[1], distinguiremos entre los siguientes componentes:

[1] Modelo-Vista-Controlador

- **JSP**: empleados para la **vista** de la aplicación.
- **Servlets**: actúan como **controladores** de la lógica de navegación de la aplicación.
- **JavaBeans / EJBs**: se emplean como representación del **modelo** de la aplicación. Diferenciamos entre modelo de datos / modelo de negocio. Pueden a su vez intermediar con un gestor de bases de datos (DBMS) donde se persiste el modelo de datos.

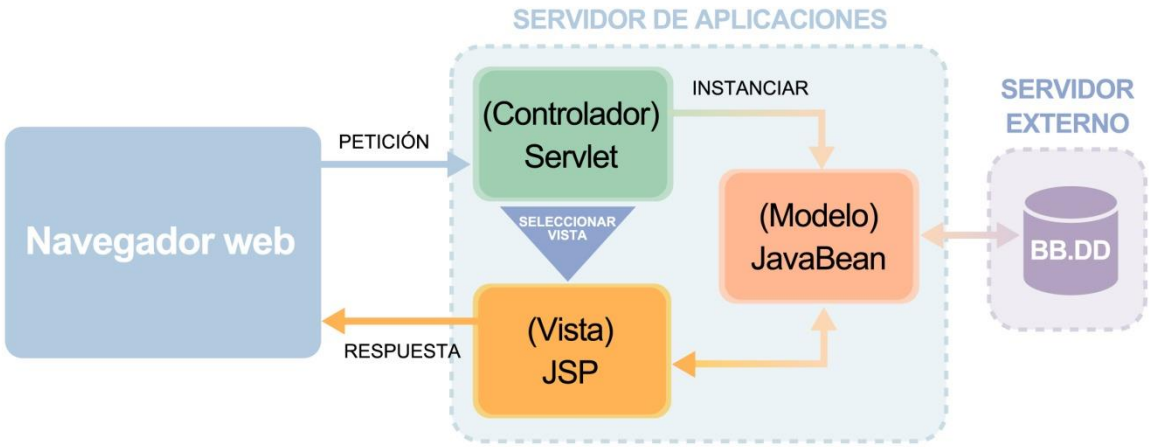


Figura 2. MVC para una aplicación web en Java EE

4.1 VISA: Aplicación de ejemplo

En esta práctica 1 vamos a partir de una aplicación de ejemplo que cubre los tres aspectos de MVC.

Se pretende crear una aplicación que gestione pagos con tarjeta VISA. Ésta se ejecuta como una aplicación independiente de las posibles tiendas online que pudieran querer hacer uso de estos servicios de medios de pago.

La aplicación recibe una petición de pago del comercio, se encarga de mostrar un formulario para rellenar los datos de la tarjeta VISA, realiza el pago tras comprobar la tarjeta, y permite al usuario volver al comercio desde donde se inició el pago. La petición de pago inicial tiene que incluir los siguientes parámetros:

- El **identificador del comercio**: único para cada comercio. Lo tiene que incluir el comercio como parámetro del pago.
- El **identificador de transacción en el comercio**: identificador único por cada pago de este comercio. Lo establece el comercio, el cual deberá garantizar que es único con respecto a otros pagos.
- **Importe**: cantidad de dinero que se va a cargar a la tarjeta.
- **Ruta de retorno**: URL al que se debe volver una vez completado el pago.

Tras la petición de pago, nuestra aplicación genera un formulario que recoge los siguientes datos referentes a la tarjeta:

- Número de tarjeta de crédito.
- Nombre del titular.

- Fecha de emisión (MM/AA). Esta fecha es la fecha desde la que es válida la tarjeta que se usa para realizar el pago.
- Fecha de caducidad (MM/AA).
- Código de verificación (CVV2).

Tras validar los campos de la tarjeta, y comprobar que está autorizada para cargar el pago, se realizará el mismo y se generará un **identificador de autorización** y un **código de respuesta** (correcto, incorrecto) que se devolverán al comercio. **Dicho identificador de autorización será único y global a todos los comercios que nuestro servicio pueda atender**.

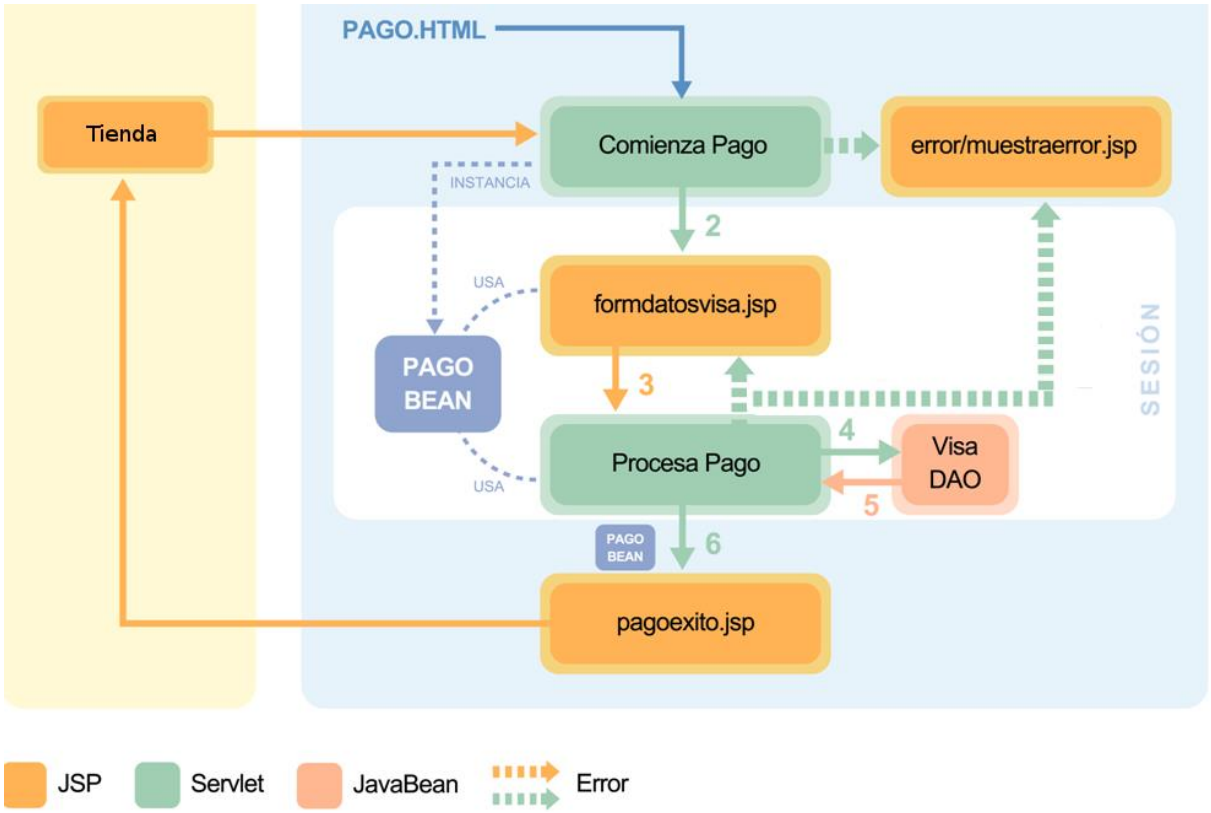


Figura 3. Navegación a través de la aplicación de la práctica 1

Las peticiones de pago son recibidas por el *servlet ComienzaPago*. Estas provienen de los comercios subscritos a nuestro servicio. Además, se ha incluido como parte de nuestra aplicación una página estática pago.html para realizar pruebas. Esta debería ser eliminada si la aplicación se desplegara en un entorno de producción real. *ComienzaPago* valida los parámetros del nuevo pago, crea una nueva sesión, y redirige la petición al formulario *formdatosvisa.jsp*. Este pide al usuario los datos de la tarjeta, y se los envía al *servlet ProcesaPago*. Aquí se validan los campos de la tarjeta. En caso de que hubiera un fallo en alguno, se vuelve a llamar *formdatosvisa.jsp* para que muestre el error correspondiente. Si todos son correctos se instancia *VisaDAO* para:

- Comprobar si la tarjeta está autorizada (método **compruebaTarjeta()**).
- Realizar el pago (método **realizaPago()**). Este método, además, modificará el objeto pago para incluir el número de autorización obtenido de la capa de datos.

Un fallo en alguna de las dos operaciones anteriores redirige a *error/muestraerror.jsp*. Si el pago se ha realizado correctamente, se cierra la sesión, y se redirige a *pagoexito.jsp*. Esta última página muestra un comprobante del pago, y genera un enlace para que el usuario pueda volver al comercio.

4.1.1 Componentes

Páginas JSP/HTML:

- *pago.html*: formulario estático para probar enviar peticiones de pago.
- *cabecera.jsp*: parte superior común a todas las páginas JSP de la aplicación.
- *pie.html*: parte inferior común a todas las páginas JSP de la aplicación.
- *pagoexito.jsp*: página que se muestra si el pago ha concluido con éxito. Genera la URL para poder retornar al comercio.
- *formdatosvisa.jsp*: formulario para recoger los datos de la tarjeta VISA.
- *error/muestraerror.jsp*: página para mostrar los posibles errores que se produzcan.

Código java. Se distinguen los siguientes paquetes:

Paquete *ssii2.controlador*: Agrupa los *servlets* de la aplicación

- *ComienzaPago*: *servlet* que recibe las peticiones de pago través de *pago.html*.
- *ProcesaPago*: recibe un pago y los datos de la tarjeta. Valida los campos de la misma y procesa el pago si la tarjeta está autorizada.
- *ServletRaiz*: clase abstracta que encapsula funcionalidad común a ambos *servlets*.

Paquete *ssii2.filtros*

- *CompruebaSesion*: filtro que previene de accesos incorrectos o malintencionados a la aplicación.

Paquete *ssii2.visa*: agrupa aquellas clases relacionadas con los datos de la aplicación.

- *PagoBean*: *bean* para almacenar los parámetros de un pago.
- *TarjetaBean*: *bean* para almacenar los parámetros de una tarjeta.
- *ValidadorTarjeta*: clase que se encarga de validar los campos de una tarjeta.

Paquete *visa.dao*: incluye las clases que permiten acceder al modelo de datos de VISA. DAO corresponde a las siglas “Data Access Object”, objeto de acceso a datos.

- *VisaDAO*: *bean* que permite acceder al servicio de pago de visa. En esta práctica se provee un *bean* muy básico para realizar pruebas. En sucesivas prácticas se irá modificando según las necesidades.

Paquete *visa.error*: engloba los posibles errores que se puedan producir durante la validación de la tarjeta.

Archivos de configuración:

- *web.xml*: archivo de configuración de la aplicación (véase sección 15).

El flujo de la aplicación completa se puede visualizar en la Figura 3.

4.1.2 Modelo de datos

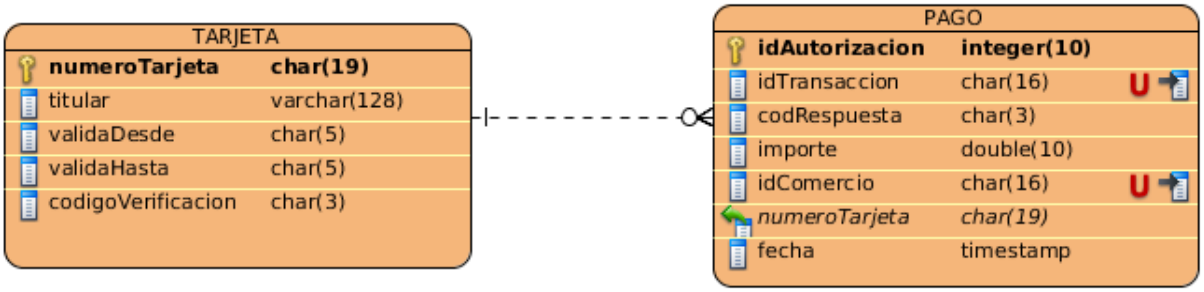


Figura 4: Diagrama E-R de base de datos de VISA

- **TARJETA**: listado con las tarjetas válidas para realizar un pago.
- **PAGO**: registro de todos los pagos realizados desde nuestra tienda.

4.2 Compilación y despliegue

Los siguientes ficheros adicionales son importantes para la compilación y despliegue de esta aplicación:

- **build.xml**: Fichero *ant* para automatizar la regeneración de la aplicación. Algunos de los *targets* que emplearemos son:
  - ayuda: muestra ayuda de las opciones disponibles.
  - compilar: se encarga de la compilación (invocación de javac) de las clases.
  - empaquetar: se encarga de la preparación del paquete con la aplicación web (fichero .war).
  - desplegar: se encarga del despliegue (instalación) de la aplicación contra el servidor de aplicaciones.
  - replegar: se encarga del repliegue (desinstalación) de la aplicación web.
  - regenerar-bd: se encarga de la recreación de la base de datos.
  - limpiar: se encarga de la limpieza del proyecto.
  - todo: se encarga de la limpieza, compilación, recreación de bases de datos y despliegue de la aplicación.
- **build.properties**: fichero de propiedades donde se parametrizan variantes de despliegue (dirección IP remota, usuarios de servidor de aplicaciones, rutas locales con paquetes a desplegar y ficheros de contraseñas).
- **postgresql.xml**: fichero *ant* para automatizar la reconstrucción de la parte de bases de datos del proyecto. Esto incluye la creación de la base de datos en sí, así como recursos JDBC asociados y *pools* de conexiones. Este fichero no debe ser utilizado directamente, lo incluirá build.xml para importar sus *targets*. Este fichero podría ser reemplazado por uno similar en caso de querer trabajar con otro DBMS. Sus dos *targets* principales son:
  - setup-db: se encarga de la reconstrucción de la base de datos.
  - unsetup-db: se encarga de la eliminación de la base de datos.

- **postgresql.properties**: fichero de propiedades específicos de la parte de base de datos. Se incluye aquí, usuarios DBMS, nombres de bases de datos, recursos JDBC, pools, etc.

4.3 Configuración del despliegue

Para que la aplicación se despliegue de forma correcta es necesario realizar la configuración de los servidores en donde lo hará. Por ejemplo, es necesario especificar la dirección IP del servidor de aplicaciones donde se desplegará la aplicación. Para ello se ha de fijar la dirección IP de dicho servidor en la variable **as.host** del fichero build.properties. Del mismo modo es necesario especificar en qué host se encontrará el servidor postgresql que contendrá la base de datos de la aplicación. Para ello habrá que fijar la dirección IP de dicho host en la variable **db.host** del fichero postgresql.properties. Finalmente, la aplicación puede acceder a la base de datos a través de un objeto DataSource (este es un objeto que simplifica el acceso a recursos de almacenamiento) y un pool de conexiones (que permite reutilizar conexiones a la base de datos, pues crear y destruir estas conexiones es relativamente costoso). Para ello, será necesario configurar el servidor de aplicaciones donde se van a crear estos recursos, que será el host donde se encuentre la aplicación que los utilizará, que actuará como cliente de la base de datos. Esto se hará indicando la dirección IP de dicho servidor de aplicaciones en la variable **db.client.host** del fichero postgresql.properties.

Nota importante: la dirección IP que se debe utilizar en los ficheros .properties para indicar el host sobre el que realizar el despliegue de la aplicación (variable as.host), el *DataSource* (db.client.host) y el servidor donde se encontrará la base de datos (db.host), será aquella **asignada a cada grupo de prácticas** para la máquina virtual. Es decir, todos los servidores (aplicaciones y postgresql se encontrarán en la misma máquina virtual).

Nota importante: se entrega un fichero passwordfile al que se hace referencia en el fichero build.properties. Si se desea cambiar la contraseña de glassfish para evitar accesos involuntarios por otras parejas se puede usar el mandato:

```
asadmin change-admin-password –user admin
```

y alterar dicho fichero passwordfile en consonancia.

Ejercicio 1. Prepare e inicie una máquina virtual a partir de la plantilla **si2srv** con: 1GB de RAM asignada, 2 CPUs. A continuación:

- Modifique los ficheros que considere necesarios en el proyecto para que se despliegue tanto la aplicación web como la base de datos contra la dirección asignada a la pareja de prácticas.
- Realice un pago contra la aplicación web empleando el navegador en la ruta http://10.X.Y.Z:8080/P1

  Conéctese a la base de datos (usando el cliente Tora por ejemplo) y obtenga evidencias de que el pago se ha realizado.
- Acceda a la página de pruebas extendida, http://10.X.Y.Z:8080/P1/testbd.jsp. Compruebe que la funcionalidad de listado de y borrado de pagos funciona correctamente. Elimine el pago anterior.

Incluya en la memoria de prácticas todos los pasos necesarios para resolver este ejercicio así como las evidencias obtenidas. Se pueden incluir por ejemplo capturas de pantalla.

5. Conexión mediante JDBC

Con Java podemos utilizar varios mecanismos para conectarnos a una base de datos. Los más sencillos son la conexión directa (mediante instanciación del *driver* JDBC) o a través de un *DataSource* identificado por su nombre (*Data Source Name o DSN*). El uso de JPA (api de persistencia de java) u otros mecanismos

para abstraernos de la base de datos están siempre montados sobre JDBC, aunque no se estudiarán en estas prácticas.

5.1 Conexión directa

Este método se caracteriza por su sencillez, aunque tiene el problema de ser poco genérico (se requiere conocer el *driver* para conectar y la cadena de conexión que incluye host y base de datos) y además tiene problemas de rendimiento al no reutilizar las conexiones (dado que la apertura es una operación costosa). En la práctica no se suele utilizar este método. Consulte los apéndices para más información sobre cómo emplearlo.

5.2 Conexión mediante DataSource

Un *DataSource* u "Origen de datos" es un recurso catalogado en el servidor de aplicaciones.

Éste será creado de forma previa a la ejecución de la aplicación, y viene caracterizado por su nombre JNDI (*Java Naming and Directory Interface*), normalmente de la forma: "*jdbc/<RecursoBD>*".

En nuestra implementación, este paso de registro se realizó en el despliegue con *postgresql.xml* / *postgresql.properties* aunque también puede realizarse a través de la Consola de Administración.

Aunque este mecanismo requiere preparación adicional (su catalogación en la fase de despliegue), tiene la ventaja de ser más genérico (la dependencia con el gestor DBMS queda acotada en la definición del recurso) así como de permitir el trabajo con *pools* de conexiones, donde las conexiones con la base de datos son reutilizadas por las aplicaciones.

Consulte los apéndices o el código de **VisaDAO.java** para mayores detalles sobre cómo emplearlo desde la aplicación.

5.3 Pool de conexiones

El coste de abrir una conexión a la base de datos normalmente no es despreciable, en especial tratándose de una conexión a un DBMS remoto.

Un *pool* de conexiones es un conjunto de objetos *Connection* que el servidor de aplicaciones mantiene asociados a un recurso JDBC (*DataSource*). El número de conexiones oscila entre dos valores mínimo y máximo, de modo que toda aplicación que solicite una conexión al *DataSource* recibirá una conexión **preabierta** y **reutilizable**. Una vez finalice el trabajo con ella, tras realizar el *close*(), ésta será devuelta al *pool*.

La principal ventaja de este mecanismo es, por tanto, que el coste de abrir la conexión pasa a ser mucho menor, consiguiendo mejoras importantes en el rendimiento de acceso a datos para nuestra aplicación.

Al desplegar la base de datos en los pasos anteriores, se habrán creado un *pool* y un recurso *jdbc*, que pueden ser examinados en la consola de administración de Glassfish (http://10.X.Y.Z:4848) en el apartado Resources → JDBC.

5.4 Cierre de conexión

En cualquiera de los dos casos anteriores (conexión directa o *datasource*), la conexión se cierra mediante el método *close()* de la clase *Connection*. No obstante, el cierre en el caso de un pool de conexiones no es tal, ya que lo que se hace es *devolver la conexión al pool* para que pueda ser reutilizada.

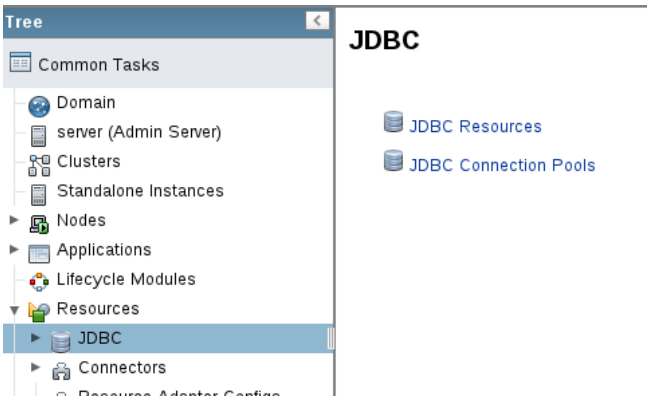


Figura 5: Recursos JDBC en el servidor de aplicaciones

Ejercicio 2. La clase **VisaDAO** implementa los dos tipos de conexión descritos anteriormente, los cuales son heredados de la clase DBTester. Sin embargo, la configuración de la conexión utilizando la conexión directa es incorrecta. Se pide completar la información necesaria para llevar a cabo la conexión directa de forma correcta. Para ello habrá que fijar los atributos a los valores correctos. En particular, el nombre del driver JDBC a utilizar, el *JDBC connection string* que se debe corresponder con el servidor posgresql, y el nombre de usuario y la contraseña. Es necesario consultar el apéndice 10 para ver los detalles de cómo se obtiene una conexión de forma correcta. Una vez completada la información, acceda a la página de pruebas extendida, http://10.X.Y.Z:8080/P1/testbd.jsp y pruebe a realizar un pago utilizando la conexión directa y pruebe a listarlo y eliminarlo. Adjunte en la memoria evidencias de este proceso, incluyendo capturas de pantalla

Ejercicio 3. Examinar el archivo *postgresql.properties* para determinar el nombre del recurso JDBC correspondiente al *DataSource* y el nombre del *pool*. Acceda a la *Consola de Administración*. Compruebe que los recursos JDBC y pool de conexiones han sido correctamente creados. Realice un *Ping* JDBC a la base de datos. Anote en la memoria de la práctica los valores para los parámetros ***Initial and Minimum Pool Size, Maximum Pool Size, Pool Resize Quantity, Idle Timeout, Max Wait Time***. Comente razonadamente qué impacto considera que pueden tener estos parámetros en el rendimiento de la aplicación.

Ejercicio 4. Localice los siguientes fragmentos de código SQL dentro del proyecto proporcionado (P1-base) correspondientes a los siguientes procedimientos:

- Consulta de si una tarjeta es válida.
- Ejecución del pago.

Incluya en la memoria de prácticas dichas consultas.

6. El log del servidor de aplicaciones

6.1 Consulta del log del servidor de aplicaciones

Generalmente es difícil determinar las razones de error de una aplicación Java EE que se ejecuta en el servidor de aplicaciones Glassfish. La razón es que cuando se produce un error lo único que percibe el cliente en su navegador son mensajes de error del protocolo HTTP, que no son muy informativos sobre la verdadera razón del error. Para determinar la razón del error, algo que puede resultar muy útil es consultar

el log del servidor de aplicaciones. Esto se puede llevar a cabo a través de la consola de administración pulsando en *server* y en la opción *View Log Files*, como se ilustra en la Figura 6. También es posible consultar el log del servidor de aplicaciones desde la máquina virtual, editando el fichero:

```
/opt/glassfish4.1.2/glassfish/domains/domain1/logs/server.log
```

Esto se puede llevar a cabo mediante el editor vi, por ejemplo.

Figura 6 Visualización del log del servidor de aplicaciones

6.2 Escritura en el log del servidor de aplicaciones

Para detectar posibles fallos en las aplicaciones J2EE algo que resulta muy útil es el introducir trazas en el log del servidor de aplicaciones Glassfish. Es decir, forzar a que la aplicación introduzca un mensaje en dicho log que nos pueda indicar sobre qué acciones han sido completadas hasta el momento del error. Para ello se puede utilizar el método *System.err.println(String cadena)*, que escribe en el log el string pasado como argumento. En general, algunas de las clases facilitadas en la aplicación también tienen métodos propios para escribir en log en determinadas circunstancias. Por ejemplo, la clase VisaDAO contiene el método errorLog(String cadena), que escribe en el log del servidor de aplicaciones el string pasado como argumento sí y solo sí la opción debug ha sido habilitada.

En el caso de los servlets, estos también pueden escribir en el log el servidor de aplicaciones utilizando el contexto. Se puede invocar el método HTTPServlet.getContext() para obtener el contexto del servlet. Esto es un objeto de tipo ServletContext. Tras ello, se puede usar el método ServletContext.log(String mensaje), para escribir en el log la cadena mensaje. Ver el apéndice 15.1 para más detalles.

Ejercicio 5:

- Edite el fichero VisaDAO.java y localice el método errorLog. Compruebe en qué partes del código se escribe en log utilizando dicho método. Realice un pago utilizando la página testbd.jsp con la opción de debug activada. Visualice el log del servidor de aplicaciones y compruebe que dicho log contiene información adicional sobre las acciones llevadas a cabo en VisaDAO.java.

  Incluya en la memoria una captura de pantalla del log del servidor.

7. SOAP Web Services

Los Web Services son un mecanismo de RPC que actualmente representan la base del diseño de las actuales Arquitecturas Orientadas a Servicios (SOA, *Service Oriented Architecture*) utilizando ***SOAP*** (*Simple Object Access Protocol*, Protocolo Simple de Acceso a Objetos) y ***WSDL*** (*Web Services Description Language*, Lenguaje de Descripción de Servicios Web). El objetivo fundamental de los Web services es

facilitar la interoperabilidad en la publicación y uso de servicios distribuidos. Así, distintas aplicaciones de software desarrolladas en lenguajes de programación diferentes, y ejecutadas sobre cualquier plataforma, pueden utilizar los servicios web para intercambiar datos en redes como Internet.

Su implementación en el servidor de aplicaciones GlassFish se realiza a través de clases Servlet que se autogeneran a partir del descriptor del servicio WSDL.

7.1 Nomenclatura

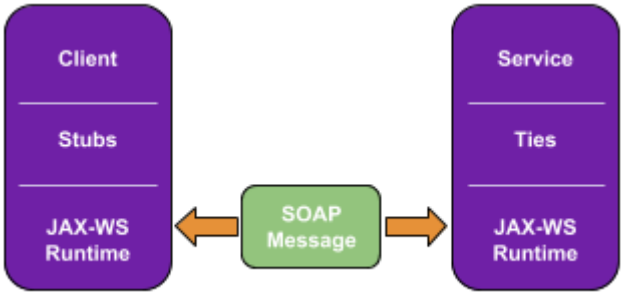


Figura 7. Esquema de capas en la intercomunicación de un cliente y su servicio

- **Cliente**: clase que invoca al servicio remoto como si de uno local se tratase
- **Stubs**: clases que intermedian entre el cliente y el *runtime* SOA (en nuestro caso, JAX-WS). Estas clases son **autogeneradas** (mediante la herramienta *wsimport*) a partir de la definición del servicio.
- **Mensaje SOAP**: mensaje en XML transmitido a través de la red, incluyendo todo lo necesario para invocar el servicio (operación remota – método, parámetros, etc). Normalmente no nos tenemos que preocupar por este mensaje ya que es generado por el *runtime* de JAX-WS
- **Ties**: clases que intermedian entre el *runtime* de JAX-WS y la clase que implementa el servicio en el lado del servidor. Ofrecen a la clase servidora la impresión de estar siendo llamada como un método local.
- **Servicio**: clase que implementa la funcionalidad en el lado del servidor

7.2 VISA: Un servicio web de compra electrónica

En esta sección de la práctica se va a implementar un servicio web para implementar la funcionalidad del pago VISA que se realizó en el apartado anterior, para aportar una interfaz SOAP que permita acceder directamente al servicio desde aplicaciones externas.

Esto nos permitirá desacoplar la pasarela de pagos en dos niveles:

- **Interfaz de usuario** (*servlets* y JSP). La pasarela de pago ofrece una serie de pantallas por defecto para realizar el pago. Los comercios que quieran optar a presentar su propia interfaz de usuario para realizar los pagos, pueden omitir estas pantallas y conectar directamente con el servicio.
- **Servicio de pago**. El servicio web, que proporciona dos métodos para consultar si una tarjeta es válida y otro para realizar el pago propiamente dicho.

El servicio de pago continuará apoyándose en la base de datos **PostgreSQL**, donde consultará el listado de tarjetas válidas y posteriormente almacenará la información de los pagos realizados. El objetivo es convertir el *bean* VisaDAO (que encapsula el acceso a datos) en el servicio web.

Al convertir VisaDAO en un servicio, introducimos una nueva capa en nuestra aplicación JAVA EE que nos permite distribuir la aplicación en más servidores, aumentando la escalabilidad.

A partir de ahora distinguiremos:

- **Paquete "cliente":** el cliente del servicio. Son los *servlets* de la aplicación web que en adelante están desacoplados del acceso a datos y realizan su interacción se hace a través del nuevo servicio web.
- **Paquete "servidor":** implementación del servicio de pago como servicio web.

VisaDAO cliente y servidor **no se empaquetarán conjuntamente**. Cada uno tendrá su empaquetado independiente, y se desplegarán en distintos servidores, realizándose el acceso del cliente al servicio web a través de la red de área local.

Dado que en esta parte de la práctica es necesario realizar despliegues antes de finalizar la compilación de todos los elementos, exponemos aquí la configuración sobre la que se desplegará, que seguirá la estructura de la siguiente figura:

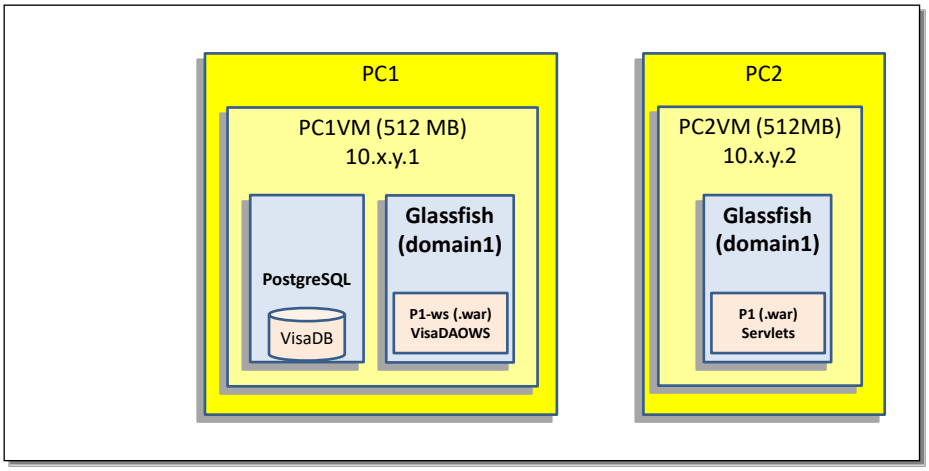


Figura 8. Diagrama de despliegue de la práctica con servicio web[2]

Los dos servidores de aplicaciones desplegados en PC1VM y PC2VM son independientes. No hay conflicto porque en ambos el dominio se llame domain1, pero hay que tener claro que son dos dominios diferentes.

Nota importante: de nuevo habrá que configurar en los ficheros .properties la dirección IP que se debe utilizar para realizar el despliegue de las aplicaciones cliente y servidor, el *DataSource* y el servidor donde se encontrará la base de datos. En este caso habrá que considerar dos direcciones IP. La de PC1VM y la de PC2VM. La configuración se llevará a cabo siguiendo el esquema de la Figura 8.

7.3 Material de partida

El alumno dispone del archivo P1-ws.tgz, que descomprimirá para conseguir la siguiente estructura de archivos:

P1-ws	
|-- build.xml	Archivo ant para desplegar
|-- web	

[2] Errata: El servicio se despliega con P1-ws-ws.war y el cliente con P1-ws-cliente.war

|-- (vacío)	Completar con el mismo directorio de la parte 1
|-- sql	
|-- (vacío)	Completar con el mismo directorio de la parte 1
|-- datagen	
|-- (vacío)	Completar con el mismo directorio de la parte 1
|-- conf	
|-- serverws	Configuración servidor para WebServices
|-- META-INF	
|-- MANIFEST.MF	Archivo MANIFEST para WebServices
|-- src	
|-- client	Fuentes del cliente
|-- (vacío)	Completar según instrucciones que siguen
|-- server	Fuentes del servidor
|-- (vacío)	Completar según instrucciones que siguen

7.4 Combinar ambos proyectos

A continuación, el alumno preparará un nuevo proyecto combinando el contenido de P1-base y P1-ws. Para ello deberán seguirse los siguientes pasos:

- Tomar la carpeta P1-ws como punto de partida. De esta nos quedaremos con el nuevo build.xml que incluye las nuevas reglas de despliegue del servicio web, así como la nueva organización en carpetas.
- Copiar de P1-base => P1-ws los directorios: datagen, sql, web.
- Copiar de P1-base => P1-ws los ficheros: postgresql.properties, postgresql.xml.
- Reorganizar la carpeta src del proyecto P1-ws en su nueva ubicación dentro de P1-ws conforme a la siguiente tabla. Nótese que:
    - VisaDAO.java deberá ser renombrado como VisaDAOWS.java en su nueva ubicación, así como la clase Java en él contenida.

P1-base: Carpeta src/	**P1-ws: Carpeta src/**
src/ssii2/visa/error/*	src/client/ssii2/error/*
src/ssii2/visa/ValidadorTarjeta.java	src/client/ssii2/visa/ValidadorTarjeta.java
src/ssii2/filtros/CompruebaSesion.java	src/client/ssii2/filtros/CompruebaSesion.java
src/ssii2/controlador/ComienzaPago.java	src/client/ssii2/controlador/ComienzaPago.java
src/ssii2/controlador/GetPagos.java	src/client/ssii2/controlador/GetPagos.java
src/ssii2/controlador/DelPagos.java	src/client/ssii2/controlador/DelPagos.java
src/ssii2/controlador/ServletRaiz.java	src/client/ssii2/controlador/ServletRaiz.java
src/ssii2/controlador/ProcesaPago.java	src/client/ssii2/controlador/ProcesaPago.java
src/ssii2/visa/dao/DBTester.java	src/server/ssii2/visa/DBTester.java
src/ssii2/visa/dao/VisaDAO.java	**src/server/ssii2/visa/VisaDAOWS.java**

src/ssii2/visa/PagoBean.java src/ssii2/visa/TarjetaBean.java	src/server/ssii2/visa/PagoBean.java src/server/ssii2/visa/TarjetaBean.java

También será necesario copiar, si no ha sido copiado ya, el .jar con el driver para la base de datos postgresql, "postgresql-jdbc-4.jar", que se encontraba en la práctica P1-base al directorio P1-ws. Esto es debido a que dicho driver es necesario para crear el pool de conexiones y el objeto DataSource en el servidor de aplicaciones glassfish. Si no se realiza la copia se producirá error al intentar llevar a cabo el objetivo setup-db.

7.5 Conversión de la clase VisaDAO en un Web Service (VisaDAOWS)

En este apartado vamos a modificar la clase Java VisaDAOWS.java para que cumpla con los requisitos necesarios para convertirse en un servicio web.

Para ello emplearemos las siguientes anotaciones:
`@WebService():` `indica que la clase Java implementa un servicio web.`
`@WebMethod(operationName = "nombreMetodo"):` `indica que el método Java que le sigue será exportado como un método público del servicio.`
`@WebParam(name = "nombreArgumento"):` `indica que el argumento que viene a continuación también es un argumento del método equivalente del servicio.`

Los siguientes *import* Java deben ser incluidos para disponer de dichas anotaciones:

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

Ejercicio 6. Realícense las modificaciones necesarias en VisaDAOWS.java para que implemente de manera correcta un servicio web. Los siguientes métodos y todos sus parámetros deberán ser publicados como métodos del servicio.

- compruebaTarjeta()
- realizaPago()
- isDebug() / setDebug() (**Nota**: VisaDAO.java contiene dos métodos setDebug que reciben distintos argumentos. Solo uno de ellos podrá ser exportado como servicio web)[3].

[3] También se da la circunstancia de que todos los métodos public se exportan como servicio web, independientemente de que tengan o no la anotación @WebMethod. Esto hace que dé error al desplegar el servicio. Para conseguir el despliegue de forma correcta las alternativas son:

1. Evitar la publicación de manera explícita con @WebMethod(exclude=true)

- isPrepared() / setPrepared()

De la clase DBTester, de la que hereda VisaDAOWS.java, deberemos publicar así mismo:

- isDirectConnection() / setDirectConnection()

Para ello, implemente estos métodos también en la clase hija. Es decir, haga un override de Java, implementando estos métodos en VisaDAOWS mediante invocaciones a la clase padre (super). En ningún caso se debe añadir ni modificar nada de la clase DBTester.

Modifique así mismo el método **realizaPago()** para que éste devuelva el pago modificado tras la correcta o incorrecta realización del pago:

- Con identificador de autorización y código de respuesta correcto en caso de haberse realizado.
- Con null en caso de no haberse podido realizar.

Incluye en la memoria cada fragmento de código donde se han ido añadiendo las modificaciones requeridas.

Por último, conteste a la siguiente pregunta:

- ¿Por qué se ha de alterar el parámetro de retorno del método realizaPago() para que devuelva el pago el lugar de un boolean?

7.6 Nuevas reglas de compilación en el servidor

En el nuevo fichero P1-ws/*build.xml* se han añadido un conjunto de objetivos que se irán ejecutando para obtener el servicio web y un cliente (web) que acceda a él:

1. compilar-servicio
2. empaquetar-servicio
3. desplegar-servicio

7.6.1 Compilar el servidor

El primer paso para obtener el servicio web es compilar las clases relativas al servicio web, lo que se conoce como *Service Endpoint* o Punto de Acceso al Servicio. Para la compilación de las clases del servicio web VISA, que deben encontrarse en el directorio *src/server*, se ejecutará el programa *ant* con el objetivo *compilar-servidor*:

```
ant compilar-servicio
```

7.6.2 Empaquetar el servicio

El siguiente paso consiste en empaquetar los archivos del servicio en un archivo .war. Para esto se ejecutará nuevamente la herramienta ant con el objetivo empaquetar:

```
ant empaquetar-servicio
```

Esto generará un archivo dentro del directorio **dist/servidor** llamado **P1-ws-ws.war**. Este archivo .war no contiene las clases "***tie***" o ataduras con la API de JAX-WS. El siguiente apartado tratará este tema. Las clases se crearon en el primer paso, explicado previamente. **También hay que incluir el .jar con el driver**

---

2. Publicar con otro nombre: @WebMethod(operationName = "setDebugS")
   @RequestWrapper(className="server.ssii2.visa.jaxws.setDebugS")
   @ResponseWrapper(className="server.ssii2.visa.jaxws.setDebugSResponse")
3. Borrar o renombrar el método que no se usa.

de conexión para acceder a la base de datos, si fuera necesario. No se incluye archivo *web.xml* porque éste, al igual que las clases *tie* serán generadas de forma automática por el servidor de aplicaciones para el servicio web.

7.6.3 Desplegar del servicio

Finalmente, la tarea de despliegue del servicio se puede llevar a cabo indicándole a *ant* el objetivo *desplegar-servidor*:

```
ant desplegar-servicio
```

Al realizar el despliegue de este archivo, el servidor de aplicaciones detecta que se trata de un servicio web, por lo que realiza las llamadas necesarias para generar las clases *tie*, así como el archivo *web.xml*. También es posible generar estas clases de forma manual usando la herramienta **wsgen**. Esto puede ser útil para depurar si las anotaciones se han escrito de forma incorrecta. No obstante, no es necesario si todo funciona correctamente, pues se genera de forma automática por parte del servidor de aplicaciones.

Para verificar que el servicio se ha desplegado correctamente, se puede acceder al gestor del servidor de aplicaciones. El navegador debería mostrar una página como la de la siguiente figura.

En el árbol de la izquierda aparece la aplicación P1-ws-ws, y a la derecha, el Servicio Web listado como *VisaDAOWS*. Si se pincha en el enlace *'View Endpoint'*, aparecerá la página asociada a la gestión de este servicio web, incluyendo el URL del archivo WSDL del servicio, o el archivo *web.xml* generado automáticamente. Dicha página también incluye un botón de *Test* (probar)[4], que permite enviar parámetros a las operaciones declaradas por el servicio web. Dichas operaciones deben tener como parámetros tipos simples, pues de lo contrario no es posible enviar los valores.

[4] Al intentar usar el Tester os puede aparecer el error: *"schema_reference: Failed to read schema document 'xjc.xsd', because 'file' access is not allowed due to restriction set by the accessExternalSchema property."*

Para corregir este problema podéis editar el fichero:
"/opt/glassfish4.1.2/glassfish/domains/domain1/config/domain.xml" y añadir una línea de tipo:

```
<java-config>
  ...
  <jvm-options>-Djavax.xml.accessExternalSchema=all</jvm-options>
</java-config>
```

Después reiniciar el dominio y listo.

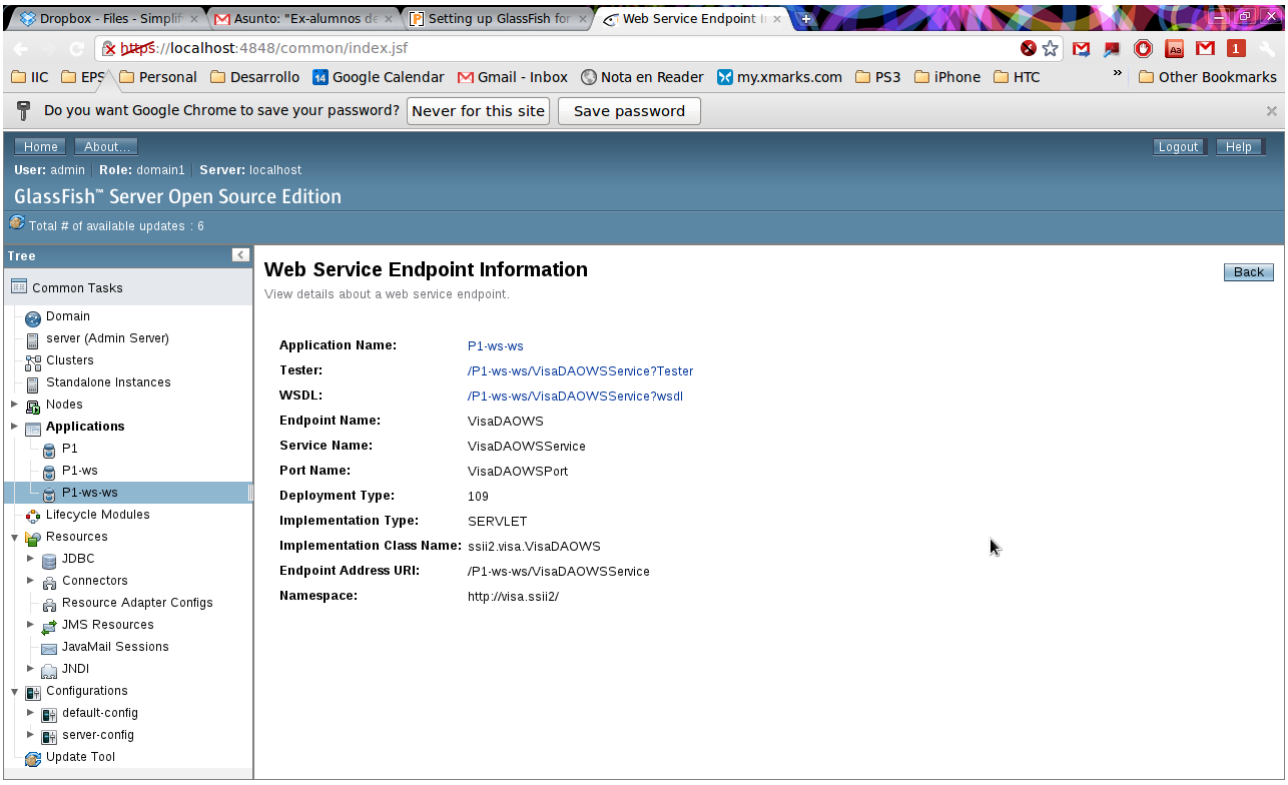


Figura 9. Vista del servicio web en el administrador de Glassfish

Del URL del servicio, indicado como “URI de dirección de punto final”, la parte “*P1-ws-ws*” es la ruta de contexto de los *servlet* que implementa funcionalidades sobre el servicio de visa (*VisaDAOWSService*).

El servicio se puede replegar y redesplegar con los objetivos adecuados contenidos en el archivo *build.xml*.

Ejercicio 7. Despliegue el servicio con la regla correspondiente en el *build.xml*. Acceda al WSDL remotamente con el navegador e inclúyalo en la memoria de la práctica (**habrá que asegurarse que la URL contiene la dirección IP de la máquina virtual donde se encuentra el servidor de aplicaciones**).

Comente en la memoria aspectos relevantes del código XML del fichero WSDL y su relación con los métodos Java del objeto del servicio, argumentos recibidos y objetos devueltos [5].

Conteste a las siguientes preguntas:

- ¿En qué fichero están definidos los tipos de datos intercambiados con el webservice?
- ¿Qué tipos de datos predefinidos se usan?
- ¿Cuáles son los tipos de datos que se definen?
- ¿Qué etiqueta está asociada a los métodos invocados en el webservice?
- ¿Qué etiqueta describe los mensajes intercambiados en la invocación de los métodos del webservice?
- ¿En qué etiqueta se especifica el protocolo de comunicación con el webservice?
- ¿En qué etiqueta se especifica la URL a la que se deberá conectar un cliente para acceder al webservice?

[5] Para más información sobre los elementos de un fichero WSDL, ver https://www.tutorialspoint.com/wsdl/

7.7 Modificar el cliente (web)

El cliente del servicio web se emplea en aquellos *servlets* que originalmente invocaban métodos de la clase VisaDAO. En su implementación original, el servlet **ProcesaPago** era el encargado de realizar sendas llamadas a los métodos ***compruebaTarjeta***, tomando como parámetro a una tarjeta (*TarjetaBean*) cuya corrección será comprobada para a continuación hacer otra llamada a ***realizaPago***. Esta segunda llamada toma como parámetro a una clase *PagoBean*, que es devuelta modificada con los valores de la transacción y del código asociado.

Puesto que ahora el cliente ya no debería disponer de ninguna instancia de la clase VisaDAO, se deben realizar modificaciones para que la invocación se haga a partir de los *stubs* imagen del servicio remoto. Para ello habrá que llevar a cabo ciertas modificaciones en el cliente descritas más abajo.

Nótese que los stubs pueden ser:
- *Stubs estáticos:* se crean en tiempo de compilación.
- *Stubs dinámicos:* se crean en tiempo de ejecución.

En este apartado realizaremos las modificaciones necesarias para que los *servlet* cliente invoquen al servicio remoto mediante *stubs* estáticos. Los siguientes *import* de Java deben emplearse allí donde se quiera usar el servicio remoto:

```
...
import ssii2.visa.VisaDAOWSService; // Stub generado automáticamente
import ssii2.visa.VisaDAOWS; // Stub generado automáticamente
import javax.xml.ws.WebServiceRef;
...
```

La instanciación de la clase remota pasa a hacerse en dos pasos:

- **VisaDAOWSService**. Es una referencia al servicio remoto, que nos permitirá obtener el *stub* local. Se instancia una sóla vez en el método que invoque al servicio:

  ```
  VisaDAOWSService service = new VisaDAOWSService();
  ```

- **VisaDAOWS** (versión stub). No debe ser instanciado directamente, sino obtenido a partir del objeto VisaDAOWSService)

  ```
  VisaDAOWS dao = service. getVisaDAOWSPort ();
  ```

Ejercicio 8. Realícese las modificaciones necesarias en ProcesaPago.java para que implemente de manera correcta la llamada al servicio web mediante *stubs* estáticos. Téngase en cuenta que:

- El nuevo método realizaPago() ahora no devuelve un boolean, sino el propio objeto Pago modificado.
- Las llamadas remotas pueden generar nuevas excepciones que deberán ser tratadas en el código cliente.

  Incluye en la memoria una captura con dichas modificaciones.

Una aproximación intermedia entre *stubs* estáticos y dinámicos puede ser modificar la ruta del servidor remoto sin cambiar la definición del servicio. Esto puede ser útil si se migra dicho servicio a una ubicación física distinta. Se haría de la siguiente forma[6]:

[6] La ruta al servidor remoto a indicar en una variable del fichero web.xml podéis obtenerla dentro del elemento <service> del fichero WSDL del servicio web.

```
BindingProvider bp = (BindingProvider) dao;
bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
"http://url.al.servidor.remoto”);
```

Ejercicio 9. Modifique la llamada al servicio para que la ruta al servicio remoto se obtenga del fichero de configuración web.xml. Para saber cómo hacerlo consulte el apéndice 15.1 para más información y edite el fichero web.xml y analice los comentarios que allí se incluyen.

Ejercicio 10. Siguiendo el patrón de los cambios anteriores, adaptar las siguientes clases cliente para que toda la funcionalidad de la página de pruebas testbd.jsp se realice a través del servicio web. Esto afecta al menos a los siguientes recursos:

- Servlet DelPagos.java: la operación dao.delPagos() debe implementarse en el servicio web.
- Servlet GetPagos.java: la operación dao.getPagos() debe implementarse en el servicio web.

Tenga en cuenta que no todos los tipos de datos son compatibles con JAXB (especifica como codificar clases java como documentos XML), por lo que es posible que tenga que modificar el valor de retorno de alguno de estos métodos. Los apéndices contienen más información. Más específicamente, se tiene que modificar la declaración actual del método getPagos(), que devuelve un PagoBean[], por:

public ArrayList<PagoBean> getPagos(@WebParam(name = "idComercio") String idComercio)

Hay que tener en cuenta que la página listapagos.jsp espera recibir un array del tipo PagoBean[]. Por ello, es conveniente, una vez obtenida la respuesta, convertir el ArrayList a un array de tipo PagoBean[] utilizando el método toArray() de la clase ArrayList.

Incluye en la memoria una captura con las adaptaciones realizadas.

7.8 Nuevas reglas de compilación del cliente

En el fichero build.xml se han añadido varios targets adicionales para compilar la parte cliente:

1. generar-stubs.
2. compilar-cliente.
3. empaquetar-cliente.
4. desplegar-cliente.

7.8.1 Generar los *stubs* del cliente (web)

El paquete Java cliente de un servicio web, necesitará que sus *stubs* sean autogenerados a partir de la definición del servicio (WSDL) remoto.

La herramienta *wsimport* permite añadir las clases anteriores, así como otros posibles objetos intermedios (javabeans) que sean necesarios como parámetros o valores de retorno de los procedimientos remotos.

Para ello se toma el archivo WSDL que ha sido desplegado con el servicio. Dado que es necesario el archivo WSDL para poder generar los *stubs*, el servicio deberá estar desplegado en el servidor.

Se usa de la siguiente forma:

```
wsimport –d <directorio_salida_clases> -p paquete http://ruta/al/wsdl
```

Donde:

- directorio_salida_clases: el directorio donde se generarán los .java intermedios. Estas clases serán necesarias para que las llamadas remotas cuenten con la definición de todos los objetos necesarios. En nuestro proyecto usaríamos build/client/WEB-INF/classes.
- paquete: paquete Java, epígrafe bajo el que se definirán todas las clases generadas. En nuestro caso, sería ssii2.visa
- Ruta al WSDL: URL al fichero wsdl ya desplegado en el servidor.

Se puede usar la opción `-Xnocompile` para revisar el código Java generado.

Ejercicio 11. Realice una importación manual del WSDL del servicio sobre el directorio de clases local. Anote en la memoria qué comando ha sido necesario ejecutar en la línea de comandos, qué clases han sido generadas y por qué. **Téngase en cuenta que el servicio debe estar previamente desplegado.**

Ejercicio 12. Complete el target *generar-stubs* definido en build.xml para que invoque a wsimport (utilizar la funcionalidad de ant exec para ejecutar aplicaciones). Téngase en cuenta que:

- El raíz del directorio de salida del compilador para la parte cliente ya está definido en build.properties como ${build.client}/WEB-INF/classes
- El paquete Java raíz (ssii2) ya está definido como ${paquete}
- La URL ya está definida como ${wsdl.url}

7.8.2 Compilar el cliente (web)

Una vez se ha codificado el cliente, se puede ejecutar la herramienta *ant* con el objetivo *compilar-cliente*:

```
ant compilar-cliente
```

Esto compila los archivos con referencias y usos del Servicio Web *en los directorios de fuente del cliente* y escribe las clases resultantes (el resto de clases del paquete) en el subdirectorio *build/client*.

7.8.3 Empaquetar, desplegar y ejecutar el cliente (web)

Una vez compilado el cliente se puede empaquetar en un .war, como hace el objetivo *empaquetar-cliente*:

```
ant empaquetar-cliente
```

Con esto se obtiene un archivo llamado P1-ws.jar en el subdirectorio *dist*, el cual debe desplegarse en el servidor de aplicaciones con el objetivo *desplegar-cliente*:

```
ant desplegar-cliente
```

Y se podrá ejecutar el cliente, en tecleando en el navegador la URL:

```
http://<IP_PC2VM>:8080/P1-ws-cliente
```

Ejercicio 13:

- Realice un despliegue de la aplicación completo en **dos nodos** tal y como se explica en la Figura 8. Habrá que tener en cuenta que ahora en el fichero build.properties hay que especificar la dirección IP del servidor de aplicaciones donde se desplegará la parte del cliente de la aplicación y la dirección IP del servidor de aplicaciones donde se desplegará la parte del servidor. Las variables **as.host.client** y **as.host.server** deberán contener esta información.
- Probar a realizar pagos correctos a través de la página testbd.jsp. Ejecutar las consultas SQL

necesarias para comprobar que se realiza el pago. Anotar en la memoria práctica los resultados en forma de consulta SQL y resultados sobre la tabla de pagos.

Incluye evidencias en la memoria de la realización del ejercicio.

Cuestiones:

Cuestión 1. Teniendo en cuenta el diagrama de la Figura 3, indicar las páginas html, jsp y servlets por los que se pasa para realizar un pago desde pago.html, pero en el caso de uso en que se introduce una tarjeta cuya fecha de caducidad ha expirado.

Cuestión 2. De los diferentes servlets que se usan en la aplicación, ¿podría indicar cuáles son los encargados de solicitar la información sobre el pago con tarjeta cuando se usa pago.html para realizar el pago, y cuáles son los encargados de procesarla?

Cuestion 3. Cuando se accede a pago.html para hacer el pago, ¿qué información solicita cada servlet? Respecto a la información que manejan, ¿cómo la comparten? ¿dónde se almacena?

Cuestión 4. Enumere las diferencias que existen en la invocación de servlets, a la hora de realizar el pago, cuando se utiliza la página de pruebas extendida testbd.jsp frente a cuando se usa pago.html. ¿Podría indicar por qué funciona correctamente el pago cuando se usa testbd.jsp a pesar de las diferencias observadas?

8. Entrega

La fecha de entrega de esta parte es la semana del 24 de febrero al 28 de febrero de 2020, antes del comienzo de la clase.

La **entrega de los resultados de esta práctica se regirá por las normas expuestas durante la presentación de la asignatura**. El incumplimiento de estas normas conllevará a considerar que la práctica no ha sido entregada en tiempo. Esto implica que se tendrá que volver a enviar correctamente y que se aplicará la penalización por retraso pertinente. Nomenclatura del fichero a entregar: SI2P1A_<grupo>_<pareja>.zip (ejemplo: SI2P1A_2311_1.zip)

• Contenido del fichero:

- Informe técnico siguiendo la plantilla publicada en la página del laboratorio con las respuestas a todas las preguntas
- P1-base con las modificaciones necesarias para la primera parte.
- P1-ws con las modificaciones necesarias para la segunda parte.

9. Bibliografía recomendada

- *API Java EE 7 (http://download.oracle.com/javaee/7/api/)*
- *Java EE 7 Tutorial*: http://download.oracle.com/javaee/7/tutorial/doc/

Apéndices

10. Apéndice sobre acceso a datos / JDBC

10.1 Conexión directa

Para conectar de forma directa lo primero que debemos hacer es instanciar el *driver* JDBC del gestor de bases de datos. Para ello bastará el siguiente fragmento de código (ver la clase DBTester.java):

```
Class.forName("org.postgresql.Driver").newInstance();
```

A continuación, podemos utilizar el método estático de la clase *DriverManager* para obtener nuevas conexiones, para lo cual requeriremos conocer la cadena de conexión JDBC, usuario y clave:

```
Connection c = DriverManager.getConnection (
    "jdbc:postgresql://host:puerto/nombreBD", "usuario", "clave"
);
```

10.2 Conexión mediante datasource

Para obtener el objeto *DataSource* deberemos ejecutar:

```
DataSource ds = (DataSource) new InitialContext().lookup("jdbc/<RecursoBD>");
```

A continuación podremos obtener conexiones de la siguiente forma:

```
Connection c = ds.getConnection();
```

10.3 Desconexión

La conexión en ambos casos se realiza:

```
c.close()
```

Únicamente en caso de ser una conexión directa la conexión se estará realmente cerrando, en conexiones a través de *datasource* estamos devolviendo la conexión al pool.

11. Apéndice sobre Servicios Web

Un Servicio Web (en inglés, ***Web Service***) es un conjunto de protocolos y estándares que sirven para intercambiar datos entre aplicaciones. Aplicaciones de software distintas, desarrolladas en lenguajes de programación diferentes, y ejecutadas sobre cualquier plataforma, pueden utilizar los servicios web para intercambiar datos en redes de ordenadores como Internet. La interoperabilidad necesaria se consigue mediante la adopción de estándares abiertos.

Las organizaciones OASIS y W3C son los comités responsables de la arquitectura y reglamentación de los servicios Web. Para mejorar la interoperabilidad entre distintas implementaciones de servicios Web se ha creado el organismo WS-I, encargado de desarrollar diversos perfiles para definir de manera más exhaustiva estos estándares.

11.1 Razones para crear servicios Web

La principal razón del éxito de los servicios Web es que se basan en el protocolo tan ampliamente distribuido y usado de Internet HTTP sobre TCP (Transmission Control Protocol). Dado que la mayor parte de organizaciones protegen sus redes mediante cortafuegos o firewalls -que filtran y bloquean gran parte del tráfico de Internet-, cierran casi todos los puertos TCP (salvo el 80 o 443 para HTTPS), que son,

precisamente, los que usan los navegadores. Los servicios Web utilizan este puerto, por la simple razón de que no resultan bloqueados.

Otra razón es que, antes de que existiera SOAP, no había buenas interfaces para acceder a las funcionalidades de otros ordenadores en red. Las que había eran ad hoc y poco conocidas, tales como EDI (*Electronic Data Interchange*), RPC (*Remote Procedure Call*), u otras APIs.

Una tercera razón por la que los servicios Web son muy prácticos es que pueden aportar gran independencia entre la aplicación que usa el servicio Web y el propio servicio. De esta forma, los cambios a lo largo del tiempo en uno no deben afectar al otro. Esta flexibilidad será cada vez más importante, dado que la tendencia a construir grandes aplicaciones a partir de componentes distribuidos más pequeños es cada día más utilizada.

Se espera que para los próximos años mejoren la calidad y cantidad de servicios ofrecidos basados en los nuevos estándares.

11.2 Ventajas de los servicios web

- Aportan interoperabilidad entre aplicaciones de software independientemente de sus propiedades o de las plataformas sobre las que se instalen.
- Los servicios Web fomentan los estándares y protocolos basados en texto, que hacen más fácil acceder a su contenido y entender su funcionamiento.
- Permiten que servicios y software de diferentes compañías ubicadas en diferentes lugares geográficos puedan ser combinados fácilmente para proveer servicios integrados.

11.3 Inconvenientes de los servicios Web

Para realizar transacciones no pueden compararse en su grado de desarrollo con los estándares abiertos de computación distribuida como CORBA (Common Object Request Broker Architecture).

Su rendimiento es bajo si se compara con otros modelos de computación distribuida, tales como RMI (Remote Method Invocation), CORBA o DCOM (Distributed Component Object Model). Es uno de los inconvenientes derivados de adoptar un formato basado en texto. Y es que entre los objetivos de XML no se encuentra la concisión ni la eficacia de procesamiento.

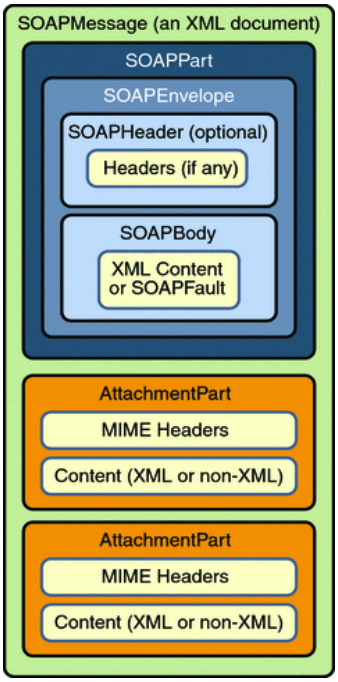
Al apoyarse en HTTP, pueden esquivar medidas de seguridad basadas en firewall cuyas reglas tratan de bloquear o auditar la comunicación entre programas a ambos lados de la barrera.

11.4 Plataformas

Algunos servidores de aplicaciones que implementan Servicios Web son:

- JAX-WS con GlassFish
- JBoss servidor de aplicaciones JAVA EE Open Source de Red Hat inc.
- Axis y el servidor Jakarta Tomcat (de Apache)
- Java Web Services Development Pack (JWSDP) de Sun Microsystems (basado en Jakarta Tomcat)
- WebLogic
- WebSphere
- Microsoft .NET
- ColdFusion MX de Macromedia
- Oracle Fusion Middleware
- …

11.5 Entorno JAX-WS



En esta práctica, en lo que se refiere a los Servicios Web, se va a utilizar la API y herramientas JAX-WS de Java, que facilitan el desarrollo de servicios web. En JAX-WS, una llamada a un servicio web se representa por un protocolo basado en XML como SOAP. La especificación de SOAP define la estructura del "sobre" (SOAP es orientado a mensajes), reglas de codificación y convenciones para representar las llamadas y respuestas a los procedimientos remotos.

Estas llamadas y respuestas se transmiten como mensajes SOAP (en archivos XML) sobre HTTP. Aunque los mensajes SOAP pueden resultar complejos, la API de JAX-WS oculta esta complejidad del desarrollador de la aplicación. En el lado del servidor, el desarrollador especifica los procedimientos remotos definiendo y codificando métodos en una clase escrita en Java. Dicha clase y sus métodos deben contener anotaciones que los definan como servicios web. Los programas clientes también son fáciles de codificar. Un cliente crea un *proxy*, un objeto local que representa el servicio, y simplemente invoca los métodos del *proxy*. Con JAX-WS, el desarrollador no genera ni analiza los mensajes SOAP, sino que en tiempo real se convierten las llamadas a su API a mensajes SOAP. JAX-WS no se restringe a Java, pues puede acceder a servicios Web y recibir peticiones implementadas en otra plataforma distinta (.Net, Perl, etc.). Esto es posible porque se hace uso de estándares tales como HTTP, SOAP y WSDL.

11.6 Codificar y compilar el servidor

El primer paso para obtener el servicio web es codificar las clases relativas al servicio web, lo que se conoce como *Service Endpoint* o Punto de Acceso al Servicio. Dicha codificación se puede realizar en una interfaz y una implementación de dicha interfaz, o directamente en una clase que implemente el servicio. Para ello se emplean un conjunto de anotaciones, que serán empleadas posteriormente en el proceso de generación de código. En cualquier caso se deben cumplir las siguientes reglas:

- La clase que implemente el servicio debe incluir anotaciones tales como *javax.jws.WebService (@WebService) o javax.jws.WebServiceProvider.*
- La clase que implementa el servicio puede incluir una referencia explícita al *Service Endpoint* usando el elemento *endpointInterface* de la anotación *@WebService*, pero no es obligatorio. Si este elemento no se especifica, el *Service Endpoint* se define implícitamente para dicha clase.
- La clase que implementa el servicio no debe ser declarada final (*final*), y tampoco puede ser abstracta (*abstract*).
- La clase que implementa el servicio debe tener un constructor público por defecto, sin parámetros.
- La clase que implementa el servicio no debe definir el método *finalize*().
- Los métodos de la clase que proporcionan el servicio deben declararse como públicos (*public*), y no deben ser declarados ni estáticos (*static*) ni finales (*final*).
- Los métodos que proporcionan el servicio deben anotarse con *javax.jws.WebMethod (@WebMethod).*
- Los métodos que proporcionan el servicio deben tener **parámetros de entrada y de salida compatibles con JAXB**. Esto incluye la mayor parte de tipos simples de Java (String, int, etc.), y JavaBeans que únicamente reciban o devuelvan dichos tipos de datos. En el caso de los

JavaBeans es necesario explicitar qué propiedades se consideran elementos JAXB. Para ello hay que usar la anotación *javax.xml.bind.annotation.XmlElement (@XmlElement*). Para más información en este sentido, consulte el tutorial de Java EE:

- http://java.sun.com/javaee/5/docs/tutorial/doc/bnazq.html

- La clase puede incluir las anotaciones *javax.annotation.PostConstruct* y *javax.annotation.PreDestroy* en los métodos que se creen para las tareas relacionadas con el ciclo de vida. El método anotado con *@PostConstruct* se llama por el servidor de aplicaciones antes de que la clase que implementa el servicio comience a responder a solicitudes de los clientes. El método anotado con *@PreDestroy* se llama por el contenedor antes de que el servicio deje de estar operativo.

12. Apéndice sobre JSPs

12.1 ¿Qué es una página JSP?

Una página JSP es un documento de texto que contiene dos tipos de texto: un conjunto de datos estáticos, que pueden ser expresados en cualquier formato de texto, como HTML, XML, etc.; y elementos JSP, que son los que construyen el contenido dinámico.

Las etiquetas son reconocibles por los caracteres “<” y “>”. JSP acepta etiquetas estándar y también creados por el usuario. Por ejemplo, el aspecto de una página JSP que permite convertir entre pesetas y euros es el siguiente:

```
<%@ page language="java" import="ConverterBean"%>
<jsp:useBean id="help"scope="request" class="ConverterBean" />
<jsp:setProperty name="help" property="*"/>
<html>
      <body>
      Resultado:
      <%= Integer.parseInt(request.getParameter("mode")) == 1 ?
help.PtasToEuros(Double.parseDouble(request.getParameter("valor"))) :
help.EurosToPtas(Double.parseDouble(request.getParameter("valor")))
      %>
</body>
</html>
```

Las etiquetas incluyen:

- contenido estático **(<html>**, **<body>**, etc) que suponemos ya conocidos
- acceso a componentes de JavaBean e instanciarlos: **<jsp:useBean …>**
- poner un atributo a un JavaBean **<jsp:setProperty …>**
- Directivas que se aplican a toda la página (**<%@ page ...%>**), que en este caso importan las clase Java ConverterBean, y establecen el tipo de lenguaje usado en los *Scriptlets* como Java.

Aparate de estos elementos se puede

- introducir pequeños segmentos de código basado en Java conocido como *scriptlet* delimitados por “<%” y “%>”
- Expresiones (**<%= ... %>**), que usamos para insertar el valor devuelto una sentencia

12.2 El valor de una variable. ${…}. El ciclo de vida de una página JSP

Un *servlet* maneja las peticiones a páginas JSP. Esto es, el ciclo de vida y muchas de las capacidades de las páginas JSP (en los aspectos dinámicos) son determinados por la tecnología Java *Servlet*.

Cuando se realiza una petición a una página JSP, esta es manejada por un *servlet* especial que primero chequea si el *servlet* de la página JSP es más antiguo que la página JSP. Si es así, se traduce la página JSP a una clase *servlet* y se compila la clase. Durante el desarrollo, una de las ventajas de las páginas JSP sobre *servlets* es que el proceso de construcción se realiza de forma automática.

12.2.1 Traducción y Compilación

Durante la fase de traducción, cada tipo de dato de la página JSP se trata de forma diferente. Los datos de la página web (HTML, etc.) se convierten en código que enviará los datos al stream de datos devuelto al cliente. Los elementos JSP son tratados de la siguiente manera:

- Las directivas se usan para controlar como el contenedor Web traduce y ejecuta la página JSP.
- Los elementos de *scripting* se insertan en la clase *servlet* de la página JSP.
- Los elementos de la forma *<jsp:XXX ... />* son convertidos en llamadas a métodos de componentes JavaBeans o invocaciones al API Java *Servlet*.

Para una página JSP llamada *Nombre_pagina*, el código fuente del *servlet* de la página JSP es mantenido en el directorio /opt/glassfish4/glassfish/domains/domain1/generated/jsp/2ee-modules/petstore/… /Nombre_pagina_jsp.java.

Tanto las fases de traducción como de compilación pueden provocar errores que solo son observados cuando la página es solicitada la primera vez. Si un error ocurre durante la traducción, el servidor devolverá un *ParseException*, y el código fuente de la clase del *servlet* estará vacío o incompleto. La última línea incompleta apuntará al elemento incorrecto.

Si un error ocurre durante la compilación, el servidor devolverá un *JasperException* y un mensaje que incluirá el nombre del *servlet* de la página JSP y la línea donde ocurrió el error.

Una vez la página ha sido traducida y compilada:

- Si una instancia del *servlet* de la página JSP no existe, el contenedor:
    1. Carga la clase del *servlet* de la página JSP.
    2. Crea una instacia de la clase *servlet*.
    3. Inicializa la instancia del *servlet* llamando al método *jspInit*.
- Invoca al método *_jspService*, pasándole un objeto request y otro response.

Si el contenedor necesita eliminar el *servlet* de la página JSP, llama al método *jspDestroy*.

12.2.2 Ejecución

Es posible controlar varios parámetros de la ejecución mediante las directivas *page*. Estas directivas están relacionadas con la regulación de la salida y el manejo de errores.

12.2.3 Regulación de la salida

Cuando una página JSP se ejecuta, la salida escrita en el objeto respuesta es automáticamente colocada en un buffer. Es posible establecer el tamaño del buffer con la siguiente directiva *page*:

```
<%@ page buffer="none|xxxkb" %>
```

Un buffer de gran tamaño permite escribir un mayor contenido antes de que todo sea enviado al cliente. Esto permite a la página JSP disponer de más tiempo para establecer los códigos de estado y las cabeceras o para remitir a otro recurso Web. Un buffer pequeño reduce la carga de memoria del servidor y permite al cliente comenzar a recibir datos más rápidamente.

12.2.4 Manejo de errores

Cuando se ejecuta una página JSP se pueden producir un gran número de excepciones. Para especificar al contenedor Web que remita el control a una página de error si ocurre una excepción, incluir la siguiente directiva *page* al principio de la página JSP:

```
<%@ page errorPage="file_name" %>
```

Al principio de la página de error se debe indicar que se trata de una página de error con la siguiente directiva *page*:

```
<%@ page isErrorPage="true|false" %>
```

Esta directiva hace el objeto excepción disponible en la página de error, de modo que se puede recuperar, interpretar y mostrar cualquier información sobre la causa de la expeción.

12.2.5 Elementos de scripting de JSP

Los elementos de *scripting* se usan para crear y acceder a objetos, definir métodos, y manejar el flujo de control. Uno de los objetivos de la tecnología JSP es separar los datos estáticos del código necesario para generar contenido dinámico.

La tecnología JSP permite a un "contenedor" soportar cualquier lenguaje de *scripting* que pueda llamar a objetos Java. Si se desea usar un lenguaje de *scripting* que no sea el por defecto, Java, se debe especificar en una directiva **page** al principio de la página JSP:

<%@ page language="lenguaje_*scripting*" %>

Como los elementos de *scripting* son convertidos a declaraciones del lenguaje de programación en la clase *servlet* de la página JSP, se debe importar cualquier clase y package usado en la página JSP. Si el lenguaje es Java, se debe importar una clase o *package* con la directiva *page*:

```
<%@ page import="nombre_package.*, nombre_clase" %>
```

12.2.6 Declaraciones

Una declaración en JSP se usa para declarar variables y métodos en el lenguaje de *scripting* de la página. La sintaxis para las declaraciones es:

```
<%! declaración_lenguaje_scripting %>
```

Un ejemplo:

```
<%!
private int variable;

public void jspInit() {
...
}

public void jspDestroy() {
...
}
%>
```

Cuando el lenguaje de *scripting* es Java, las variables y métodos de las declaraciones JSP son declaraciones de la clase *servlet* de la página JSP.

12.2.7 Scriptlets

Un scriptlet JSP se usa para contener cualquier fragmento de código que es válido para el lenguaje de *scripting* usado en la página. La sintaxis para los scriptlets es:

<% instrucciones_lenguaje_*scripting* %>

Un ejemplo:

```
<%

Iterator i = lista.getItems().iterator();
while(i.hashNext()) {

MiLista item = (MiLista)i.netx();

%>

<tr>
<td align="right" bgcolor="#ffffff">
<%= item.getQuantity() %>
<td>
...

<%

// fin while
}

%>
```

Cuando el lenguaje de *scripting* es Java, un scriptlet se transforma en un fragmento de programa Java y es insertado en un método del *servlet* de la página JSP.

12.2.8 Expresiones

Una expresión en JSP se usa para insertar un valor de una expresión en lenguaje de *scripting*, convertido a cadenas de caracteres, y devolverlo como dato al cliente. Cuando el lenguaje de *scripting* es Java, la expresión se transforma en una instrucción que convierte la expresión a un objeto String.

La sintaxis para una expresión es:

<%= expresión_lenguaje_*scripting* %>

Es importante conocer que no se permite el uso del ';' en las expresiones JSP, incluso si la expresión tiene un ';' cuando se usa dentro de un scriplet.

En el apartado anterior, `<%= item.getQuantity() %>` es un ejemplo de expresión.

Las expresiones también se pueden representar utilizando `${ expresión }`

Los *scriptlets* y expresiones pueden contener objetos implícitos (predefinidos):

- `pageContext`: el contexto de la página JSP. Proporciona acceso a varios objetos, incluyendo:
  - `servletContext`: el contexto del *servlet* de la página JSP y cualquier componente Web incluido en la misma aplicación.
  - `session`: el objeto que contiene la sesión de un cliente.
  - `request`: el objeto que contiene la solicitud enviada al servidor.
  - `response`: el objeto que contiene la respuesta que el servidor envía al cliente.

Además, también hay otros objetos implícitos disponibles para facilitar el acceso a la siguiente información:

- `param`: correspondencia entre el nombre de un parámetro de la solicitud HTTP y un valor único.
- `paramValues`: correspondencia entre el nombre de un parámetro de la solicitud HTTP y un array de valores.
- `header`: correspondencia entre el nombre de un parámetro de la cabecera y un valor único.
- `headerValues`: correspondencia entre el nombre de un parámetro de la cabecera y un array de valores.
- `cookie`: correspondencia entre el nombre de una cookie y su valor.
- `initParam`: correspondencia entre el nombre de un parámetro de inicialización y un valor único.

Por ejemplo, la expresión `${initParam.nombre_de_parametro}` devolvería el valor del parámetro llamado `nombre_de_parametro` que estuviera incluido en el archivo web.xml.

12.2.9 Inicializar y finalizar una página JSP

Es posible realizar un proceso que permita a la página JSP leer datos de configuración persistentes, inicializar recursos y realizar cualquier otra actividad "one-time" sobreescribiendo el método *jspInit* de la interfaz *JspPage*. Para la liberación de los recursos se usa el método *jspDestroy*.

12.2.10 Incluir contenido en una página JSP

Existen dos mecanismos para incluir otros recursos Web en páginas JSP: la directiva *include* y el elemento *jsp:include*.

La directiva *include* se procesa cuando la página JSP se traduce a una clase *servlet*. El efecto de la directiva es insertar el contenido de texto de otro fichero -ya sea estático u otra página JSP- en la página JSP que la incluye. La directiva *include* es útil para incluir banners, información de copyright, o cualquier tipo de contenido que se quiera reusar. La sintaxis para la directiva *include* es:

<%@ include file="nombre_fichero" %>

El elemento *jsp:include* se procesa cuando la página JSP se ejecuta. Si el recurso es estático, su contenido es insertado cuando se llama al fichero JSP. Si el recurso es dinámico, la petición es enviada al recurso incluido, la página incluida es ejecutada, y entonces el resultado es incluido en la respuesta de la llamada a la página JSP. La sintaxis para el elemento *jsp:include* es el siguiente:

<jsp:include "nombre_fichero" />

12.2.11 Transferir el control a otro componente Web

El mecanismo para transferir el control a otro componente Web desde una página JSP emplea la funcionalidad que provee el API de *Servlets* de Java. Se puede acceder a esta funcionalidad desde una página JSP con el elemento *jsp:forward*:

<jsp:forward page="/main.jsp" />

Es importante conocer que si se ha devuelto cualquier dato al cliente, el elemento *jsp:forward* fallará con una *IllegalStateException*.

Cuando un elemento *include* o *forward* se invoca, se le pasa el objeto petición a la página objetivo. Si se desea proveer de datos adicionales a esa página, se pueden añadir parámetros al objeto de petición con el elemento *jsp:param*:

```
<jsp:include page="..." >
        <jsp:param name="param1" value="valor" />
</jsp:include>
```

13. Apéndice sobre JavaBeans en páginas JSP

Como se indicaba en la práctica anterior, los componentes JavaBeans son clases Java que se pueden reutilizar fácilmente para componer aplicaciones. Cualquier clase Java que siga ciertas convenciones de diseño es un componente JavaBeans.

La tecnología Java Server Pages soporta el uso de JavaBeans como elementos estándar de las páginas JSP. Dichos beans se pueden crear para obtener o modificar el valor de sus propiedades

13.1 Convenciones de diseño de componentes JavaBeans

Las convenciones de diseño de los componentes JavaBeans definen las propiedades de las clases así como los métodos públicos que proporcionan acceso a éstas.

Una propiedad de un componente JavaBeans puede ser accesible en lectura, escritura o ambas. A la vez, dicha propiedad debe ser simple (contiene un único valor) o indexada (contiene un array de valores).

Una propiedad no tiene por qué estar implementada como un ejemplar de variable, sino que únicamente debe ser accesible usando los métodos públicos según las siguientes convenciones:

- Para cada propiedad que se pueda leer, el *bean* debe tener un método de la forma

  PropertyClass getProperty() { ... }
- Para cada propiedad que se pueda escribir, el *bean* debe tener un método de la forma

  setProperty(PropertyClass pc) { ... }
- Además, estos componentes deben poseer un constructor sin parámetros.

Por ejemplo, el *bean PagoBean* posee una propiedad que se puede leer y escribir, dado que posee los métodos getValor y setValor.

13.2 Uso de JavaBeans en páginas JSP

Para declarar que se usa un *bean* en una página JSP se utiliza la etiqueta jsp:useBean. Esto se puede hacer de dos formas:

```
<jsp:useBean id="nombreBean"
  class="nombre.calificado.de.clase" scope="ámbito"/>
```

y

```
<jsp:useBean id="nombreBean"
  class="nombre.calificado.de.clase" scope="ámbito" >
  <jsp:setProperty .../>
</jsp:useBean>
```

La segunda forma se utiliza si se quiere incluir cláusulas jsp:setProperty para inicializar las propiedades del *bean*.

El elemento jsp:useBean declara que la página utilizará un *bean* y que es accesible para el ámbito especificado que puede ser *application* (para toda la aplicación), *session* (sólo para la sesión actual), *request* (sólo para la solicitud actual) o *page* (sólo para la página actual). Si este *bean* no existía previamente, esta cláusula lo creará y almacenará como un atributo del ámbito para el que se define. El valor del atributo *id* determina el nombre del *bean* en el ámbito y el identificador que se usará para referenciar al *bean* en otros elementos y expresiones en la página JSP. El valor proporcionado para el atributo class debe ser un nombre de clase calificado (incluyendo el paquete al que pertenece).

13.3 Modificación de propiedades de JavaBeans

La manera estándar de modificar propiedades de un *bean* en una página JSP es con el elemento jsp:setProperty, que puede tener varias sintaxis:

Valor de la fuente	Sintaxis
Cadena de caracteres constante	<jsp:setProperty name="nombreBean" property="nombrePropiedad" value="valor

	constante"/>
Parámetro de la solicitud	<jsp:setProperty name="nombreBean" property="nombrePropiedad" param="nombreParámetro"/>
Parámetro de la solicitud cuyo nombre concuerda con el de la propiedad del *bean*	<jsp:setProperty name="nombreBean" property="nombrePropiedad"/> <jsp:setProperty name="nombreBean" property="*"/>
Expresión	<jsp:setProperty name="nombreBean" property="nombrePropiedad" value="expresión"/> <jsp:setProperty name="nombreBean" property="nombrePropiedad" > <jsp:attribute name="valor"> expresión </jsp:attribute> </jsp:setProperty>

Hay que tener en cuenta que:

1. nombreBean debe ser el mismo que el que se ha declarado para el atributo id en el elemento jsp:useBean.
2. Debe existir un método setNombrePropiedad en el *bean*.
3. nombreParámetro debe ser el nombre de un parámetro de la solicitud.

13.4 Obtención de propiedades de JavaBeans

La principal forma de obtener valores de propiedades es utilizando el lenguaje de expresiones JSP. Para ello, se puede utilizar la siguiente expresión:

${nombreBean.nombrePropiedad}

Otra forma de obtener los valores de propiedades es utilizar el elemento jsp:getProperty. Este elemento convierte el valor de la propiedad en una cadena de caracteres e inserta su valor en el stream de respuesta:

<jsp:getProperty name="nombreBean" property="nombrePropiedad"/>

Al igual que antes, nombreBean debe ser el mismo que el que se especificó para el atributo id en el elemento jsp:useBean, y debe existir un método getNombrePropiedad en el *bean*.

14. Apéndice sobre localización de recursos

Finalmente merece la pena clarificar los distintos métodos de referenciar un recurso accesible desde una página web. Por recurso se entiende, por ejemplo, una página web estática, un *servlet*, una página JSP o un servicio web.

1. **Direccionamiento absoluto**. Se emplea la URL completa del recurso, en la cual se incluye la

máquina y puerto dónde se encuentra el recurso, más la ruta absoluta que ocupa dentro del servidor web. Esta forma de direccionamiento sólo se utilizará, si hay que acceder a un servidor remoto, ya que si no, portar la aplicación de un equipo sería costosa.

Ej. "http://mi.servidor.com:puerto/aplicacion/web.jsp"

2. **Relativo al servidor del recurso actual**. En este caso se omite la información sobre la máquina y el puerto, ya que se toma por defecto la del servidor que alberga el recurso actual. La ruta se especifica desde la raíz del árbol de ficheros del servidor. Se empleará cuando haya que acceder a otras aplicaciones del servidor.

   Ej. "/aplicacion/web.jsp"
3. **Relativo al directorio del recurso actual.** Se toma cómo raíz de la ruta el directorio de la página actual. Se utilizará en el caso de tener que acceder a otros recursos de la misma aplicación.

   Ej. "web.jsp", "../aplicacion/web.jsp"
4. **Relativo a la página actual.** La ruta hace referencia al propio recurso, por lo que sólo se puede utilizar cuando un recurso precisa referenciarse a sí mismo.

   Ej. ""

15. Apéndice sobre servlets

Los servlets son objetos java que se ejecutan en el servidor de aplicaciones en el contenedor de servlets. Cuando el servidor de aplicaciones recibe una petición HTTP que se corresponde con un servlet, se invoca el método doGet del servlet, en el caso de peticiones GET, o doPost, en el caso de peticiones de tipo POST. Dichos métodos reciben como argumentos la petición HTTP (una referencia a un objeto HTTPServletRequest) y la respuesta que enviará el servidor (una referencia a un objeto HTTPServletResponse). Este último objeto es útil para que el servlet personalice la respuesta del servidor.

La principal utilidad de los servlets es la de procesar información que se haya introducido a través de un formulario, y generar contenido dinámico o alterar el flujo de información de la aplicación. En particular, un servlet puede extraer del objeto de clase HTTPServletRequest los campos del formulario que haya completado el usuario. Para ello, se puede utilizar el método HTTPServletRequest.getParameter(String nombre_parametro).

El método HTTPServletRequest.getSession() devuelve una referencia a un objeto de tipo HTTPSession que representa la sesión de un usuario particular con el servidor de aplicaciones. Dicho objeto puede ser utilizado para pasar información relativa al usuario entre varios servlets, o entre servlets y paginas jsp. Se pueden utilizar los métodos HTTPSession.setAttribute y HTTPSession.getAttribute para dicho proósito.

Los servlets también pueden redirigir la petición HTTP a otro servlet, o a una página jsp, tras haber realizado cierto procesamiento. Para ello se puede utilizar un objeto de tipo RequestDispatcher, que se puede obtener del contexto del servlet (esto se explica en la siguiente sección) mediante el método ServletContext.getRequestDispatcher(). Para realizar la redirección, basta ejecutar el método RequestDispatcher.forward(HTTPRequest request, HTTPResponse response).

Finalmente, la configuración de los servlets en el servidor de aplicaciones se encuentra en el fichero web.xml. Dicho fichero contiene una relación de los servlets que habrá en la aplicación web, las clases que los implementarán, y la url a la que responderá cada servlet. También es posible especificar aquí parámetros para los servlets.

15.1 El contexto del servlet

Los *servlets* que se ejecutan en un mismo servidor de aplicaciones disponen de acceso a una zona de información compartida denominada contexto. El contexto es único e independiente de las sesiones de los usuarios. El contexto se obtiene mediante *getServletContext*() y permite a un *servlet*:

- **Pasar información a otros *servlet***: Cada *servlet* puede añadir objetos a ese contexto que son accesibles por el resto de los *servlet*. Para ello, al igual que los dos casos anteriores, se crea un

nuevo atributo que consta de un par nombre-valor donde el valor se corresponde con el objeto que se quiere añadir.

- **Escribir en los logs del servidor.**
- **Obtener parámetros de inicialización**: Se pueden acceder a los parámetros de inicialización especificados en el archivo web.xml mediante el método getServletContext().getInitParameter("nombre-parámetro").