UAM UNIVERSIDAD AUTONOMA DE MADRID		**Escuela Politécnica Superior Ingeniería Informática Prácticas de Sistemas Informáticos 2**			
Grupo	**2401**	**Práctica**	1A	**Fecha**	07/03/2021
Alumno/a		Lerma, Martínez, Francisco			
Alumno/a		Morato, Martín, Jesús			

PRÁCTICA 1: ARQUITECTURA DE JAVA EE (PRIMERA PARTE)

Para la práctica se da por hecho que el sistema ha sido configurado correctamente con el script vrtualip.sh y además la variable de entorno "export J2EE_HOME=/opt/glassfish4/glassfish" ha sido definida.

La IP de nuestra primera máquina será 10.1.4.1 al ser del grupo 1401, pareja 4 y primera máquina.

EJERCICIO 1:

En el fichero build.properties hemos realizado los siguientes cambios en la ip del host de la aplicación como se indicaba en el enunciado con la IP de nuestra máquina virtual, en este caso 10.1.4.1.

```
as.host=10.1.4.1
```

Y por otro lado también hemos modificado el fichero postgresql.properties para desplegar la base de datos (tanto el host como el cliente) en nuestra máquina virtual.

```
db.host=10.1.4.1
```

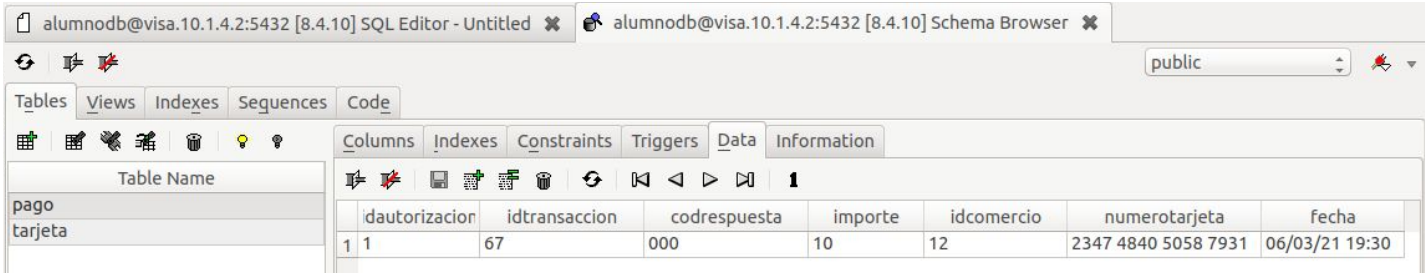
```
db.client.host=10.1.4.1
```

Lo siguiente fue abrir una nueva máquina virtual con la ip 10.1.4.1 y ejecutar el servidor mediante el comando "asadmin start-domain domain1". Una vez ejecutado lo anterior, en nuestro sistema ejecutamos los comandos ant compilar, implementar, regenerar-bd y desplegar. Una vez realizado los pasos anteriores ya tendríamos desplegada la aplicación de pagos Visa.

Tras desplegar nuestra aplicación y accediendo a http://10.1.4.1:8080/P1 encontramos un formulario (hemos puesto de idTransacción 67, idComercio 12 y un importe de 10). Una vez realizada esta nos aparece otro formulario y rellenamos con una tarjeta válida. Los datos correctos de una tarjeta válida los hemos obtenido del fichero inserta.sql.

Una vez completado el pago nos aparece que el pago ha sido realizado con éxito y nos muestra el comprobante.

Si abrimos Tora y nos conectamos la base de datos de nombre “visa” que se encuentra 10.1.4.1:5432 podemos acceder a través del Schema Browser Public a los datos, y vemos que en la tabla de los pagos aparece uno nuevo, que corresponde con el anterior.



Para comprobar que se ha realizado con éxito la compra, entraremos en la página de test de la base de datos y observamos que para el ID de comercio que habíamos puesto, efectivamente se ha realizado con éxito la transacción, como podemos ver en la siguiente captura:

Pago con tarjeta

Lista de pagos del comercio 12

idTransaccion	Importe	codRespuesta	idAutorizacion
67	10.0	000	1

Volver al comercio

Prácticas de Sistemas Informáticos II

Ademas el ejercicio también nos pide borrar la transacción por lo que procederemos a llevar a cabo el borrado con la opción que nos da en esta misma página y obtenemos lo siguiente:

Pago con tarjeta

Se han borrado 1 pagos correctamente para el comercio 12

Volver al comercio

Prácticas de Sistemas Informáticos II

Una vez borrado los pagos, se puede ver como si refrescamos en Tora el Schema Browser Public se ha borrado la columna con el pago y se ha quedado vacía la tabla. Igualmente si intentamos consultar los pagos del mismo comercio como hemos hecho en el apartado anterior, no obtendremos resultado.

Pago con tarjeta

Lista de pagos del comercio 12

idTransaccion	Importe	codRespuesta	idAutorizacion

Volver al comercio

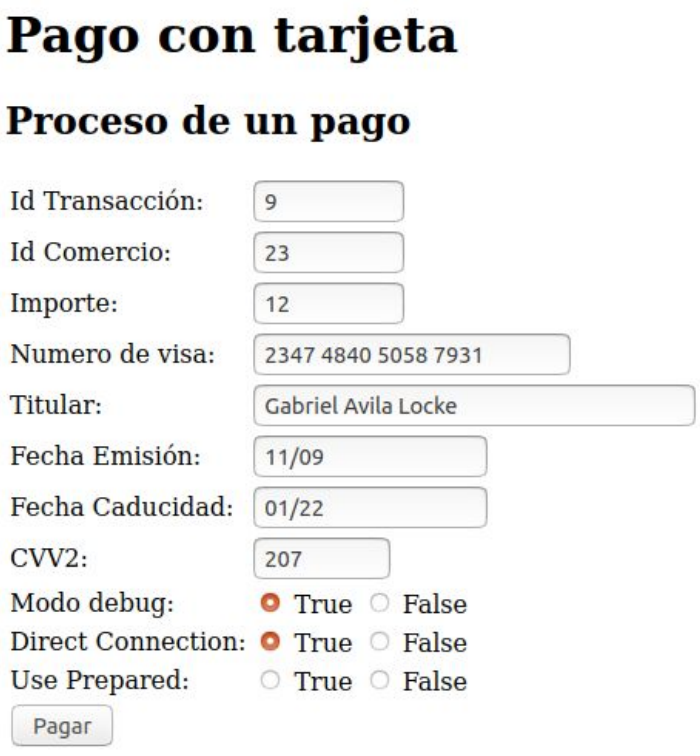
EJERCICIO 2:

Para este ejercicio vamos a cambiar una serie de macros en el fichero DBTester. java como se nos explica en el Apéndice 10, de forma que nos queda lo siguiente:

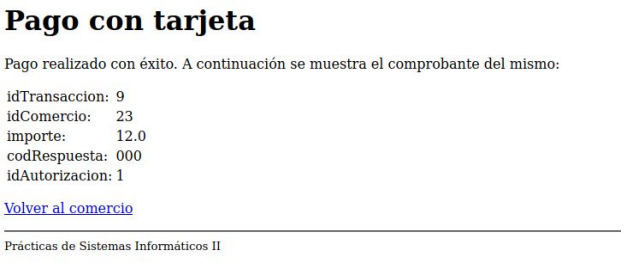
```
private static final String JDBC_DRIVER =
       "org.postgresql.Driver";

// TODO: Definir la cadena de conexion a la base de datos
/*****************************************************/
private static final String JDBC_CONNSTRING = "jdbc:postgresql://10.1.4.1:5432/visa";
/*****************************************************/
private static final String JDBC_USER = "alumnodb";
private static final String JDBC_PASSWORD = "****";
```

Utilizaremos el usuario y contraseña definidos en postgresql.properties y la ip y puerto que se especifica en db.port y db.host.
Ahora una vez hemos vuelto a redesplegar nuestra aplicación (hemos vuelto a compilar, empaquetar y redesplegar) seguiremos realizando los pasos que hemos hecho en el anterior ejercicio en la parte del test pero esta vez utilizando conexión directa como se nos indica en la siguiente captura:



A continuación vamos a poner una serie de capturas de comprobación de los pasos que hemos seguido para crear, listar y borrar la transacción:



Pago con tarjeta

Se han borrado 1 pagos correctamente para el comercio 23

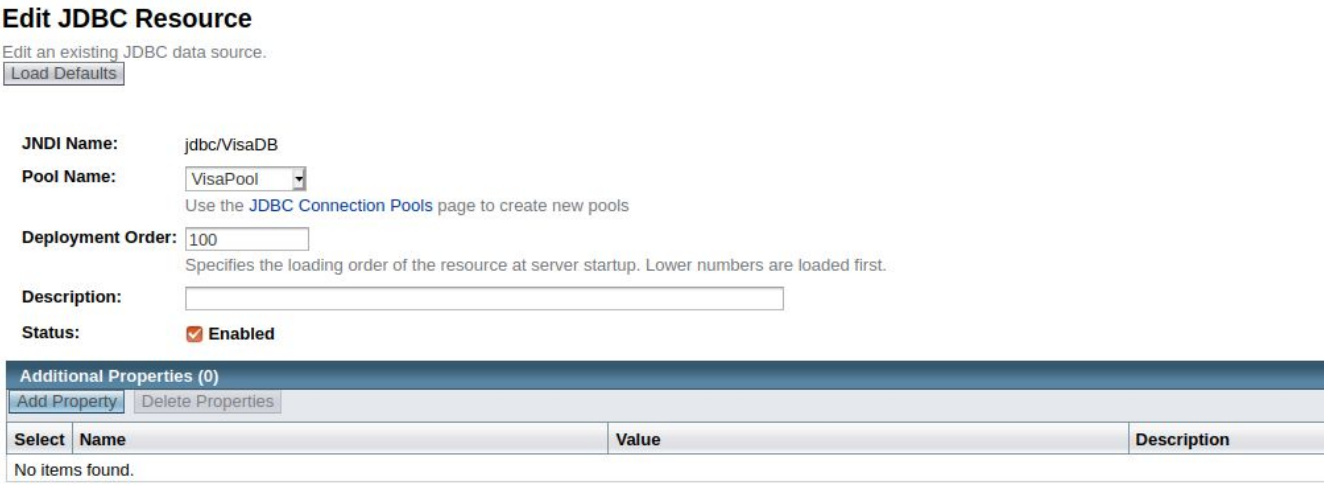
Volver al comercio

EJERCICIO 3:

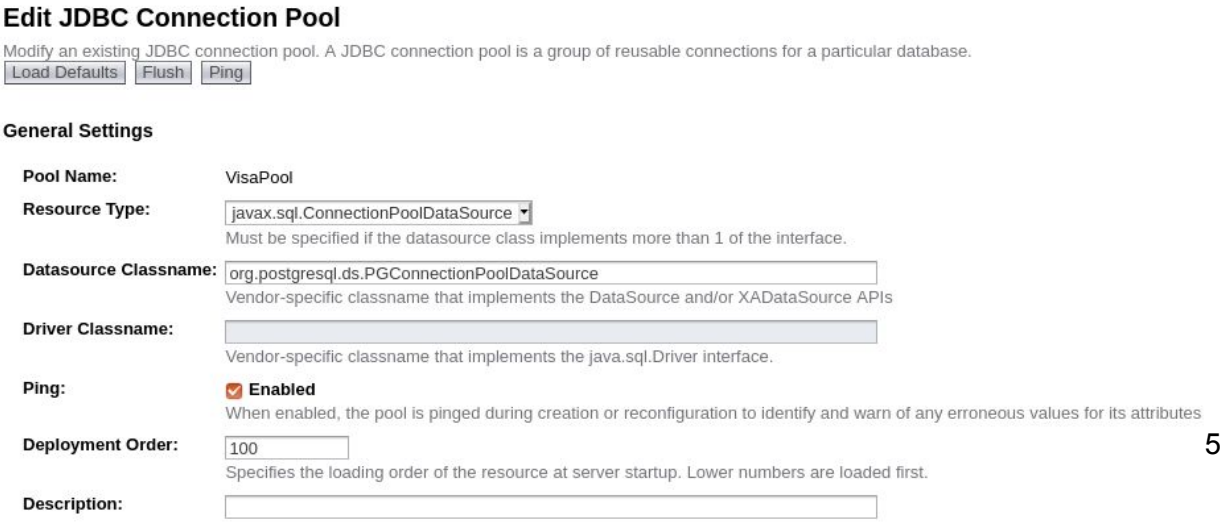
Vemos que el nombre del recurso JDBC correspondiente al DataSource es: jdbc/VisaDB y el nombre del pool: VisaPool

```
db.pool.name=VisaPool
db.jdbc.resource.name=jdbc/VisaDB
```

Veamos ahora que los recursos JDBC están correctamente creados, nos metemos en la Consola de Administración a traves de su dirección http://10.1.4.1:4848 , nos logueamos con el usuario “admin” y la contraseña “adminadmin” y nos dirigimos hasta la pestaña del menú de la izquierda que pone JDBC y vamos al subapartado “JDBC resources” y vemos que existe el recurso correspondiente al DataSource:



Si seleccionamos el subapartado “JDBC Connection Pools” podemos ver la pool “VisaPool” correctamente creada.



Si hacemos ping a la base de datos visa obtenemos lo siguiente (siendo 10.1.4.1 la IP de la base de datos y 5432 el puerto):

```
francisco@francisco:~/SI2/P1-base$ nmap -p 5432 10.1.4.2

Starting Nmap 7.60 ( https://nmap.org ) at 2021-03-07 06:37 CET
Nmap scan report for 10.1.4.2
Host is up (0.0022s latency).

PORT     STATE SERVICE
5432/tcp open  postgresql
```

Vemos que el host está activo, que es un servicio PostgreSQL y que está abierto en el puerto 5432 con un protocolo TCP.

Vemos además los ajustes que tiene el Pool y lo que se nos indica en el enunciado del ejercicio 3 qué debemos indicar sobre el Pool:

Pool Settings

Initial and Minimum Pool Size:	8	Connections
	Minimum and initial number of connections maintained in the pool	
Maximum Pool Size:	32	Connections
	Maximum number of connections that can be created to satisfy client requests	
Pool Resize Quantity:	2	Connections
	Number of connections to be removed when pool idle timeout expires	
Idle Timeout:	300	Seconds
	Maximum time that connection can remain idle in the pool	
Max Wait Time:	60000	Milliseconds
	Amount of time caller waits before connection timeout is sent	

Initial and Minimum Pool Size: es el número de hilos concurrentes que se crean al inicio para tratar las conexiones de los clientes. En nuestro caso empezamos con un tamaño inicial y mínimo de 8.
A más tamaño mínimo de pool con más conexiones empezaremos pero mayor coste computacional y de recursos tendrá en el servidor, lo cual podría afectar a su rendimiento. A menor tamaño menos coste pero podrían quedar clientes en espera y tendríamos menos concurrencia.

Maximum Pool Size: es el número máximo de hilos y por tanto de conexiones simultáneas que tendremos. En este caso serán 32 de tamaño máximo. A más tamaño máximo de pool más clientes podremos tener de manera simultánea, pero a su vez como en el caso anterior mayor coste tendrá para el servidor y podría afectar a su rendimiento. A menor tamaño menos concurrencia, menor coste de recursos, pero menor concurrencia y posiblemente haya clientes en espera.

Pool Resize Quantity: número de conexiones que se eliminan cuando expira el tiempo “Idle Timeout”, siendo 2 en nuestro sistema. Si aumentamos mucho el “pool resize quantity” podemos cerrar conexiones que aún sean útiles, y si lo bajamos mucho podemos mantener vigentes conexiones que no están siendo utilizadas.

Idle Timeout: es el tiempo máximo que una conexión puede permanecer inactiva. En nuestro sistema es de 300 segundos. Si ponemos un tiempo muy bajo, podremos terminar conexiones muchas conexiones que no debemos que no han acabado o que son más lentas. Si lo ponemos muy alto estaremos manteniendo a su vez muchas conexiones inactivas que podrían ser eliminadas, y teniendo que crear nuevos hilos en el pool para gestionar las nuevas, en vez de asignar los hilos inactivos.

Max Wait Time: tiempo que espera un cliente hasta que recibe un timeout de la conexión. En el ejemplo es de 1 min. Si es muy alto, los clientes podrían estar esperando mucho tiempo. Mientras que si es muy bajo entonces el cliente podría abandonar la conexión muy pronto.

EJERCICIO 4:

Este es el fragmento de código SQL que se encarga de comprobar que una tarjeta se encuentra en la Base de Datos de la aplicación, buscando en la tabla correspondiente una tarjeta que concuerde con todos los datos del formulario.

```
String getQryInsertPago(PagoBean pago) {
    String qry = "insert into pago("
                + "idTransaccion,"
                + "importe,idComercio,"
                + "numeroTarjeta)"
                + " values ("
                + "'" + pago.getIdTransaccion() + "',"
                + pago.getImporte() + ","
                + "'" + pago.getIdComercio() + "',"
                + "'" + pago.getTarjeta().getNumero() + "'"
                + ")";
    return qry;
}
```

Y este es el fragmento que realiza la ejecución del pago añadiendo a la tabla pago de la Base de Datos junto con los datos requeridos de la transacción.

```
/**
 *  getQryCompruebaTarjeta
 */
String getQryCompruebaTarjeta(TarjetaBean tarjeta) {
    String qry = "select * from tarjeta "
                + "where numeroTarjeta='" + tarjeta.getNumero()
                + "' and titular='" + tarjeta.getTitular()
                + "' and validaDesde='" + tarjeta.getFechaEmision()
                + "' and validaHasta='" + tarjeta.getFechaCaducidad()
                + "' and codigoVerificacion='" + tarjeta.getCodigoVerificacion() + "'";
    return qry;
}

/**
```

EJERCICIO 5:

```
/**
 * Imprime traza de depuracion
 */
public void errorLog(String error) {
    if (isDebug())
        System.err.println("[directConnection=" + this.isDirectConnection() +"] " +
                           error);
}
```

El errorLog se comprueba en los siguientes puntos: en la query que devuelve la tarjeta, para ver que no ha dado ningún error este “select”, también a la hora de realizar el pago e insertarlo en la tabla Pago, comprueba que no ha habido ningún error en el “insert”, en ambos casos además se comprueba que no haya error en las querys cuando está seleccionada la opción de Use Prepared.
Por otro lado también comprueba los casos en los que se hace un “select” de los pagos ya realizados cuando se quiere comprobar el historial de pagos así como cuando se quiere eliminar un pago. Por último, también se hace log de las excepciones en caso de error (como puede ser por ejemplo una PSQLException si metemos mal los parámetros como el usuario, IP o puerto de la base de datos).

Si accedemos a los logs del servidor se puede apreciar que son de tipo “SEVERE” los log que dan más información con la opción debug. Si hacemos un primer pago incorrecto con el debug en false podemos ver sólo un log correspondiente al pago incorrecto, pero si volvemos a hacer la misma consulta incorrecta con el debug a true podemos ver en los logs las consultas realizadas y el error en este caso, que es que la clave del identificador de la compra está duplicado.

Log Viewer Results (40)

Records before 1332 | Log File Record Numbers 1332 through 1385 | Records after 1385

Record Number	Log Level	Message	Logger	Timestamp	Name-Value Pairs
1385	INFO	WebModule[null] ServletContext.log():[ERROR] :Pago incorrecto(details)	javax.enterprise.web	06-mar-2021 22:25:47.516	{levelValue=800, timeMillis=1615098347516}
1384	SEVERE	[directConnection=false] org.postgresql.util.PSQLException: ERROR: duplicate key value violates uniq... (details)		06-mar-2021 22:25:47.514	{levelValue=1000, timeMillis=1615098347514}
1383	SEVERE	[directConnection=false] insert into pago(idTransaccion,importe,idComercio,numeroTarjeta) values ('9... (details)		06-mar-2021 22:25:47.512	{levelValue=1000, timeMillis=1615098347512}
1382	SEVERE	[directConnection=false] select * from tarjeta where numeroTarjeta='2347 4840 5058 7931' and titular... (details)		06-mar-2021 22:25:47.509	{levelValue=1000, timeMillis=1615098347509}
1381	INFO	WebModule[null] ServletContext.log():[INFO] Acceso correcto:/procesapago(details)	javax.enterprise.web	06-mar-2021 22:25:47.445	{levelValue=800, timeMillis=1615098347445}
1380	INFO	WebModule[null] ServletContext.log():[ERROR] :Pago incorrecto(details)	javax.enterprise.web	06-mar-2021 22:25:20.223	{levelValue=800, timeMillis=1615098320223}
1379	INFO	WebModule[null] ServletContext.log():[INFO] Acceso correcto:/procesapago(details)	javax.enterprise.web	06-mar-2021 22:25:20.169	{levelValue=800, timeMillis=1615098320169}

EJERCICIO 6:

Para desplegar el servicio web con JAX-WS descargamos el archivo P1-WS.jar de moodle y extraemos el contenido. Dentro de la carpeta extraída pegamos los directorios sql, web y datagen del directorio P1-base. también debemos pegar los archivos postgresql.properties (con cliente en 10.1.4.2 y host en 10.1.4.1) y postgres.xml . Aunque el fichero build.properties lo modificamos para tener en la ip 10.1.4.1 el servidor y en la ip 10.1.4.2 el cliente.

```
as.host.client=10.1.4.2
as.host.server=10.1.4.1
```

Después de organizar todos los ficheros tal y como nos indican en la práctica nos queda modificar el fichero VisaDAOWS.java.

Lo primero que hacemos es hacer import de WebService, WebMethod y WebParam.

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

Lo siguiente es cambiar el nombre de la clase y del constructor de VisaDAO a VisaDAOWS. Y ponemos la etiqueta @WebService encima de la clase para indicar que será un servicio web.

```
*/
@WebService()
public class VisaDAOWS extends DBTester {
```

Lo siguiente es poner la etiqueta @WebMethod(operationName=”nombreMetodo”) encima de las funciones que queremos publicar como métodos web. También será necesario etiquetar los parámetros de estos métodos si los tuvieran con la etiqueta @WebParam(name=”nombreArgumento”)

A continuación se muestra para los métodos compruebaTarjeta(), isDebug(), setDebug() del cual se ha eliminado la segunda función ya que no se usará y no deben haber duplicados, isPrepared() y setPrepared().

```
@WebMethod(operationName="compruebaTarjeta")
public boolean compruebaTarjeta(@WebParam(name="tarjeta") TarjetaBean tarjeta) {
```

```
@WebMethod(operationName="isPrepared")
public boolean isPrepared() {
    return prepared;
}

@WebMethod(operationName="setPrepared")
public void setPrepared(@WebParam(name="prepared")boolean prepared) {
    this.prepared = prepared;
}
/*****************************************************/

/**
 * @return the debug
 */
@WebMethod(operationName = "isDebug")
public boolean isDebug() {
    return debug;
}

/**
 * @param debug the debug to set
 */
@WebMethod(operationName = "setDebug")
public void setDebug(@WebParam(name="debug")boolean debug) {
    this.debug = debug;
}
```

En el caso de la función realizaPago() también le hemos añadido las etiquetas anteriores de @WebParam y @WebMethod. Además hemos cambiado el retorno para que devuelva el PagoBean en vez de un boolean. Simplemente es cambiar el tipo de la función y de la variable ret dentro de la función por PagoBean. Y donde antes ponía “ret = false” cambiarlo por “ret = null” y donde ponía “ret

= true” cambiarlo por “ret = pago”. No se adjunta captura ya que ocupa demasiado y se adjunta el código en la entrega.

Para las dos funciones isDirectConnection() y setDirectConnection() al ser VisaDAOWS una clase que hereda de DBTester podemos definir otra vez los métodos en VisaDAOWS y hacer un override de ellos y así añadimos las etiquetas @WebMethod y @WebParam.

```
    @Override
    @WebMethod(operationName="isDirectConnection")
    public boolean isDirectConnection(){
        return super.isDirectConnection();
    }

    @Override
    @WebMethod(operationName="setDirectConnection")
    public void setDirectConnection(@WebParam(name="directConnection") boolean directConnection) {
        super.setDirectConnection(directConnection);
    }
}
```

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

Se ha de alterar ya que el servlet ProcesaPago utiliza una instancia de un PagoBean que se genera en ComienzaPago. Pero ahora los servlets están desacoplados al acceso de los datos (se encuentran en otro nodo cliente) y realizan su iteración a través del servicio web. De esta manera para obtener el PagoBean correspondiente tendrá que recibir la instancia del servicio web, por lo al final necesitaremos que realizaPago() devuelva un PagoBean.

EJERCICIO 7:

Para compilar el servicio web del servidor solo es necesario abrir una terminal en el directorio P1-ws y ejecutar los siguientes comandos en orden:
ant compilar-servicio
ant empaquetar-servicio
ant desplegar-servicio

Si accedemos a la página http://10.1.4.1:4848 e iniciamos sesión ahora con el admin, en el panel izquierdo tendremos en la pestaña de Aplicaciones el servicio web llamado P1-ws-ws. Si accedemos a él y pinchamos en “View Endpoint” podremos acceder a la URL del archivo WDSL.

A través del enlace http://10.1.4.1:8080/P1-ws-ws/VisaDAOWSService?wsdl accedemos al fichero WDSL.

1. ¿En qué fichero están definidos los tipos de datos intercambiados con el webservice?

   Los tipos de datos se encuentran definidos dentro de la etiqueta <types>, en particular se encuentra en la url del fichero
   http://10.1.4.1:8080/P1-ws-ws/VisaDAOWSService?xsd=1

2. ¿Qué tipos de datos predefinidos se usan?

   Si vamos al fichero anterior vemos que se usan los tipos xs:boolean, xs:int y xs:string.
3. ¿Cuáles son los tipos de datos que se definen?

   Se definen las clases que utilizamos como parámetros o retornos y los métodos del web service y se definen con <complexType>. En el caso de las clases se puede apreciar pagoBean, tarjetaBean.

4. ¿Qué etiqueta está asociada a los métodos invocados en el webservice?

   Está asociada la etiqueta <operation> seguida del nombre del método. Dentro de la etiqueta se definen los mensajes de input del servicio y los mensajes de output del servicio

5. ¿Qué etiqueta describe los mensajes intercambiados en la invocación de los métodos del webservice?

   La etiqueta <message> que describe el nombre del mensaje y los elementos que contiene de parámetros.

6. ¿En qué etiqueta se especifica el protocolo de comunicación con el webservice?

   Se especifica dentro de la etiqueta <binding>, en nuestro caso si nos fijamos en el atributo “transport” vemos que estamos usando HTTP.

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

7. ¿En qué etiqueta se especifica la URL a la que se deberá conectar un cliente para acceder al webservice?

   Se especifica dentro de la etiqueta <service>.

```
<service name="VisaDAOWSService">
<port name="VisaDAOWSPort" binding="tns:VisaDAOWSPortBinding">
          <soap:address
          location="http://10.1.4.1:8080/P1-ws-ws/VisaDAOWSService"/>
</port>
</service>
```

   Siendo la URL la que se encuentra en <soap:address location>.

EJERCICIO 8:

Hemos cambiado la instanciación de la clase remota, ya que ahora el cliente no debe tener ninguna instancia de la clase VisaDAOWS, de modo que:

```
// VisaDAO dao = new VisaDAO();
VisaDAOWSService service = new VisaDAOWSService();
VisaDAOWS dao = service.getVisaDAOWSPort ();
```

Por otro lado como ya cambiamos el retorno de la función realizaPago, tenemos que llevar a cabo una pequeña modificación para que se adapte al nuevo retorno “null”:

```
if (dao.realizaPago(pago) == null) {
        enviaError(new Exception("Pago inc
        return;
```

Por último, añadiremos las excepciones que pueden generar las llamadas a clases remotas de forma que puedan controlarse:

```
VisaDAOWSService service = null;
VisaDAOWS dao = null;

try {
    service = new VisaDAOWSService();
    dao = service.getVisaDAOWSPort ();

}catch(Exception e){
    enviaError(e, request, response)
    return;
}
```

EJERCICIO 9:

Lo primero que haremos en este ejercicio será modificar el web.xml para poder hacer que la llamada al servicio remoto se lleve a cabo desde el este fichero de tal forma que añadiremos en el context la ruta del servicio de la siguiente forma:

```
<context-param>
    <param-name>VisaWSPath</param-name>
    <param-value>http://10.1.4.1:8080/P1-ws-ws/VisaDAOWSService</param-value>
</context-param>
```

Una vez tengamos configurado de la siguiente manera este fichero modificaremos por último el ProcesaPago para que realice la conexión a traves de este archivo, los nuevos cambios se añadirán dentro del apartado de excepciones que habíamos ya creado con anterioridad:

```
VisaDAOWSService service = null;
VisaDAOWS dao = null;
BindingProvider bp = null;

try {
    service = new VisaDAOWSService();
    dao = service.getVisaDAOWSPort ();

    bp = (BindingProvider) dao;
    bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, getServletContext().getInitParameter("VisaWSPath"));
}catch(Exception e){
    enviaError(e, request, response)
    return;
}
```

EJERCICIO 10:

En este ejercicio modificaremos los ficheros de GetPagos y DelPagos, añadiendo la misma estructura para la conexión remota y directa con el servicio de la misma manera que en el ejercicio anterior, además como se nos indica en este ejercicio se deberá modificar la declaración del método getPagos() de forma que devolverá ahora un array de PagoBean, posteriormente en el fichero GetPagos() se incluirá un toArray para castear lo recibido de esta función del modo siguiente:

```
/* Petici&oacute;n de los pagos para el comercio */
PagoBean[] pagos = dao.getPagos(idComercio).toArray(new PagoBean[0]);
```

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

Además en la función getPagos del fichero VisaDAOW también hemos tenido que realizar una serie de cambios para devolver un ArrayList y no un boolean:

```
            //ret = new PagoBean[pagos.size()];
            //ret = pagos.toArray(ret);

            // Cerramos / devolvemos la conexion al pool
            pcon.close();

        } catch (Exception e) {
            errorLog(e.toString());

        } finally {
            try {
                if (rs != null) {
                    rs.close(); rs = null;
                }
                if (pstmt != null) {
                    pstmt.close(); pstmt = null;
                }
                if (pcon != null) {
                    closeConnection(pcon); pcon = null;
                }
            } catch (SQLException e) {
            }
        }

        //return ret;
        return pagos;
    }
```

EJERCICIO 11:

El comando para generar los stubs ha sido el siguiente:

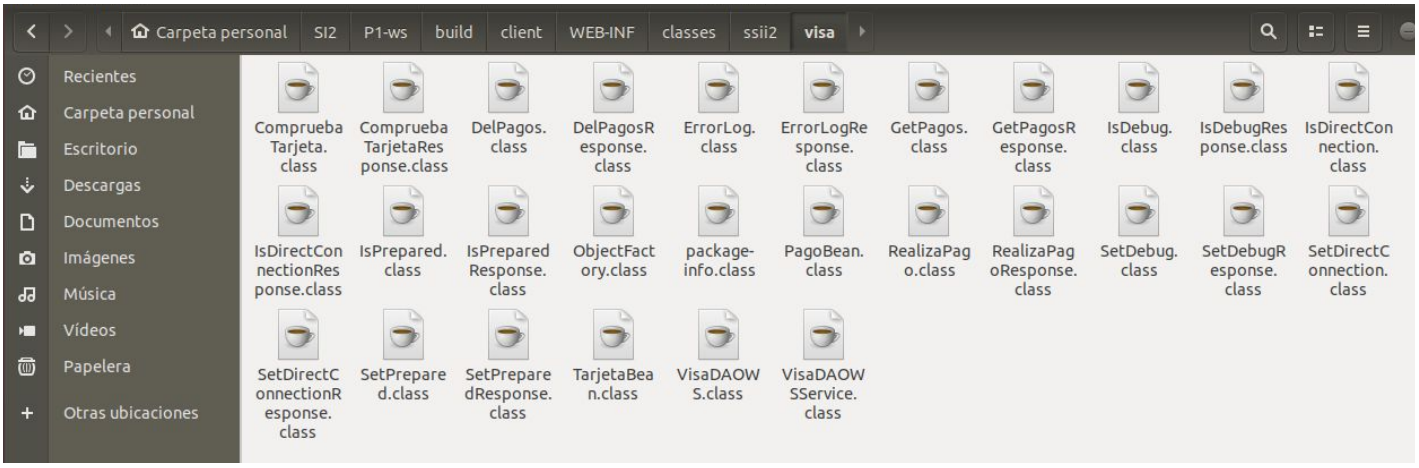
wsimport -d build/client/WEB-INF/classes -p ssii2.visa http://10.1.4.1:8080/P1-ws-ws/VisaDAOWSService?wsdl

```
francisco@francisco:~/SI2/P1-ws$ wsimport -d build/client/WEB-INF/classes -p ssii2.visa http://10.1.4.1:8080/P1-ws-ws/VisaDAOWSService?wsdl
analizando WSDL...


Generando código...


Compilando código...
```

Las clases que se han generado han sido las siguientes:



Vemos que se generan dos clases para cada método del cliente: uno para el input de los datos y otro para la respuesta del servicio con los datos. Después el web service que es VisaDAOW tiene su clase y la de su servicio. Por último disponemos de las clases de TarjetaBean y Pagobean.

EJERCICIO 12:

Simplemente es poner el comando del ejercicio anterior pero dentro de build.xml utilizando las variables ${build.client}, ${paquete} y ${wsdl.url}, utilizando la funcionalidad ant exec.

Hemos añadido al fichero build.xml lo siguiente:

```
<exec executable="${wsimport}">
  <arg line=" -d ${build.client}/WEB-INF/classes"/>
  <arg line=" -p ${paquete}.visa"/>
  <arg line=" ${wsdl.url}"/>
</exec>
```

EJERCICIO 13:

El fichero build.properties lo modificamos para tener en la ip 10.1.4.1 el servidor y en la ip 10.1.4.2 el cliente. El fichero postgresql.properties habrá que cambiar también el cliente por 10.1.4.2.

```
as.host.client=10.1.4.2
as.host.server=10.1.4.1
```

Una vez desplegado el cliente con ant generar-stubs, ant compilar-cliente. ant empaquetar-cliente, ant desplegar-cliente y con el servidor ya desplegado podemos acceder a http://10.1.4.2:8080/P1-ws-cliente/testdb.jsp

Rellenamos el pago:

El pago es correcto:

Podemos usar la página del test para comprobar que existe un pago, borrar y comprobar despues que ya no existe (toda esta funcionalidad implica consultas sql que ya se han mostrado en el ejercicio 4).

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.

Se empieza desde paho.html que te pedirá que rellenes un formulario con id de la transacción, el id del comercio y el importe. Esta información la recibe “Comienza Pago” que es el servlet encargado de comprobar que los datos del anterior formulario son correctos. Si son correctos como en este caso nos envía la JSP formdatosvisa.jsp para que rellenes el formulario con los datos de la tarjeta. Estos

datos son recibidos por el servlet “Procesa Pago” que comprobará que la tarjeta es correcta pero que ha caducado y nos redireccionará a la JSP error/muestraerror.jsp que nos mostrará el error adecuado.

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?

El servlet Comienza Pago se encarga de comprobar los datos de la transacción y el comercio y solicita los datos de la tarjeta. Mientras que el servlet Procesa Pago se encarga de comprobar los datos de la tarjeta.

Cuestión 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?

Comienza Pago solicita el idTransacción, idComercio y el importe.
Procesa Pago solicita el número de la tarjeta , titular, fecha emisión, fecha caducidad y cvv2.
La información la comparten gracias a una HTTPSession del usuario, y el servidor enviará una cookie al cliente que almacenará el navegador para que pueda acceder a la información del Hashmap de la sesión que se almacena en el servidor.

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?

Si se usa pago.html primero accederemos al servlet Comienza Pago y después a Procesa Pago. En cambio si se usa testdb.jsp accederemos al servlet Procesa Pago únicamente.

Funciona bien el testdb.jsp ya que la propia página nos pide toda la información necesaria para realizar un pago de una, mientras que en pago.html primero nos pedían los datos del comercio, transacción e importe y después los datos de la tarjeta. Mientras que en el test podemos pasar todos estos datos a la vez, además de otras opciones de debug, direct y prepared. Además Procesa Pago es el encargado de comprobar los datos de la tarjeta, comprobar errores y de realizar e insertar el pago. De esta manera el test al solo acceder a este último servlet podrá realizar pagos de manera correcta.