		Escuela Politécnica Superior Ingeniería Informática Prácticas de Sistemas Informáticos 2			
Grupo	**2401**	**Práctica**	4	**Fecha**	28/02/2017
Alumno/a		Hernando, Bernabé, Amanda			
Alumno/a		Pérez, Manso, Pablo			

Práctica 1A: Arquitectura de JAVA EE

Ejercicio número 1:

- 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.

Los ficheros modificados, junto a los cambios realizados para un correcto despliegue son los siguientes:

- build.properties
  - as.host => as.host=10.1.4.2
- postgresql.properties
  - db.host => db.host=10.1.4.2
  - db.client.host => db.client.host=10.1.4.2

- Realice un pago contra la aplicación web empleando el navegador en la ruta http://10.1.4.2:8080/P1. Conéctese a la base de datos y obtenga evidencias de que el pago se ha realizado.

1. Creamos una transacción:

2. Añadimos una tarjeta a la transacción (que previamente hemos insertado en la base de datos):

3. Comprobamos en la base de datos que se ha añadido correctamente aunque previamente nos haya salido el mensaje de éxito.

	idautorizacion [PK] serial	idtransaccion character(16)	codrespuesta character(3)	importe double precision	idcomercio character(16)	numerotarjeta character(19)	fecha timestamp without time zone
1	2	1	000	1	1	1111 1111 1111 1111	2017-02-15 09:23:13.70593
*							

-Acceda a la página de pruebas extendida, http://10.1.4.2:8080/P1/testbd.jsp. Compruebe que la funcionalidad de listado y borrado de pagos funciona correctamente. Elimine el pago anterior.

1. Listamos los pagos del comercio 1:

Pago con tarjeta

Lista de pagos del comercio 1

idTransaccion	Importe	codRespuesta	idAutorizacion
1	1.0	000	2

Volver al comercio

Prácticas de Sistemas Informáticos II

2. Borramos los pagos del comercio 1:

10.1.4.2:8080/P1/delpagos

Pago con tarjeta

Se han borrado 1 pagos correctamente para el comercio 1

Volver al comercio

Prácticas de Sistemas Informáticos II

Ejercicio número 2:

Se pide completar la información necesaria en la clase VisaDAO para llevar a cabo la conexión directa de forma correcta.

Para realizar de forma correcta la conexión directa en la clase VisaDAO, hemos fijado los siguientes atributos de dicha clase:

1. Nombre del driver JDBC:

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

2. JDBC connection string correspondiente con el servidor posgresql:

   ```
   private static final String JDBC_CONNSTRING =  "jdbc:postgresql://10.1.4.1:5432/visa";
   ```

3. Nombre de usuario:

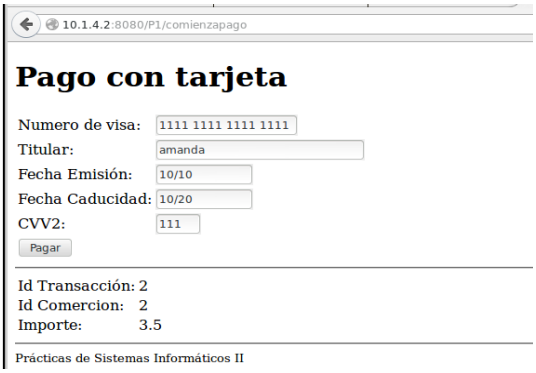
   ```
   private static final String JDBC_USER = "alumnodb";
   ```

4. Contraseña:

   ```
   private static final String JDBC_PASSWORD = "alumnodb";
   ```

Una vez completada la información, acceda a la página de pruebas extendida, http://10.1.4.2:8080/P1/testbd.jsp y pruebe a realizar un pago utilizando la conexión directa y pruebe a listarlo y eliminarlo.

1. Realizamos un pago.



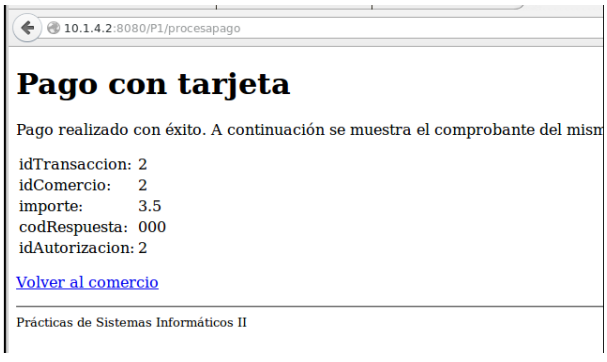
10.1.4.2:8080/P1/comienzapago

Pago con tarjeta

Numero de visa: 1111 1111 1111 1111
Titular: amanda
Fecha Emisión: 10/10
Fecha Caducidad: 10/20
CVV2: 111
Pagar

Id Transacción: 2
Id Comercion: 2
Importe: 3.5

Prácticas de Sistemas Informáticos II



10.1.4.2:8080/P1/procesapago

Pago con tarjeta

Pago realizado con éxito. A continuación se muestra el comprobante del mism

idTransaccion: 2
idComercio: 2
importe: 3.5
codRespuesta: 000
idAutorizacion: 2

Volver al comercio

Prácticas de Sistemas Informáticos II

2 . Comprobamos en la base de datos que se ha llevado a cabo correctamente.

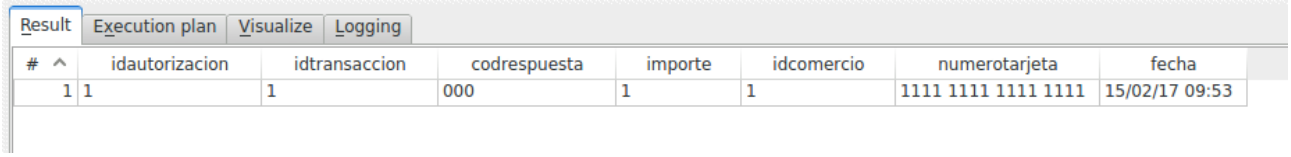
Result | Execution plan | Visualize | Logging

#	idautorizacion	idtransaccion	codrespuesta	importe	idcomercio	numerotarjeta	fecha
1	1	1	000	1	1	1111 1111 1111 1111	15/02/17 09:53
2	2	2	000	3,5	2	1111 1111 1111 1111	15/02/17 09:56

3. Listamos los pagos del comercio 2 y como era de esperar, sólo aparece uno.

3. Borramos los pagos del comercio 2.

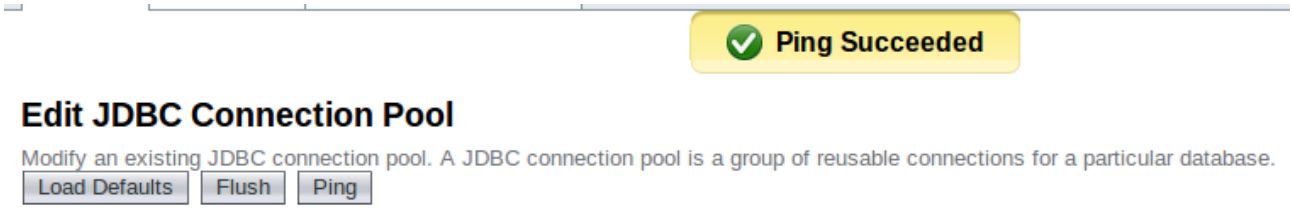
4 Comprobamos que la operación se ha realizado con éxito.



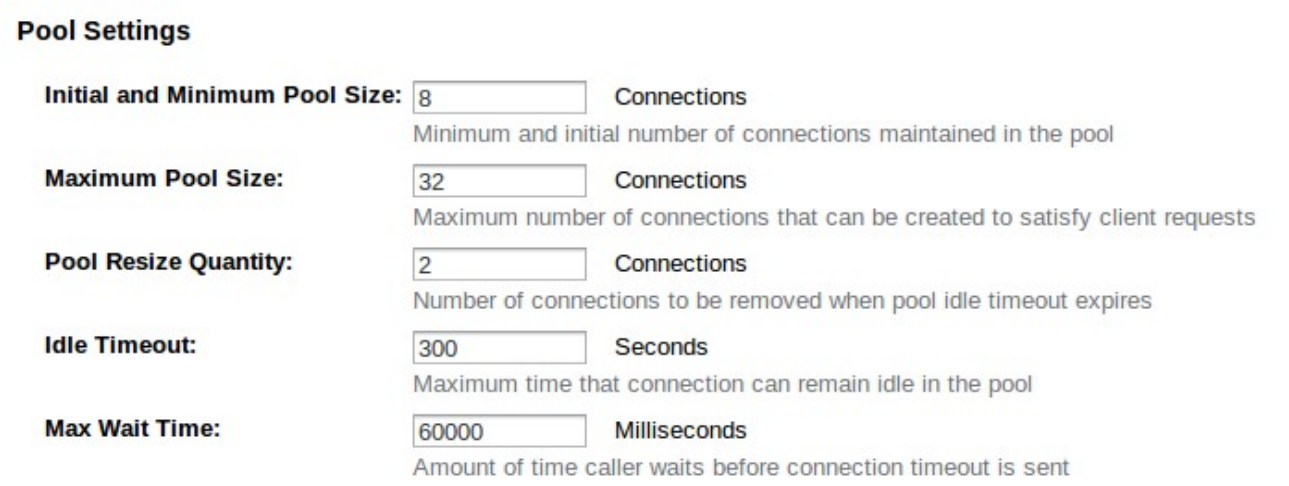
Ejercicio número 3:

Examinar el archivo postgresql.properties para determinar el nombre del recurso JDBC correspondiente al DataSource y el nombre del pool.
Nombre del pool: `db.pool.name=VisaPool`
Nombre de recurso JDBC: `db.jdbc.resource.name=jdbc/VisaDB`

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.

Se denomina *connection pool* al manejo de una colección de conexiones abiertas a una base de datos de manera que puedan ser reutilizadas al realizar múltiples consultas o actualizaciones.

Al mantenerse abierto un grupo de conexiones, éstas son atribuidas a los diferentes hilos de ejecución únicamente el tiempo de una transacción con la base de datos. Al finalizar su utilización, la conexión se pone a disposición de otro hilo de ejecución que necesite de ese recurso, en lugar de cerrarla o de asignarla permanentemente a un único hilo de ejecución.

Según las políticas de agrupamiento de conexiones, cuando todas están en uso se establecen nuevas conexiones, y si ello no es posible (porque se ha llegado al *maximum pool size*), se deja el hilo de ejecución en espera de la liberación de alguna conexión. A la inversa, si pasa mucho tiempo (i*dle timeout*) sin que se utilicen las conexiones, algunas de ellas o todas podrían ser cerradas.

Como consecuencia, si el tamaño máximo del pool es alto se pueden realizar más conexiones simultáneas a la base de datos y las solicitudes de conexión pasan menos tiempo en la cola. Sin embargo, el acceso a la tabla de conexiones es más lento. Por tanto, la aplicación será más eficiente cuanto mayor sea el tamaño del pool siempre que las solicitudes sean suficientemente complejas como para que el coste de acceso a la tabla no sea significativo.

Por otro lado, para mejorar el rendimiento habría que ajustar el tiempo de espera máximo a cero. Esto básicamente bloquea el hilo del llamante hasta que una conexión esté disponible, lo que permite al servidor aliviar la tarea de rastrear el tiempo de espera transcurrido para cada solicitud.

En lo relativo al *idle timeout*, habría que mantener este tiempo de espera más corto que el tiempo de espera del servidor de base de datos para evitar la acumulación de conexiones inutilizables. Además, para obtener mayor rendimiento, habría que establezcerlo a cero segundos, de modo que no se eliminen las conexiones inactivas. Esto garantiza que normalmente no haya penalidad al crear nuevas conexiones y deshabilita el hilo que controla las conexiones inactivas. Sin embargo, existe el riesgo de que el servidor de base de datos restablezca una conexión que no se utiliza durante demasiado tiempo.

Ejercicio número 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.

```
 private static final String SELECT_TARJETA_QRY =
                   "select * from tarjeta " +
                   "where numeroTarjeta=? " +
                   " and titular=? " +
                   " and validaDesde=? " +
                   " and validaHasta=? " +
                   " and codigoVerificacion=? ";
```

- Ejecución del pago.

```
 private static final String INSERT_PAGOS_QRY =
                    "insert into pago(" +
                    "idTransaccion,importe,idComercio,numeroTarjeta)" +
                    " values (?,?,?,?)";
```

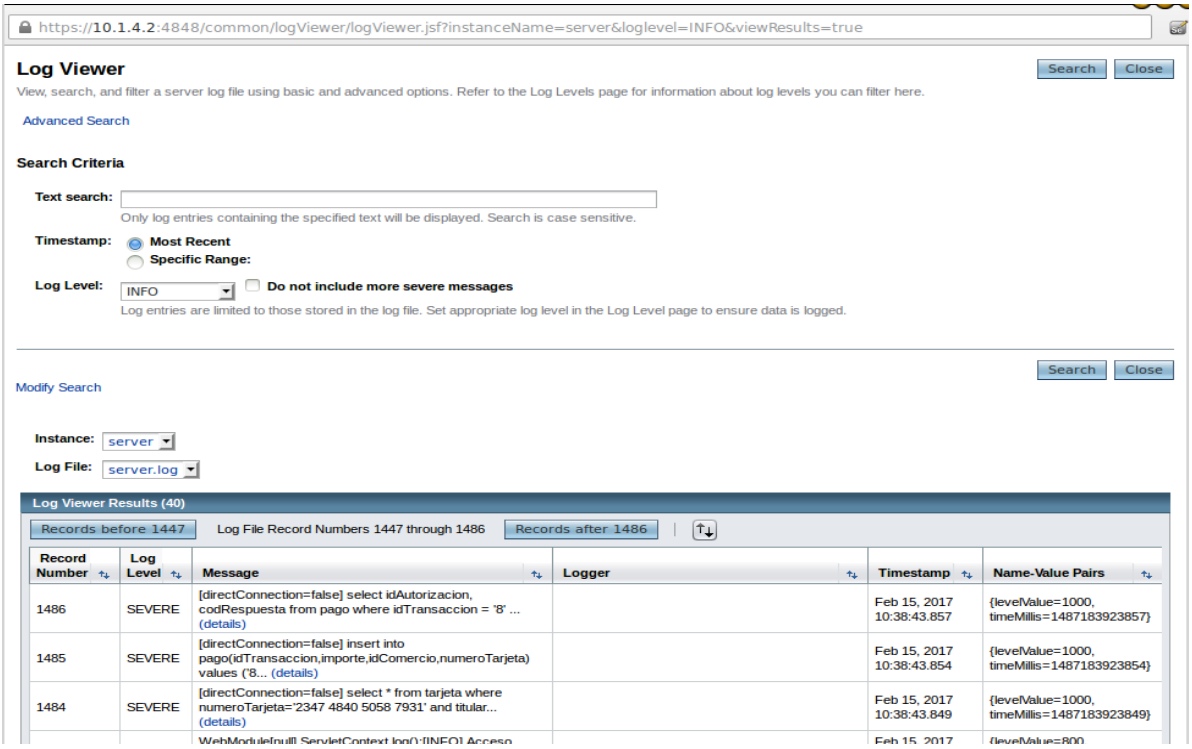
Ejercicio número 5:

Edite el fichero VisaDAO.java y localice el método errorLog. Compruebe en qué partes del código se escribe en log utilizando dicho método.

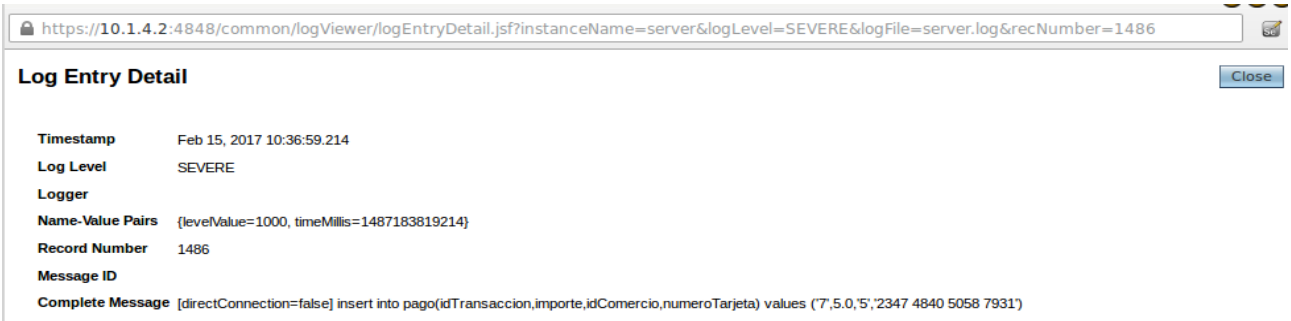
La rutina *errorLog* aparece justo antes de ejecutar una consulta, en el caso de que esté preparada, antes incuso de asociar los parámetros. En estas ocasiones se escribe la consulta a realizar. Además aparece en el caso en que salte una excepción, mostrando el error ocurrido.

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.

Log del servidor:



Si seleccionamos las primeras entradas, observamos la consulta realizada con los valores que esperábamos. A continuación mostramos la consulta relativa a la tarjeta.



Ejercicio número 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)**
- **isPrepared() / setPrepared()**

En 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).

Los cambios se encuentran en el código proporcionado. Básicamente se reducen en introducir antes del método que va a ser publicado *@WebMethod* y antes de sus parámetros *@WebParam* como a continuación:

```
@WebMethod(operationName ="realizaPago")
   public synchronized PagoBean realizaPago(@WebParam(name="pago") PagoBean pago){...}
```

Además la clase hay que declararla como servicio web:

```
@WebService()
public class VisaDAOWS extends 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.**

Los cambios se encuentran en el código, pero simplemente hay que cambiar el prototipo de la función y devolver el pago o null dependiendo del resultado de la operación.

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?

Ahora, en la implementación de este servicio web, la interfaz cliente (los servlets y JSP), es independiente del servicio web y está desacoplado del acceso a los datos. Por tanto se necesita la devolución del pago, para comprobar la correcta realización mediante el código de respuesta y para saber el identificador de autorización.

Ejercicio número 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.

```
−<definitions targetNamespace="http://dao.visa.ssii2/" name="VisaDAOWSService">
 +<types></types>
 −<message name="realizaPago">
    <part name="parameters" element="tns:realizaPago"/>
  </message>
 +<message name="realizaPagoResponse"></message>
 +<message name="isPrepared"></message>
 +<message name="isPreparedResponse"></message>
 +<message name="setPrepared"></message>
 +<message name="setPreparedResponse"></message>
 +<message name="isDirectConnection"></message>
 +<message name="isDirectConnectionResponse"></message>
 +<message name="setDirectConnection"></message>
 +<message name="setDirectConnectionResponse"></message>
 +<message name="compruebaTarjeta"></message>
 +<message name="compruebaTarjetaResponse"></message>
 +<message name="isDebug"></message>
 +<message name="isDebugResponse"></message>
 +<message name="setDebug"></message>
 +<message name="setDebugResponse"></message>
 +<portType name="VisaDAOWS"></portType>
 +<binding name="VisaDAOWSPortBinding" type="tns:VisaDAOWS"></binding>
 +<service name="VisaDAOWSService"></service>
</definitions>
```

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.

WSDL (Web Services Description Language) es una gramática XML que se utiliza para describir la interfaz pública de los servicios Web de forma estandarizada. Un documento WSDL describe tres propiedades fundamentales de un servicio web: las operaciones soportadas y qué mensajes las activan, el protocolo de comunicación en el que se envía el mensaje y por último, la forma en qué cada operación se compone de mensajes formateados de una forma específica y transmitidos por un protocolo concreto de red.

Conteste a las siguientes preguntas:

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

En el fichero XSD referenciado en el WSDL.

•¿Qué tipos de datos predefinidos se usan?

Se usan xsd:string, xsd:double y xsd:boolean.

•¿Cuáles son los tipos de datos que se definen?

Se define un elemento por cada método del servicio y un elemento por cada respuesta del método. Además, de crear un elemento por cada tipo de dato (estructuras) a utilizar. Resumiendo, un elemento por cada clase a utilizar en la comunicación.

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

<operation>

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

<message>

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

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

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

```
<service name="VisaDAOWSService">
  <port name="VisaDAOWSPort" binding="tns:VisaDAOWSPortBinding">
    <soap:address location="http://10.1.4.2:8080/P1-ws-ws/VisaDAOWSService"/>
  </port>
</service>
```

Ejercicio número 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.

De esta forma eliminaremos la línea que dice

```
return ret;
```

Y en su lugar añadiremos los siguiente

```
// Si la respuesta del servidor es errónea, devolvemos null
if (ret == false)
    return null;

// Sino devolvemos el pago
return pago;
```

• Las llamadas remotas pueden generar nuevas excepciones que deberán ser tratadas en el código cliente.

De esta forma instanciaremos el WebService:

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

try{
  bp.getRequestContext().put(
                          BindingProvider.ENDPOINT_ADDRESS_PROPERTY, “http:...”);
} catch (Exception e){
    enviaError(new Exception("Servidor no encontrado"), request, response);
    return;
}
```

Ejercicio número 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.

De esta forma en ProcesaPago.java cogerá la url de la siguiente manera:

```
        String url=getServletContext().getInitParameter("url-server");

        try{
            bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, url);
        } catch (Exception e){
            enviaError(new Exception("Servidor no encontrado"), request, response);
            return;
        }
```

Y al fichero Web.xml añadimos las siguientes líneas:

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

Ejercicio número 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.

Se han modificado los metodos de processRequest de delPagos y de getPagos de forma que ambos incluyan la llamada al webservice, pero además para que cojan la URL del fichero de configuración como ya hicimos en el ejercicio anterior.

DelPagos.java

```
VisaDAOWSService service = new VisaDAOWSService();
VisaDAOWS dao = service.getVisaDAOWSPort();
BindingProvider bp = (BindingProvider) dao;
String url=getServletContext().getInitParameter("url-server");

try{
    bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, url);
} catch (Exception e){
    enviaError(new Exception("Servidor no encontrado"), request, response);
    return;
}

/* Se recoge de la petici&oacute;n el par&aacute;metro idComercio*/
String idComercio = request.getParameter(PARAM_ID_COMERCIO);

/* Petici&oacute;n de los pagos para el comercio */
int ret = dao.delPagos(idComercio);
```

```
if (ret != 0) {
       request.setAttribute(ATTR_BORRADOS, ret);
       reenvia("/borradook.jsp", request, response);
}
else {
       reenvia("/borradoerror.jsp", request, response);
}
return;
```

GetPagos.java

```
VisaDAOWSService service = new VisaDAOWSService();
VisaDAOWS dao = service.getVisaDAOWSPort();
BindingProvider bp = (BindingProvider) dao;

String url=getServletContext().getInitParameter("url-server");

try{
    bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, url);
} catch (Exception e){
    enviaError(new Exception("Servidor no encontrado"), request, response);
    return;
}

/* Se recoge de la petici&oacute;n el par&aacute;metro idComercio*/
String idComercio = request.getParameter(PARAM_ID_COMERCIO);

/* Petici&oacute;n de los pagos para el comercio */
ArrayList<PagoBean> pagosArraylist = dao.getPagos(idComercio);

PagoBean [] pagos = pagosArraylist.toArray(new PagoBean[pagosArraylist.size()]);

request.setAttribute(ATTR_PAGOS, pagos);
reenvia("/listapagos.jsp", request, response);
Return;
```

Además en el caso de este último hace falta convertir la respuesta del WebService, un ArrayList a el array que la vista espera recibir para mostrar

De este mismo modo ha habido que modifcar el método del WebService para que devuelva ese ArrayList. En este caso simplemente había que borrar las líneas que convertían el ArrayList que ya hacíamos antes en Array y devolver ese mismo ArrayList.

```
pagos = new ArrayList<PagoBean>();
while (rs.next()) {
    TarjetaBean t = new TarjetaBean();
    PagoBean p = new PagoBean();
    p.setIdTransaccion(rs.getString("idTransaccion"));
    p.setIdComercio(rs.getString("idComercio"));
    p.setImporte(rs.getFloat("importe"));
    t.setNumero(rs.getString("numeroTarjeta"));
    p.setTarjeta(t);
    p.setCodRespuesta(rs.getString("codRespuesta"));
    p.setIdAutorizacion(String.valueOf(rs.getInt("idAutorizacion")));

    pagos.add(p);
}
return pagos;
```

Ejercicio número 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.

El comando a ejcutar es el siguiente:
`wsimport -d build/client/WEB-INF/classes -p ssii2.visa http://10.1.4.2:8080/P1-ws-ws/VisaDAOWSService?wsdl`

Las clases generadas son las siguientes:

- CompruebaTarjeta.class
- CompruebaTarjetaResponse.class
- IsDebug.class
- IsDebugResponse.class
- IsDirectConnection.class
- IsDirectConnectionResponse.class
- IsPrepared.class
- IsPreparedResponse.class
- ObjectFactory.class
- package-info.class
- PagoBean.class
- RealizaPago.class
- RealizaPagoResponse.class
- SetDebug.class
- SetDebugResponse.class
- SetDirectConnection.class
- SetDirectConnectionResponse.class
- SetPrepared.class
- SetPreparedResponse.class
- TarjetaBean.class
- VisaDAOWS.class
- VisaDAOWSService.class

Como se observa se generan un fichero .class por cada clase: VisaDAOWS, PagoBean, TarjetaBean… Además se crearán dos clases por cada función, una con la información de la entrada de la función y otra con la información del retorno de la función. De este modo nuestro cliente podrá compilar las referencias a estas funciones.

Ejercicio número 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}**

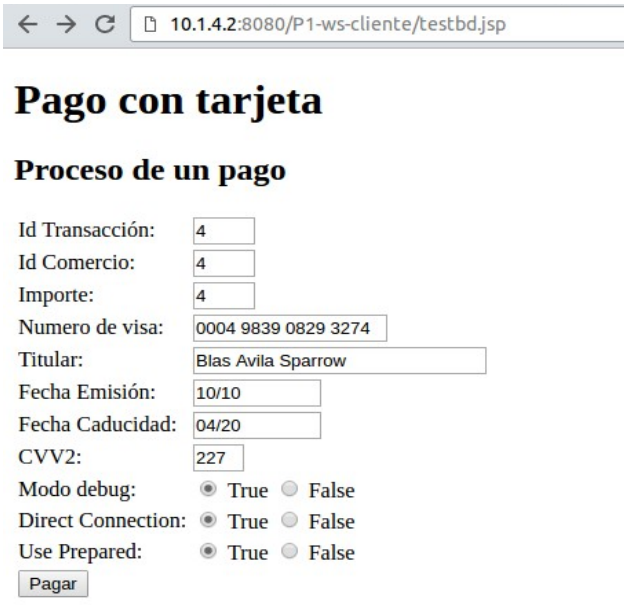
```
<target name="generar-stubs" depends="montar-jerarquia" description="Genera los stubs del cliente a partir del archivo WSDL">
	<!-- TODO - Implementar llamada wsimport -->
	<delete file="${build}/${tmpvisaclientjar}" />
	<jar jarfile="${build}/${tmpvisaclientjar}" >
		<fileset dir="${build.client}/WEB-INF/classes" />
	</jar>
	<move file="${build}/${tmpvisaclientjar}" todir="${build.client}/WEB-INF/lib" />
	<exec executable="${wsimport}">
		<arg line=" -d ${build.client}/WEB-INF/classes" />
		<arg line=" -p ${paquete}.visa" />
		<arg line=" ${wsdl.url}" />
	</exec>
</target>
```

Ejercicio número 13:

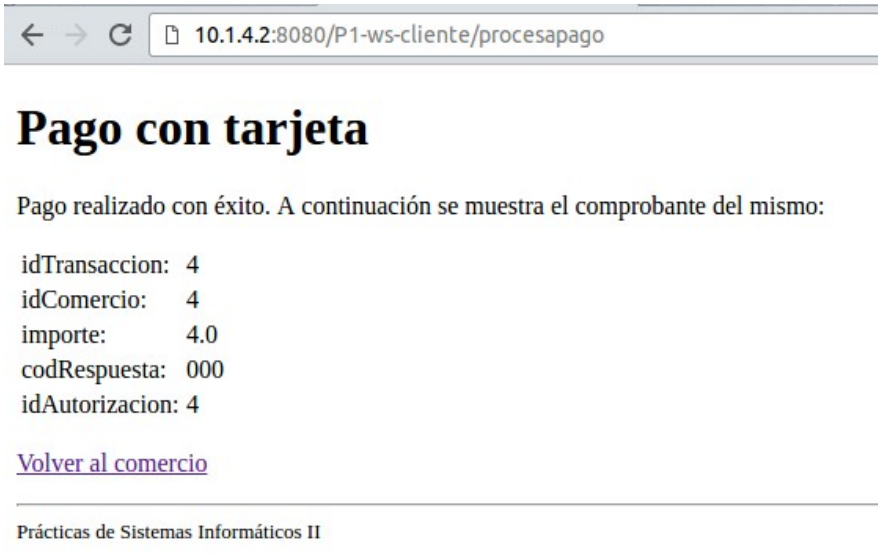
- **Realice un despliegue de la aplicación completo en dos nodos tal y como se explica en la Figura 8. Habrá que tener en cuenta que ahora en el fichero build.properties hay que especificar la dirección IP del servidor de aplicaciones donde se desplegará la parte del cliente de la aplicación y la dirección 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.**

Ejecutamos pagos desde testbd.jsp:



Ejecutamos el pago



Nos devuelve un mensaje de que el pago se ha realizado con éxito

Comprobamos en la base de datos que todo se ha realizado de forma correcta:

```
SELECT * FROM pago WHERE idautorizacion=4
```

Y la respuesta es un pago con ese id de autorización:

	idautorizaci integer	idtransacc character(	codrespu character	importe double pı	idcomercio character(1	numerotarjeta character(19)	fecha timestamp without time zone
1	4	4	000	4	4	0004 9839 0829 3274	2017-02-27 12:50:51.332013

Cuestiones

Cuestión número 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.

Los archivos por los que se pasa y su orden son el siguiente:

1. pago.html
2. servlet ComienzaPago
3. formdatosvisa.jsp
4. servlet ProcesaPago (detección del error)
5. formdatosvisa.jsp

Cuestión número 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?

Los servlets encargados de pedir información son el servlet ComienzaPago y el servlet ProcesaPago.

Cuestión número 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?

El servlet ComienzaPago solicita el id de transacción, el id de comercio, y el importe del pago. El servlet ProcesaPago solicita valores relativos a la tarjeta: número, titular, fecha de emisión y de caducidad y código de verificación.

Esta información la comparten mediante la variable de sesión (del tipo *HttpSession*), más concretamente en el atributo *ComienzaPago.ATTR_PAGO*.

Cuestión número 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?

La diferencia principal es la llamada a la función, pues la página testdb.jsp invoca directamente al servlet ProcesaPago sin pasar anteriormente por el servlet ComienzaPago o la página formdatosvisa.jsp.

Esto funciona correctamente ya que testdb.jsp solicita la información del pago y de la tarjeta a la vez. Además, en ComienzaPago se comprueba si la petición es de formdatosvisa.jsp o de testdb.jsp dependiendo de la variable de sesión. En este último caso la variable es nula y se crea un pago.