		Escuela Politécnica Superior Ingeniería Informática Prácticas de Sistemas Informáticos 2			
Grupo	2402	Práctica	1	Fecha	22/02/2019
Alumno/a		Gómez, Borzdynski, Óscar			
Alumno/a		Polanía, Bernárdez, Pablo Alejo			

Práctica 1A: Arquitectura de Java EE (Primera parte)

Ejercicio número 1:

Pasos realizados:

- Preparación del entorno (máquinas virtuales, instalación de herramientas necesarias...)
- Modificamos los siguientes ficheros en P1-base (descargado de moodle):
 - build.properties:
 - as.host → 10.2.5.1 (IP de la 1ª máquina virtual)
 - postgresql.properties:
 - db.host → 10.2.5.1
 - db.client.host → 10.2.5.1
- Para desplegar la aplicación en el entorno virtual ejecutamos **ant todo**. Esto genera la BBDD *Visa* cargándola de datos, compila la aplicación y la despliega.
- Para poder simular una compra (exitosa) accedemos a la BBDD *Visa* anteriormente generada usando pgAdmin3. Así conseguimos los datos de las tarjetas:

Edit Data - visa (10.2.5.1:5432) - visa - tarjeta					
numerotarjeta [PK] character(19)	titular character varying(128)	validadesde character(5)	validahasta character(5)	codigoverificacion character(3)	
0004 9839 0829 3274	Blas Avila Sparrow	10/10	04/20	227	
0028 1652 2262 7263	Clodoveo Moss Cozar	05/08	08/20	080	
0029 0099 6642 8003	Enjuto Vallejo Coll	03/09	10/20	126	
0039 2818 1198 8592	Hugo Linus Sparrow	01/10	09/20	971	
0060 4909 1343 5346	Pedro Cozar Martinez	02/10	05/20	187	
0065 5885 1459 8166	Luis Mojamuto Garau	08/09	02/20	351	
0075 0596 1967 0119	Gabriel Mojamuto Moss	08/10	11/20	287	
0086 5917 8302 1161	Emiliano Reyes Sparrow	08/10	07/20	252	
0093 7111 3396 6407	Blas Coll Cozar	01/09	02/20	217	
0097 9664 9744 9772	Jose Marques Gibson	11/09	11/20	491	
0134 8051 2205 1444	Gonzalo Dans Avila	01/09	11/20	171	
0148 5399 9603 9497	Emiliano Mas Linus	10/10	01/20	676	
0153 9794 4494 1695	Emiliano Sparrow Marques	02/10	06/20	153	
0170 7813 8076 4608	Blas Gonzalez Narvaez	11/09	04/20	755	
0174 1675 7273 7286	Armando Pelaez Vallejo	09/09	10/20	884	
0175 5593 1089 5719	Pedro Ribas Dans	05/08	06/20	723	
0178 1340 5311 1743	Blas Dominguez Poza	03/08	06/20	187	
0197 5440 0641 0861	Blas Argudo Dominguez	01/10	06/20	866	
0215 5694 2899 8560	John Dans Mojamuto	11/08	07/20	324	
0216 8365 8215 1330	Petra Mas Gracia	02/08	01/20	167	
0221 4618 0735 2394	Margarita Padro Dominguez	03/09	08/20	555	
0228 5679 8945 2468	Eva Lobo Padro	07/08	06/20	464	

5. Accedemos a la URL <http://10.2.5.1:8080/P1> (en modo seguro), realizamos un pago:



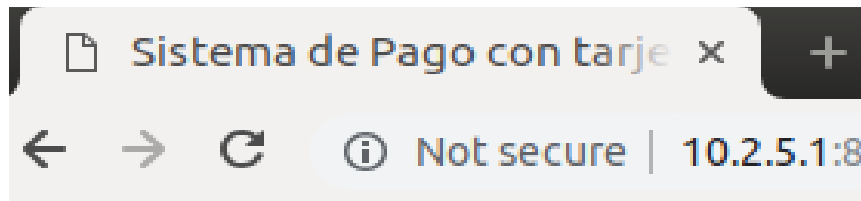
6. Comprobamos en la BBDD que el pago se registró correctamente:

idautorizacion [PK] serial	idtransaccion character(16)	codrespuesta character(3)	importe double precision	idcomercio character(16)	numerotarjeta character(19)	fecha timestamp without time zone
1	1	000	50	1	0039 2818 1198 8592	2019-02-13 10:00:57.67647

7. Accedemos a la URL <http://10.2.5.1:8080/P1/testbd.jsp> para comprobar el listado de los pagos:



8. En el mismo enlace borramos el pago que hemos hecho anteriormente:

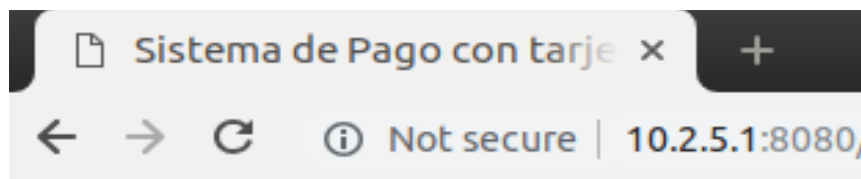


Pago con tarjeta

Se han borrado 1 pagos correctamente para

[Volver al comercio](#)

9. Comprobamos usando la funcionalidad del listado que ha sido borrado:



Pago con tarjeta

Lista de pagos del comercio 1

idTransaccion	Importe	codRespuesta	idAut
---------------	---------	--------------	-------

[Volver al comercio](#)

Ejercicio número 2:

Pasos realizados:

1. En primer lugar hemos implementado la conexión directa. Para ello hemos modificado las variables JDBC_DRIVER, JDBC_CONNSTRING, JDBC_USER y JDBC_PASSWORD del fichero DBTester.java con los datos de la BBDD del entorno virtual:

```
// Información de conexión
// Para conexiones directas, requerimos: driver, cadena de conexión,
// usuario y clave
private static final String JDBC_DRIVER =
    "org.postgresql.Driver";

// TODO: Definir la cadena de conexión a la base de datos
/*****
private static final String JDBC_CONNSTRING =
    "jdbc:postgresql://10.2.5.1:5432/visa;create=true";
*****/
private static final String JDBC_USER = "alumnodb";
private static final String JDBC_PASSWORD = "****";
```

2. En segundo lugar hemos utilizado la herramienta ant para volver a desplegar la aplicación modificada en la máquina virtual:

ant replegar

ant todo

Además, antes se debe ejecutar *reboot* en el entorno virtual.

3. Por último se prueba, al igual que en el ejercicio anterior, a efectuar, listar y borrar un pago con conexión directa.

Inserción de datos:

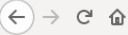


Pago con tarjeta

Proceso de un pago

Id Transacción:	<input type="text" value="3"/>
Id Comercio:	<input type="text" value="1"/>
Importe:	<input type="text" value="100"/>
Numero de visa:	<input type="text" value="0039 2818 1198 8592"/>
Titular:	<input type="text" value="Hugo Linus Sparrow"/>
Fecha Emisión:	<input type="text" value="01/10"/>
Fecha Caducidad:	<input type="text" value="09/20"/>
CVV2:	<input type="text" value="971"/>
Modo debug:	<input type="radio"/> True <input type="radio"/> False
Direct Connection:	<input checked="" type="radio"/> True <input type="radio"/> False
Use Prepared:	<input type="radio"/> True <input type="radio"/> False
<input type="button" value="Pagar"/>	

Mensaje de éxito:



10.2.5.1:8080/P1/procesapago

Pago con tarjeta

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

idTransaccion: 3
idComercio: 1
importe: 100.0
codRespuesta: 000
idAutorizacion: 1

[Volver al comercio](#)

Prácticas de Sistemas Informáticos II

Listado de pagos:



10.2.5.1:8080/P1/getpagos

Pago con tarjeta

Lista de pagos del comercio 1

idTransaccion	Importe	codRespuesta	idAutorizacion
3	100.0	000	1

[Volver al comercio](#)

Prácticas de Sistemas Informáticos II

Y finalmente borrado de pago:



10.2.5.1:8080/P1/delpagos

Pago con tarjeta

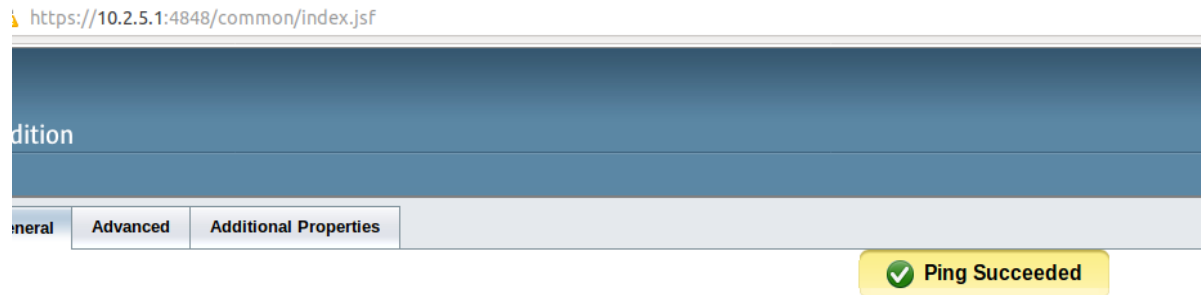
Se han borrado 1 pagos correctamente para el comercio 1

[Volver al comercio](#)

Prácticas de Sistemas Informáticos II

Ejercicio número 3:

- Al revisar el fichero `postgresql.properties` vemos que las variables toman estos valores
 - Nombre del pool: `VisaPool`
 - DataSource: `org.postgresql.ds.PGConnectionPoolDataSource`
- Accediendo a la *Consola de Administración* podemos verificar que estos recursos han sido desplegados correctamente:



Test JDBC Connection Pool

Test an existing JDBC connection pool. A JDBC connection pool is a group of reusable connections for a particular database.

[Test Defaults](#) [Flush](#) [Ping](#)

General Settings

Pool Name: VisaPool

DataSource Type: `javax.sql.ConnectionPoolDataSource`
Must be specified if the datasource class implements more than 1 of the interface.

DataSource Classname: `org.postgresql.ds.PGConnectionPoolDataSource`
Vendor-specific classname that implements the DataSource and/or XADataSource APIs

Driver Classname:
Vendor-specific classname that implements the java.sql.Driver interface.

Testing: ☒ **Enabled**
When enabled, the pool is pinged during creation or reconfiguration to identify and warn of any erroneous values for its attributes

Deployment Order: `100`
Specifies the loading order of the resource at server startup. Lower numbers are loaded first.

- Por último hemos revisado los valores de las variables para el pool de conexiones:

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

3.1. Initial and Minimal Pool Size:

Se trata del número mínimo hilos concurrentes que puede administrar el servidor y el número de hilos que se preparan cuando ésta se inicializa. Este número puede afectar al rendimiento de la aplicación en su inicialización. Si el número es muy alto, puede que el servidor tarde más en arrancar de lo deseado. Por otro lado si este número es muy bajo la ventaja que ofrece un pool de conexiones se pierde ya que no proveería el servicio de varias conexiones preparadas para la petición del cliente.

3.2. Maximum Pool Size:

Se refiere al máximo de conexiones que se pueden crear en la aplicación para satisfacer las peticiones de los clientes. Al igual que el anterior tiene desventajas si es un número muy bajo (no admitiría una gran concurrencia y ocasionaría la espera de los servicios por parte del cliente, restando eficacia a la aplicación) y si es un número muy alto (una gran cantidad de concurrencia podría sobrecargar el servidor y hacer que su funcionamiento se viese deteriorado con respuestas más lentas y corruptas, clientes en espera...)

3.3. Pool Resize Quantity:

Una vez se cumple el timeout de una conexión desocupada este parámetro entra en juego. Indica la cantidad de conexiones que se eliminan cada vez que se cumple esta condición. Si este número es muy alto podría dar problemas al ajustar la cantidad de conexiones concurrentes. Esto ocurriría cuando se hace una mala estimación para intentar corregir las conexiones ociosas.

3.4. Idle Timeout:

Es el tiempo que tarda en saltar el timeout de una conexión que está desocupada/libre. Si es muy alto se tienen recursos en uso que no son necesarios y se pierde eficacia en el rendimiento. En el caso contrario, si es muy bajo, podría dar errores al responder a peticiones de los clientes y cerrando la conexión con ellos al tardar demasiado en seguir con la petición.

3.5. Max Wait Time:

Se trata del caso contrario. Es el tiempo máximo que espera un cliente hasta enviar una señal de timeout. Si el tiempo es muy alto puede darse el caso de que el servidor haya cerrado la conexión por su cuenta. Si es muy baja puede llevar a errores en los que el servidor acabe repitiendo paquetes.

Ejercicio número 4:

Como se puede apreciar las consultas correspondientes a “Consulta de si una tarjeta es válida” y “Ejecución del pago” son:

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

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

Se ve claramente que en el caso de la consulta se lleva a acabo un SELECT para comprobar la existencia de la tupla con los datos pasados por la petición HTTP. En el caso de la ejecución del pago se ve como los nuevos valores son insertados en la tabla pago como una nueva tupla.

Ejercicio número 5:

Tras activar el modo debug y llevar a cabo una compra podemos ver en la página testdb.jsp que se han registrados los logs:

Log Level: ☐ Do not include more severe messages
Log entries are limited to those stored in the log file. Set appropriate log level in the Log Level page to ensure data is logged.

Modify Search

Instance:
Log File:

Log Viewer Results (40)

Records before 478 | Log File Record Numbers 478 through 517 | Records after 517 |

Record Number	Log Level	Message
517	SEVERE	[directConnection=false] select idAutorizacion, codRespuesta from pago where idTransaccion = '23' ... (details)
516	SEVERE	[directConnection=false] insert into pago(idTransaccion,importe,idComercio,numeroTarjeta) values ('2... (details)
515	SEVERE	[directConnection=false] select * from tarjeta where numeroTarjeta='0039 2818 1198 8592' and titular... (details)

Podemos ver que en la función *compruebaTarjeta()* se realiza un *errorLog()* con la query enviada a la base de datos, que podemos ver en la imagen con el número de registro 515.

Ejercicio número 6:

Encima de la clase VisaDAOWS se ha añadido `@WebService` para indicar que se trata de un servicio web.

Encima de los métodos `Tarjeta()` `realizaPago()` `isDebug()/setDebug()` `isPrepared()/setPrepared()` se ha añadido `@WebMethod` para indicar que serán exportados como métodos públicos del servicio.

Para los métodos `isDirectConnection()` / `setDirectConnection()` de la clase DBTester hemos utilizado la etiqueta `@Override` para sobrescribirlos en la clase VisaDAOWS y poder añadirles la etiqueta `@WebMethod`.

Para los parámetros de los métodos superiores se añade `@WebParam` para indicar que se trata de un argumento asociado a un método exportable del servicio.

El método `realizaPago()` se ha modificado de la siguiente manera:

- Las líneas `ret = false` han sido sustituidas por `ret = null`.
- Las líneas `ret = true` han sido sustituidas por `ret = pago`.
- El tipo de pago ha sido cambiado por `PagoBean`.
- El tipo devuelto por el método ha sido cambiado a `PagoBean`.

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

Porque se debe devolver un objeto de tipo `PagoBean` para poder devolver la información sobre el pago realizado.

Ejercicio número 7:

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

En el que se consigue en el URL `../P1-ws-ws/VisaDAOWSService?xsd=1`.

Se usan *XML-schemas*.

- **¿Qué tipos de datos predefinidos se usan?**

Se usan:

- `xs:string`
- `xs:boolean`
- `xs:int`
- `xs:double`

- **¿Cuáles son los tipos de datos que se definen?**

Se definen:

- Clases (`tns:tarjetaBean`)
- Métodos (`tns:compruebaTarjeta`)

Esto se lleva a cabo utilizando la etiqueta *element*.

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

Se les asocia la etiqueta *operation*. Vemos que dentro tenemos la etiqueta *input* y *output* para indicar los mensajes de entrada y de salida.

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

La etiqueta *message* que contiene partes identificadas con la etiqueta *part* que hace referencia a cada parámetro del método.

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

Se indica en la etiqueta *binding* y dentro de ella se indica con *soap:binding*.

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

La etiqueta *soap:address* que contiene la URL en el campo *location*.

Ejercicio número 8:

Se realizan los siguientes cambios en el código de *ProcesaPago.java* para que implemente el servicio web mediante *stubs* estáticos:

Este cambio se encarga de conseguir el *stub* del servidor para usarlo de manera local en el cliente y de capturar las excepciones:

```
try{ //Aquí podemos tener error de conexión (entre otros)
    VisaDAOWSService service = new VisaDAOWSService();
    VisaDAOWS dao = service.getVisaDAOWSPort();
}catch (Exception e){
    enviaError(e, request, response); //Enviamos el error con el exception trace completo
    return;
}
```

Ejercicio número 9:

Siguiendo las indicaciones del enunciado para implementar una aproximación de *stub* dinámico primero se debe modificar el fichero *web.xml* en el cual se ha añadido un contexto para definir la dirección URL del servidor:

```
<context-param>
  <param-name>direccion</param-name>
  <param-value>http://10.2.5.1:8080/P1-ws-ws/VisaDAOWSService</param-value>
</context-param>
```

Como se puede apreciar la etiqueta contexto se subdivide en un par de etiquetas nombre:valor (name:value). Para conseguir esa URL desde el código de *ProcesaPago.java* se han añadido las siguientes líneas:

```
try{ //Aqui podemos tener error de conexion (entre otros)
    VisaDAOWSService service = new VisaDAOWSService();
    VisaDAOWS dao = service.getVisaDAOWSPort();
    BindingProvider bp = (BindingProvider) dao;
    bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
                               getServletContext().getInitParameter("direccion"));
}
```

En este caso se accede al contexto del servicio con el nombre *direccion* definido anteriormente para recibir el valor de la URL. Así conseguimos un *stub* dinámico.

Ejercicio número 10:

Para que toda la funcionalidad de la página de pruebas *testbd.jsp* se realice como servicio web se deben modificar la clase *DelPagos* y la clase *GetPagos*. Esta modificación consiste en conseguir el servicio de la misma manera que en el ejercicio anterior en ambos ficheros (añadir las mismas líneas de código). Por otro lado, los métodos *delPagos()* y *getPagos()* del fichero *VisaDAWOS.java*, deben ser añadidos al servicio web con las etiquetas *@WebMethod* y *@WebParam* para sus parámetros.

Además se debe cambiar la declaración del retorno del método *getPagos()*. El enunciado recomienda cambiarlo a *ArrayList<PagoBean>* pero al compilar ha dado errores en el tipo de retorno. Por tanto se ha mantenido la sugerencia del enunciado pero a la hora de capturar este retorno se ha decidido recibirlo como tipo *List<PagoBean>*:

```
@WebMethod
public ArrayList<PagoBean> getPagos(@WebParam String idComercio) {

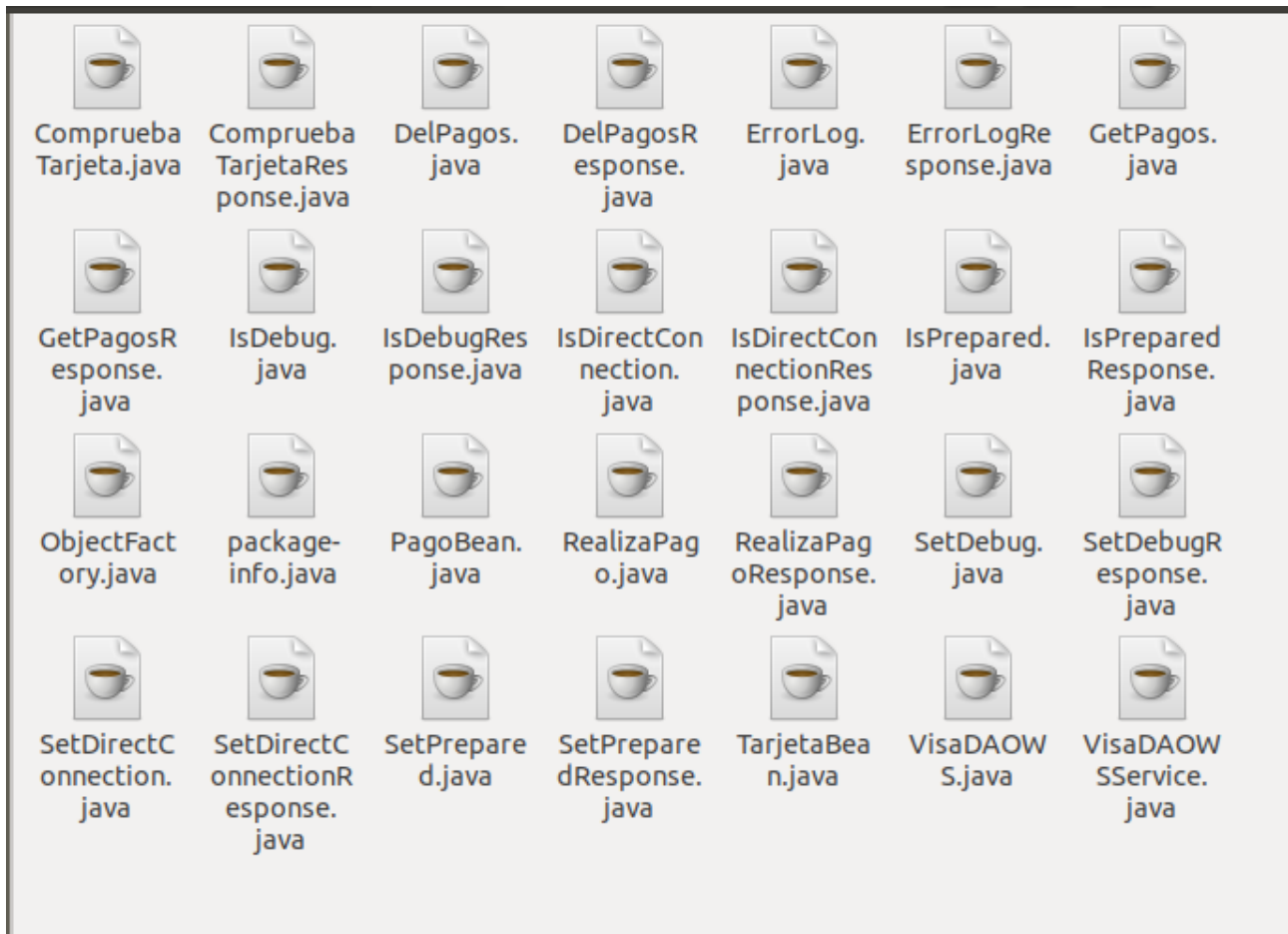
    /* Petición de los pagos para el comercio */
    List<PagoBean> paux = dao.getPagos(idComercio);
    PagoBean[] pagos = paux.toArray(new PagoBean[paux.size()]);
}
```

Ejercicio número 11:

Se ha ejecutado este comando para importar manualmente el WSDL a la carpeta local del cliente:

```
wsimport -xnocompile -d ./ -p ssii2.visa http://10.2.5.1:8080/P1-ws-ws/VisaDAOWSService?wsdl
```

Esto genera los *stubs* del cliente. Se puede comprobar que ha generado un fichero java por cada @WebService y cada @WebMethod declarados en el servidor como se esperaba:



La primera vez que se ha ejecutado el comando ha detectado un error debido a *setDebug()* del fichero *VisaDAOWS.java* pero siguiendo las indicaciones a pie de página del ejercicio 5 se ha arreglado el problema.

Ejercicio número 12:

Para este ejercicio queremos ejecutar este comando pero de manera dinámica:

```
wsimport -d ./build/client/WEB-INF/classes -p ssii2.visa http://10.2.5.1:8080/P1-ws-ws/VisaDAOWSService?wsdl
```

Es decir, cada vez que se despliegue el cliente. Por ello se modifica el fichero *build.properties*:

```
<!-- Este es el comando wsimport para generar los stubs -->
<exec executable="${wsimport}">
  <arg line=" -d ${build.client}/WEB-INF/classes "/>
  <arg line=" -p ${paquete}.visa "/>
  <arg line=" ${wsdl.url}" />
</exec>
```

Esta secuencia de etiquetas se encarga de ejecutar el comando apoyándose en las variables definidas en el fichero *build.properties*.

Ejercicio número 13:

En este apartado se pide que se realice un pago usando los dos entornos virtuales: cliente y servidor. Para ello se actualizan las variables *as.host.client* y *as.host.server* con nuestras respectivas IP's de cliente y servidor. A continuación se muestran evidencias de los pasos que se han seguido para realizar el pago:

Introducción datos tarjeta:

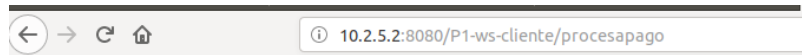


Pago con tarjeta

Proceso de un pago

Id Transacción:	<input type="text" value="1"/>
Id Comercio:	<input type="text" value="1"/>
Importe:	<input type="text" value="100"/>
Numero de visa:	<input type="text" value="0039 2818 1198 8592"/>
Titular:	<input type="text" value="Hugo Linus Sparrow"/>
Fecha Emisión:	<input type="text" value="01/10"/>
Fecha Caducidad:	<input type="text" value="09/20"/>
CVV2:	<input type="text" value="971"/>
Modo debug:	<input checked="" type="radio"/> True <input type="radio"/> False
Direct Connection:	<input type="radio"/> True <input type="radio"/> False
Use Prepared:	<input type="radio"/> True <input type="radio"/> False
<input type="button" value="Pagar"/>	

Confirmación pago tarjeta:



Pago con tarjeta

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

idTransaccion: 1
idComercio: 1
importe: 100.0
codRespuesta: 000
idAutorizacion: 1

[Volver al comercio](#)

Confirmación pago tarjeta (en BBDD):

1	1	1	000	100	1	0039 2818 1198 8592	2019-02-21 15:53:36.479891
---	---	---	-----	-----	---	---------------------	----------------------------

Lista de pagos:

←

→

↺

🏠

10.2.5.2:8080/P1-ws-cliente/getpagos

Pago con tarjeta

Lista de pagos del comercio 1

idTransaccion	Importe	codRespuesta	idAutorizacion
1	100.0	000	1

[Volver al comercio](#)

Borrado del pago:

←

→

↺

🏠

10.2.5.2:8080/P1-ws-cliente/delpagos

Pago con tarjeta

Se han borrado 1 pagos correctamente para el comercio 1

[Volver al comercio](#)

Lista de pagos tras el borrado:

←

→

↺

🏠

10.2.5.2:8080/P1-ws-cliente/getpagos

Pago con tarjeta

Lista de pagos del comercio 1

idTransaccion	Importe	codRespuesta	idAutorizacion
---------------	---------	--------------	----------------

[Volver al comercio](#)

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.

Desde *pago.html*, se accede a *Comienza Pago* (servlet), el cual envía los datos adquiridos a *formdatosvisa.jsp*, que tras ser rellenado con los datos de la tarjeta enviará esta información a *Procesa Pago* (servlet), que contactará con la base de datos *VisaDAO* para comprobar que la información es correcta. La query retorna un error por caducidad de la tarjeta, lo cual hace que *Procesa Pago* redirija al cliente a *error/muestraerror.jsp* que muestra en la pantalla del navegador un error en el pago.

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?

Cuando se usa *pago.html*, *Comienza Pago* se encarga de pedir la información de transacción, comercio e importe, mientras que *Procesa pago* se encarga de todo lo demás. La funcionalidad más importante la implementa este último servlet ya que conecta con la BBDD y aprueba el pago. En caso de éxito guarda el pago en la BBDD.

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 un identificador de la transacción, un id del comercio, un importe y una ruta de retorno (para cuando se completa el pago).

Procesa Pago necesita número de tarjeta, titular, fecha de emisión, fecha de caducidad y cvv.

Toda esta información es guardada en una sesión HTTP en el navegador del cliente (puede que de manera cifrada) y de esta forma los dos servlet comparten información. Además, una vez la información es válida se almacena en un objeto java.

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?

Se puede observar que en el primer caso, *testbd.jsp*, accede directamente al servlet de *Procesa Pago* en vez de acceder a *Comienza Pago* como es el caso de *pago.html*.

Como se puede apreciar en la imagen y al estudiar el código proporcionado el servlet encargado de "realizar el pago", es decir, acceder a la BBDD para comprobar los datos de la tarjeta y posteriormente insertar el pago, es *Procesa Pago*, no *Comienza Pago*. Por ello, al ejecutar *testbd.jsp* se piden todos los datos y funciona de la manera esperada.