		Escuela Politécnica Superior Ingeniería Informática Prácticas de Sistemas Informáticos 2			
Grupo	2391	Práctica	1A	Fecha	06/03/2021
Alumno/a	Mohedano Rodríguez, Daniel				
Alumno/a	Sopeña Niecko, Silvia				

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

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.

Para desplegar la aplicación web y la base de datos, seguimos los siguientes pasos. En primer lugar iniciamos la MV del entorno de producción e iniciamos el servidor de Glassfish con el comando:

```
$ asadmin start-domain domain1
```

A continuación, comprobamos que la MV del entorno de desarrollo estuviera correctamente configurada para la conexión con el servidor. Una vez cambiamos la IP para que se encontrase en el rango de direcciones del servidor, el último paso fue actualizar los archivos de propiedades *build.properties* y *postgresql.properties* para que reflejasen la IP de nuestro entorno de producción: 10.3.12.1. Por último, utilizamos *ant* para crear, compilar y desplegar todo:

```
$ ant todo
```

Tras realizar el pago comprobamos que tanto en Tora como en el servicio de listado de pagos se podía observar el pago realizado como se puede ver en las siguientes imágenes.

Pago con tarjeta

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

```
idTransaccion: 4867
idComercio: 479
importe: 48.0
codRespuesta: 000
idAutorizacion: 1
```

Figura 1: Pantalla tras realizar un pago (Ej1).

Pago con tarjeta

Lista de pagos del comercio 479

idTransaccion	Importe	codRespuesta	idAutorizacion
4867	48.0	000	1

[Volver al comercio](#)

Figura 2: Comprobación del pago realizado mediante el servicio de listado de pagos (Ej1).

Table Name	idautorizacion	idtransaccion	codrespuesta	importe	idcomercio	numerotarjeta	fecha
pago tarjeta	1	4867	000	48	479	1111 2222 3333 4444	26/02/21 07:38

Figura 3: Comprobación del pago realizado mediante Tora (Ej1).

La funcionalidad de borrar pagos también funcionaba correctamente.

Pago con tarjeta

Se han borrado 1 pagos correctamente para el comercio 479

[Volver al comercio](#)

Figura 4: Pantalla tras borrar el pago realizado (Ej1).

Table Name	idautorizacion	idtransaccion	codrespuesta	importe	idcomercio	numerotarjeta	fecha
pago tarjeta							

Figura 5: Comprobación del borrado del pago mediante Tora (Ej1).

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 postgresql, 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

En el archivo *VisaDAO.java* vimos que la conexión siempre se obtiene de un método *getConnection()* que está implementado en *DBTester.java*.

Por ello, para arreglar la conexión directa, modificamos las siguientes variables en dicho archivo:

```
1 private static final String JDBC_DRIVER =
2     "org.postgresql.Driver";
3 /*****
4 private static final String JDBC_CONNSTRING =
5     "jdbc:postgresql://10.3.12.1:5432/visa";
6 *****/
7 private static final String JDBC_USER = "alumnodb";
8 private static final String JDBC_PASSWORD = "";
9 /*****
10 private static final String JDBC_DSN =
11     "jdbc/VisaDB";
```

Figura 6: Variables de DBTester.java modificadas para la conexión directa.

Toda esta información la obtuvimos inspeccionando el archivo *postgresql.properties*.

Posteriormente comprobamos que funcionaba desde la página de pruebas extendida.

Pago con tarjeta

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

idTransaccion: 5432
idComercio: 789
importe: 486.0
codRespuesta: 000
idAutorizacion: 2

[Volver al comercio](#)

Figura 7: Pantalla tras realizar un pago con conexión directa.

2	2	5432	000	486	789	1111 2222 3333 4444	26/02/21 07:55
---	---	------	-----	-----	-----	---------------------	----------------

Figura 8: Comprobación del pago realizado mediante Tora..

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.

En el archivo *postgresql.properties* pudimos obtener la siguiente información:

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

Esto son el nombre del propio pool, VisaPool y el nombre del recurso correspondiente al DataSource,

jdbc/VisaDB. Además, en la consola de administración pudimos comprobar que se habían creado correctamente, como se puede observar en la siguiente imagen.

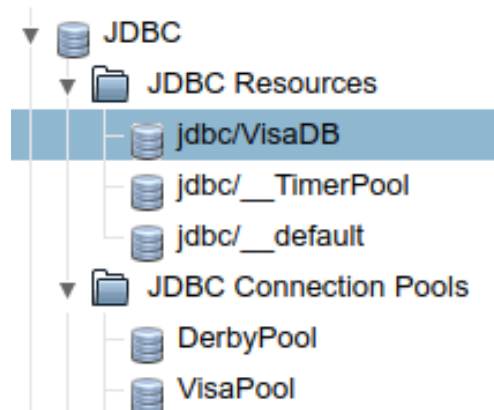


Figura 9: Recurso JDBC y pool de conexiones en la consola de administración.

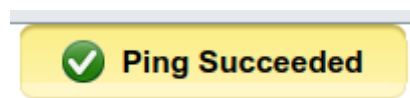


Figura 10: Resultado de Ping JDBC en la consola de administración.

La configuración del pool de conexiones fue la siguiente:

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

Figura 11: Configuración de la pool de conexiones.

El tamaño mínimo y máximo del pool de conexiones definirá la cantidad de conexiones simultáneas que puede soportar. Si ambos son muy grandes, entonces el pool debería ser un servicio muy usado que incluso en sus momentos de más bajo uso la cantidad de conexiones simultáneas sigue siendo alta (sino se estarían gastando recursos ya que un mínimo tan alto no sería necesario). Si ambos fuesen valores pequeños, entonces sucedería todo lo contrario. El *Pool Resize Quantity* permitirá al pool ser flexible y reaccionar acorde a cambios en el volumen de tráfico/uso. Si el valor es muy alto, el pool desaprovechará el mínimo de recursos (ya que cuando no se están usando quita una gran cantidad de conexiones) pero se arriesga a quitar de más y por lo tanto impactar su rendimiento negativamente. El *Idle Timeout* define como de rápido el pool se da cuenta de los cambios en volumen de uso. Con un valor muy alto podría ser ineficiente ya que tardaría mucho en reaccionar a dichos cambios. Pero, con un valor muy bajo podría reaccionar “demasiado” o incluso detectar como cambios en el volumen de uso cosas que no lo son. Por último, *Max Wait Time* evitará que el cliente desaproveche su tiempo esperando a respuestas.

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.

Los siguientes fragmentos de código los encontramos en el archivo *VisaDAO.java*.

```
1 /**
2  *  getQryCompruebaTarjeta
3  */
4 String getQryCompruebaTarjeta(TarjetaBean tarjeta) {
5     String qry = "select * from tarjeta "
6         + "where numeroTarjeta='" + tarjeta.getNumero()
7         + "' and titular='" + tarjeta.getTitular()
8         + "' and validaDesde='" + tarjeta.getFechaEmision()
9         + "' and validaHasta='" + tarjeta.getFechaCaducidad()
10        + "' and codigoVerificacion='" +
    tarjeta.getCodigoVerificacion() + "'";
11     return qry;
12 }
13
14 /**
15  *  getQryInsertPago
16  */
17 String getQryInsertPago(PagoBean pago) {
18     String qry = "insert into pago("
19         + "idTransaccion,"
20         + "importe,idComercio,"
21         + "numeroTarjeta)"
22         + " values ("
23         + "'" + pago.getIdTransaccion() + "',"
24         + pago.getImporte() + ","
25         + "'" + pago.getIdComercio() + "',"
26         + "'" + pago.getTarjeta().getNumero() + "'"
27         + ")";
28     return qry;
29 }
```

Figura 12: Queries SQL para comprobar tarjeta y ejecutar un pago.

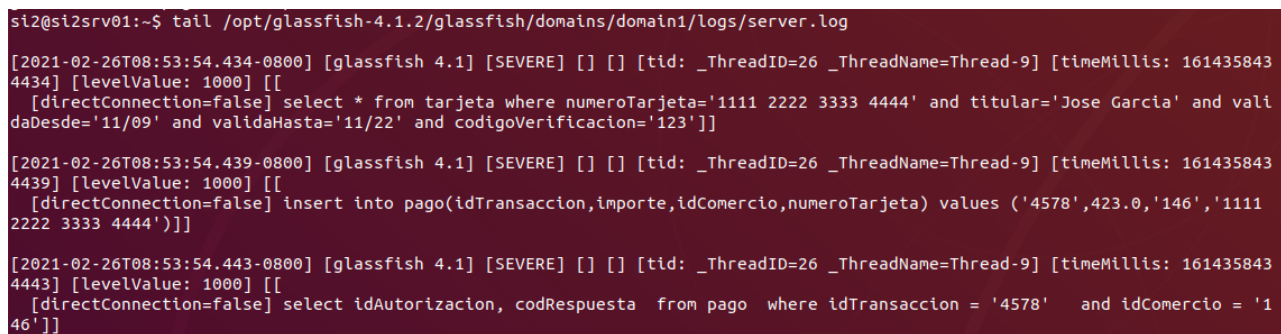
Con estas queries la aplicación puede validar tarjetas de crédito y también ejecutar pagos. Además, estas mismas queries también se encontraban escritas en strings con el placeholder “?” para poderse utilizar en prepared statements. Estas queries, a su vez, se ejecutan dentro de los métodos *compruebaTarjeta()* y *realizaPago()* de la clase *VisaDAO*.

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.

El método errorLog() se utiliza dentro de los métodos *compruebaTarjeta()*, *realizaPago()*, *getPagos()* y *delPagos()*. En todos los casos, el método se utiliza siempre antes de realizar cualquier tipo de query para escribir en el log que query se va a realizar. Además, también se utiliza cuando se captura una excepción para imprimir la propia excepción.

A continuación, se muestra el contenido del log tras realizar un pago con la opción de debug activada:



```
si2@si2srv01:~$ tail /opt/glassfish-4.1.2/glassfish/domains/domain1/logs/server.log

[2021-02-26T08:53:54.434-0800] [glassfish 4.1] [SEVERE] [] [] [tid: _ThreadID=26 _ThreadName=Thread-9] [timeMillis: 161435843
4434] [levelValue: 1000] [[
  [directConnection=false] select * from tarjeta where numeroTarjeta='1111 2222 3333 4444' and titular='Jose Garcia' and vali
daDesde='11/09' and validaHasta='11/22' and codigoVerificacion='123']]

[2021-02-26T08:53:54.439-0800] [glassfish 4.1] [SEVERE] [] [] [tid: _ThreadID=26 _ThreadName=Thread-9] [timeMillis: 161435843
4439] [levelValue: 1000] [[
  [directConnection=false] insert into pago(idTransaccion,importe,idComercio,numeroTarjeta) values ('4578',423.0,'146','1111
2222 3333 4444')]

[2021-02-26T08:53:54.443-0800] [glassfish 4.1] [SEVERE] [] [] [tid: _ThreadID=26 _ThreadName=Thread-9] [timeMillis: 161435843
4443] [levelValue: 1000] [[
  [directConnection=false] select idAutorizacion, codRespuesta from pago where idTransaccion = '4578' and idComercio = '1
46']]
```

Figura 13: Aspecto del log tras realizar un pago en modo debug.

Ejercicio 6:

Realicene 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()*

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?

```

1 public VisaDAOWS() {
2     return;
3 }
4
5
6 @WebMethod(operationName = "compruebaTarjeta")
7 public boolean compruebaTarjeta(@WebParam(name = "compruebaTarjetaParam")
    TarjetaBean tarjeta) {

```

Figura 14: Modificación al constructor de la clase y `compruebaTarjeta()`

```

1 @WebMethod(operationName = "realizaPago")
2 public synchronized PagoBean realizaPago(@WebParam(name =
    "realizaPagoParam") PagoBean pago) {
3     Connection con = null;
4     Statement stmt = null;
5     ResultSet rs = null;
6     boolean ret = false;
7     String codRespuesta = "999"; // En principio, denegado
8
9     // TODO: Utilizar en funcion de isPrepared()
10    PreparedStatement pstmt = null;
11
12    // Calcular pago.
13    // Comprobar id.transaccion - si no existe,
14    // es que la tarjeta no fue comprobada
15    if (pago.getIdTransaccion() == null) {
16        return null;
17    }
18    try {
19        .
20        .
21        .
22    } finally {
23        .
24        .
25        .
26    }
27
28    return ret ? pago : null;
29 }

```

Figura 15: Publicación y modificación del método `realizaPago()`

La única modificación que se realizó al código fue que en vez de devolver la variable `ret` (que indica si todo ha funcionado correctamente o no), se devuelve el pago actualizado o `null` en función del valor de `ret`.

```

1 @WebMethod(operationName = "isDebug")
2 public boolean isDebug() {
3     return debug;
4 }

```

Figura 16: Publicación de `isDebug()`

```

1 @WebMethod(operationName = "setDebug")
2 public void setDebug(@WebParam(name = "setDebugParam") boolean debug) {
3     this.debug = debug;
4 }
5
6
7 @WebMethod(operationName = "setDebugS")
8 @RequestWrapper(className="server.ssii2.visa.jaxws.setDebugS")
9 @ResponseWrapper(className="server.ssii2.visa.jaxws.setDebugSResponse")
10 public void setDebug(@WebParam(name = "setDebugSParam") String debug) {
11     this.debug = (debug.equals("true"));
12 }

```

Figura 17: Publicación de `setDebug()`

Decidimos publicar `setDebug()` de esta forma para seguir teniendo acceso a ambos métodos por si en el futuro era conveniente tener acceso a ambas implementaciones.

```

1 @WebMethod(operationName = "isPrepared")
2 public boolean isPrepared() {
3     return prepared;
4 }
5
6
7 @WebMethod(operationName = "setPrepared")
8 public void setPrepared(@WebParam(name = "setPreparedParam") boolean
    prepared) {
9     this.prepared = prepared;
10 }

```

Figura 18: Publicación de `isPrepared()` y `setPrepared()`


```

1 @Override
2 @WebMethod(operationName = "isDirectConnection")
3 public boolean isDirectConnection() {
4     return super.isDirectConnection();
5 }
6
7
8 @Override
9 @WebMethod(operationName = "setDirectConnection")
10 public void setDirectConnection(@WebParam(name =
    "setDirectConnectionParam") boolean directConnection) {
11     super.setDirectConnection(directConnection);
12 }

```

Figura 19: Publicación de los métodos *isDirectConnection()* y *setDirectConnection()*

Hay que modificar el parámetro de retorno del método *realizaPago()* ya que ahora ese método se encuentra en el servidor/servicio web. Esto provoca que ya no sea suficiente modificar el pago, puesto que esa actualización de la variable no le llegará al cliente (el método está tratando con una “copia” de la variable). Por ello es necesario devolver de forma activa el nuevo pago actualizado para que el cliente tenga acceso a él.

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. Conteste a las siguientes preguntas:

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

```

1 <types>
2     <xsd:schema>
3         <xsd:import namespace="http://dao.visa.ssi2/"
    schemaLocation="http://10.3.12.1:8080/P1-ws-ws/VisaDAOWSService?xsd=1" />
4     </xsd:schema>
5 </types>

```

Figura 20: Definición de *types*

Como se puede ver, se importa un archivo XSD (utilizado para definición de datos) que está en la ubicación proporcionada. En el se definen los argumentos y parámetros de retorno de las funciones publicadas.

- ¿Qué tipos de datos predefinidos se usan?

A lo largo del documento se utilizan *xs:string*, *xs:boolean*, *xs:double* y *xs:int*.

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

Aunque hay definidos muchos `xs:complexType` para, como se ha dicho antes, los argumentos y parámetros de salida de las funciones, tipos de datos definidos como tal encontramos los siguientes:

```
1 <xs:complexType name="pagoBean">
2   <xs:sequence>
3     <xs:element name="codRespuesta" type="xs:string" minOccurs="0" />
4     <xs:element name="idAutorizacion" type="xs:string" minOccurs="0" />
5     <xs:element name="idComercio" type="xs:string" minOccurs="0" />
6     <xs:element name="idTransaccion" type="xs:string" minOccurs="0" />
7     <xs:element name="importe" type="xs:double" />
8     <xs:element name="rutaRetorno" type="xs:string" minOccurs="0" />
9     <xs:element name="tarjeta" type="tns:tarjetaBean" minOccurs="0" />
10  </xs:sequence>
11 </xs:complexType>
12 ...
13 <xs:complexType name="tarjetaBean">
14   <xs:sequence>
15     <xs:element name="codigoVerificacion" type="xs:string" minOccurs="0" />
16     <xs:element name="fechaCaducidad" type="xs:string" minOccurs="0" />
17     <xs:element name="fechaEmision" type="xs:string" minOccurs="0" />
18     <xs:element name="numero" type="xs:string" minOccurs="0" />
19     <xs:element name="titular" type="xs:string" minOccurs="0" />
20   </xs:sequence>
21 </xs:complexType>
```

Figura 21. Definición de PagoBean y TarjetaBean

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

<operation>

```
1 <operation name="getPagos">
2   <input wsam:Action="http://dao.visa.ssii2/VisaDAOWS/getPagosRequest"
3     message="tns:getPagos" />
4   <output wsam:Action="http://dao.visa.ssii2/VisaDAOWS/getPagosResponse"
5     message="tns:getPagosResponse" />
6 </operation>
```

Figura 22: Ejemplo de definición de método

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

<message>

```
1 <message name="getPagos">
2   <part name="parameters" element="tns:getPagos" />
3 </message>
4 <message name="getPagosResponse">
5   <part name="parameters" element="tns:getPagosResponse" />
6 </message>
```

Figura 23: Ejemplo de mensajes intercambiados por un método

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

```
<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
```

En este caso se declara el uso de http.

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

```
<soap:address location="http://10.3.12.1:8080/P1-ws-ws/VisaDAOWSService"/>
```

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.

```
1 @Override
2 protected void processRequest(HttpServletRequest request,
3                               HttpServletResponse response) throws ServletException, IOException {
4     ...
5     VisaDAOWSService service = new VisaDAOWSService();
6     VisaDAOWS dao = service.getVisaDAOWSPort();
7     ...
8     pago.setTarjeta(tarjeta);
9     try{
10         if (! dao.compruebaTarjeta(tarjeta)) {
11             enviaError(new Exception("Tarjeta no autorizada:"), request,
12 response);
13         return;
14         }
15         pago = dao.realizaPago(pago);
16         if (pago == null) {
17             enviaError(new Exception("Pago incorrecto"), request,
18 response);
19         return;
20         }
21         request.setAttribute(ComienzaPago.ATTR_PAGO, pago);
22         if (sesion != null) sesion.invalidate();
23         reenvia("/pagoexitos.jsp", request, response);
24         return;
25     } catch (ServletException | IOException ex){
26         // Si es una excepcion esperada, simplemente se vuelve a throwear
27         throw ex;
28     } catch (Exception ee){
29         // Si es otra excepcion que no se esperaba, error
30         enviaError(ee, request, response);
31         return;
32     }
33 }
```

Figura 24: Modificación de código en `ProcesaPago.java`

Las únicas modificaciones realizadas fueron incluir la nueva creación del *dao* y al realizar las llamadas remotas (en este caso a *compruebaTarjeta()* y *realizaPago()*) se incluyó un bloque para capturar excepciones. Si alguna de las excepciones que se capturan son las que el propio método *processRequest* se supone que lanza, simplemente se vuelven a lanzar. En el caso de ser cualquier otra excepción (provocada por los métodos remotos) se envía un error con dicha excepción. Otro pequeño cambio fue actualizar el código para que se tenga en cuenta la modificación realizada a *realizaPago()*, la cual ya no devuelve boolean sino el pago actualizado.

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.

Se incluyeron las dos líneas de código proporcionadas en el guion de la práctica y también se actualizó el fichero *web.xml* de la siguiente forma:

```
1 <context-param>
2     <param-name>rutaservicio</param-name>
3     <param-value>http://10.3.12.1:8080/P1-ws-ws/VisaDAOWSService</param-
   value>
4 </context-param>
```

Figura 25: Actualización de *web.xml*

```
1 BindingProvider bp = (BindingProvider) dao;
2 bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
   getServletContext().getInitParameter("rutaservicio"));
```

Figura 26: Código añadido a *ProcesaPago.java* después de obtener el DAOWS

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.

```
1 @WebMethod(operationName = "delPagos")
2 public int delPagos(@WebParam(name = "delPagosParam") String idComercio)
```

Figura 27: Publicación de *delPagos()*

```

1 protected void processRequest(HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException {
2     VisaDAOWSService service = new VisaDAOWSService();
3     VisaDAOWS dao = service.getVisaDAOWSPort();
4     BindingProvider bp = (BindingProvider) dao;
5     bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
  getServletContext().getInitParameter("rutaservicio"));
6
7     /* Se recoge de la petici&oacute;n el par&aacute;metro idComercio*/
8     String idComercio = request.getParameter(PARAM_ID_COMERCIO);
9
10    /* Petici&oacute;n de los pagos para el comercio */
11    try {
12        int ret = dao.delPagos(idComercio);
13
14        if (ret != 0) {
15            request.setAttribute(ATTR Borrados, ret);
16            reenvia("/borradook.jsp", request, response);
17        }
18        else {
19            reenvia("/borradoerror.jsp", request, response);
20        }
21        return;
22    } catch (ServletException | IOException ex){
23        throw ex;
24    } catch (Exception ee){
25        enviaError(ee, request, response);
26        return;
27    }
28 }

```

Figura 28: Modificación de DelPagos.java

Al igual que en el ejercicio anterior, los únicos cambios fueron obtener el DAO remoto, hacer el binding y por último incluir un bloque para capturar excepciones al hacer llamadas al método remoto.

```

1 @WebMethod(operationName = "getPagos")
2 public ArrayList<PagoBean> getPagos(@WebParam(name = "getPagosParam")
  String idComercio)

```

Figura 29: Publicación de getPagos()

Además de publicar el método, se modificó para que devolviera un ArrayList en vez de un array de PagoBean. El único cambio que hubo que realizar en el código fue que devolviese un ArrayList que era precisamente donde se guardaban los resultados, eliminando un paso extra que se realizaba para convertir a array de PagoBean.

```

1 protected void processRequest(HttpServletRequest request,
  HttpServletResponse response) throws ServletException, IOException {
2     VisaDAOWSService service = new VisaDAOWSService();
3     VisaDAOWS dao = service.getVisaDAOWSPort();
4
5     BindingProvider bp = (BindingProvider) dao;
6     bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
  getServletContext().getInitParameter("rutaservicio"));
7
8     /* Se recoge de la petici&oacute;n el par&aacute;metro idComercio*/
9     String idComercio = request.getParameter(PARAM_ID_COMERCIO);
10
11     /* Petici&oacute;n de los pagos para el comercio */
12     try{
13         ArrayList<PagoBean> ret = new ArrayList<PagoBean>
  (dao.getPagos(idComercio));
14         PagoBean[] pagos = new PagoBean[ret.size()];
15         pagos = ret.toArray(pagos);
16
17         request.setAttribute(ATTR_PAGOS, pagos);
18         reenvia("/listapagos.jsp", request, response);
19         return;
20     } catch (ServletException | IOException ex){
21         throw ex;
22     } catch (Exception ee){
23         enviaError(ee, request, response);
24         return;
25     }
26 }

```

Figura 30: Modificación de GetPagos.java

El único cambio que cabe mencionar es porque se utiliza el constructor de ArrayList en vez de simplemente igualar la variable *ret* a lo que devuelve *dao.getPagos()*. La razón es muy simple, al compilar el código cambia un poco y se define que *getPagos()* no devuelve un ArrayList sino una List genérica, por lo tanto esa asignación no se puede realizar. Por ello se utiliza el constructor de ArrayList que acepta una Collection de cualquier tipo y construye el ArrayList a partir de eso. Por último, simplemente se pasa a array de nuevo el resultado para que *listapagos.jsp* pueda procesarlo correctamente.

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. El comando utilizado fue el siguiente:

```

wsimport -d build/client/WEB-INF/classes/ -p ssii2.visa
http://10.3.12.1:8080/P1-ws-ws/VisaDAOWSService?wsdl

```


CompruebaTarjeta.class	package-info.class
CompruebaTarjetaResponse.class	PagoBean.class
DelPagos.class	RealizaPago.class
DelPagosResponse.class	RealizaPagoResponse.class
ErrorLog.class	SetDebug.class
ErrorLogResponse.class	SetDebugResponse.class
GetPagos.class	SetDebugS.class
GetPagosResponse.class	SetDebugSResponse.class
IsDebug.class	SetDirectConnection.class
IsDebugResponse.class	SetDirectConnectionResponse.class
IsDirectConnection.class	SetPrepared.class
IsDirectConnectionResponse.class	SetPreparedResponse.class
IsPrepared.class	TarjetaBean.class
IsPreparedResponse.class	VisaDAOWS.class
ObjectFactory.class	VisaDAOWSService.class

Figura 31: Listado de clases generadas por wsimport

La mayoría de estas clases están relacionadas con los tipos definidos en el WSDL para los argumentos y parámetros de salida de los métodos remotos. En general, se crea todo lo necesario para que el cliente pueda acceder a los objetos/métodos del servidor como si fuesen locales.

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).

```

1 <!-- TODO - Implementar llamada wsimport -->
2 <exec executable="wsimport">
3   <arg line=" -d ${build.client}/WEB-INF/classes" />
4   <arg line=" -p ${paquete}.visa" />
5   <arg line=" http://10.3.12.1:8080/P1-ws-ws/VisaDAOWSService?wsdl" />
6 </exec>

```

Figura 32: Modificación del target generar-stubs

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.

Tras desplegar todo procedimos a realizar pagos para comprobar que todo funcionase correctamente.

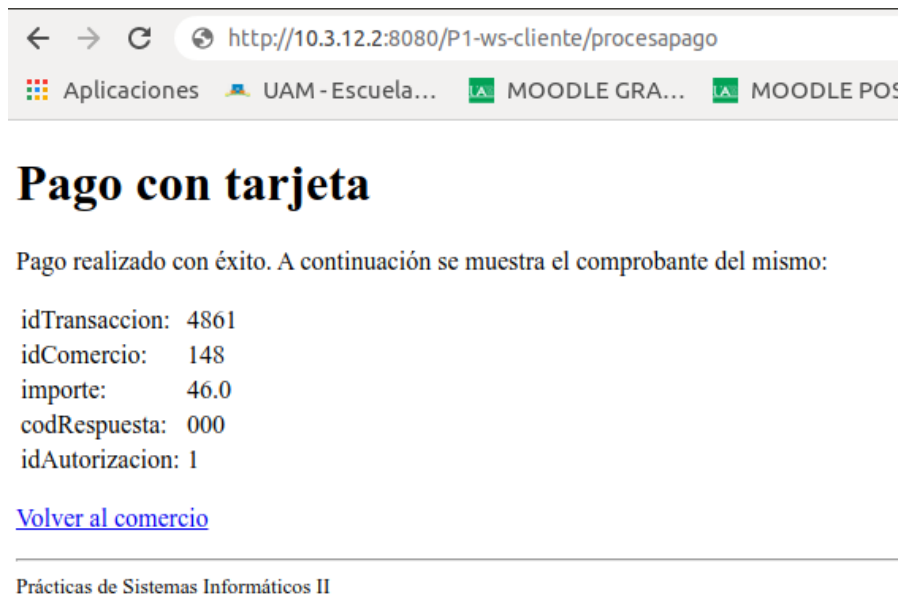


Figura 33: Página tras realizar un pago desde el cliente

	idautorizacion	idtransaccion	codrespuesta	importe	idcomercio	numero
1	1	4861	000	46	148	1111 22

Figura 34: Comprobación de la realización del pago desde TORA

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.

Al comenzar el pago desde el servlet ComienzaPago se muestra, una vez validados los campos proporcionados en el primer formulario, el segundo formulario *formdatosvisa.jsp*. Cuando se envía el segundo formulario con la fecha de caducidad expirada, los datos los recibe el servlet ProcesaPago. Este, antes de hacer nada, comprueba que la información de la tarjeta proporcionada es válida mediante una instancia de la clase ValidadorTarjeta. Aquí es donde se vería que la fecha de caducidad ha expirado y por lo tanto el servlet ProcesaPago volvería a mostrar el formulario de *formdatosvisa.jsp* de nuevo, pero en este caso mostrando un error.

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, y cuáles son los encargados de procesarla?

Como se ha comentado en la cuestión anterior, la solicitud de información es en realidad realizada por parte de los formularios. Los servlets ComienzaPago y ProcesaPago se encargan de procesar esa información y responder de forma acorde.

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?

ComienzaPago recibe información relacionada con el propio pago como el id del comercio, el de la transacción, el importe, etc. Esta información es guardada en un objeto de tipo PagoBean y se guarda en la sesión de la request. ProcesaPago recibe información relacionada con la tarjeta que posteriormente guarda en un objeto de tipo TarjetaBean. También tiene acceso al pago generado por ComienzaPago obteniéndolo de la sesión.

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 es que si se utiliza la página *testbd.jsp* en ningún momento ComienzaPago recibe información y por lo tanto no se crea el pago y se guarda en la sesión. Pero, para que funcione correctamente, ProcesaPago tiene una parte de código que se encarga de comprobar esto mismo. Si la sesión no existe y por lo tanto no está guardado el pago procedente de ComienzaPago significa que los datos que está recibiendo ProcesaPago han sido generados por testbd.jsp. Por lo tanto, pasa a crear ese pago desde la información del request, además de configurar el DAO con las opciones elegidas.

```
1 HttpSession sesion = request.getSession(false);
2 if (sesion != null) {
3     pago = (PagoBean) sesion.getAttribute(ComienzaPago.ATTR_PAGO);
4 }
5 if (pago == null) {
6     pago = creaPago(request);
7     boolean isdebug = Boolean.valueOf(request.getParameter("debug"));
8     dao.setDebug(isdebug);
9     boolean isdirectConnection =
10     Boolean.valueOf(request.getParameter("directConnection"));
11     dao.setDirectConnection(isdirectConnection);
12     boolean usePrepared =
13     Boolean.valueOf(request.getParameter("usePrepared"));
14     dao.setPrepared(usePrepared);
15 }
```

Figura 35: Comprobación de si la sesión está creada y el pago generado por ComienzaPago