		<b>Escuela Politécnica Superior</b> <b>Ingeniería Informática</b> <b>Prácticas de Sistemas Informáticos 2</b>			
<b>Grupo</b>	2401	<b>Práctica</b>	1b	<b>Fecha</b>	4/03/2015
<b>Alumno</b>	Kasner Tourné, Cristina				
<b>Alumno</b>	Guridi Mateos, Guillermo				

## Cuestión 1

Editar el archivo `VisaDAOLocal.java` y comprobar la definición de dicha interfaz. Anote sus comentarios en la memoria.

En este archivo se especifican los métodos que van a poder ser llamados sobre el EJB que implemente la interfaz `VisaDAOLocal`.

Vemos que tiene la etiqueta `@Local`, que significa:

- Debe ejecutarse en la misma Máquina Virtual Java del EJB al que accede
- Puede ser un componente web u otro EJB
- Para el cliente local, la localización del EJB al que accede no es transparente

## Ejercicio 1

Introduzca las siguientes modificaciones en el bean `VisaDAOBean` para convertirlo en un EJB `stateless` con interfaz local:

- Declarar la clase con interfaz local y la anotación `Stateless`

```

1  ...
   import javax.ejb.Stateless;
3  ...
   @Stateless(mappedName="VisaDAOBean")
5  public class VisaDAOBean extends DBTester implements
      VisaDAOLocal {
   ...

```

- Eliminar el constructor por defecto de la clase
- Ajustar los métodos `getPagos()` / `realizaPago()` a la interfaz definida en `VisaDAOLocal`

Para ajustar los métodos a la interfaz hemos quitado la etiqueta de `synchronize` ya que no se utilizan en los métodos especificados en `VisaDAOLocal`.

También hemos cambiado el retorno del método `getPagos`, antes devolvía `ArrayList` y ahora devuelve `PagoBean []` para respetar las declaraciones de `VisaDAOLocal`.

## Ejercicio 2

Modificar el servlet `ProcesaPago` para que acceda al EJB local. Para ello, modificar el archivo `ProcesaPago.java` de la siguiente manera:

En la sección de preproceso, añadir las siguientes importaciones de clases que se van a utilizar:

```
...
2 import javax.ejb.EJB;
import ssii2.visa.VisaDAOLocal;
4 ...
```

Se deberán eliminar estas otras importaciones que dejan de existir en el proyecto:

```
import ssii2.visa.VisaDAOWSService; // Stub generado
    automaticamente
2 import ssii2.visa.VisaDAOWS; // Stub generado automaticamente
```

Añadir como atributo de la clase que implementa el servlet el objeto proxy que permite acceder al EJB local, con su correspondiente anotación que lo declara como tal:

```
...
2 @EJB(name="VisaDAOBean", beanInterface=VisaDAOLocal.class)
private VisaDAOLocal dao;
4 ...
```

En el cuerpo del servlet, eliminar la declaración de la instancia del antiguo webservice `VisaDAOWS`, así como el código necesario para obtener la referencia remota

```
...
2 VisaDAOWS dao = null;
VisaDAOWSService service = new VisaDAOWSService();
4 dao = service.getVisaDAOWSPort();
```

**Importante:** Esta operación deberá ser realizada para todos los servlets del proyecto que hagan uso del antiguo `VisaDAOWS`. Verifique también posibles errores de compilación y ajustes necesarios en el código al cambiar la interfaz del antiguo `VisaDAOWS` (en particular, el método `getPagos()`).

Hemos eliminado de los servlets todo lo referente a los *Web Services* (que estaban en los respectivos métodos `ProcessRequest` del cliente) ya que ahora hacemos uso del EJB que hace uso de la interfaz `VisaDAOLocal`.

Entre las cosas que hemos tenido que quitar están los imports del paquete `javax.xml.ws` y todo el código que obtenía `VisaDAO` a través de un servicio y le asignaba una dirección de recurso dinámicamente. Por supuesto, tan bien borramos el ya no necesario parámetro de `web.xml` que especificaba las rutas al

servidor.

En `getPagos` hemos vuelto a cambiar el tipo de retorno para que coincida con la declaración que hay en `VisaDAOLocal`.

## Cuestión 2

Editar el archivo `application.xml` y comprobar su contenido. Verifique el contenido de todos los archivos `.jar` / `.war` / `.ear` que se han construido hasta el momento (empleando el comando `jar -tvf`). Anote sus comentarios en la memoria.

Vemos que en el archivo `application.xml` hay dos módulos:

- Uno que contiene los EJBs (parte ‘servidor’).
- En el otro se declara la parte web de la aplicación indicando el `.war` del cliente , y la dirección en la que será desplegada (`/P1-ejb-cliente`)

Cuando ejecutamos el comando `jar -tvf` vemos que nos muestra la tabla de contenidos del archivo al que se lo hemos aplicado. Vamos a explicar el caso del `.ear`. Contiene 4 archivos:

- `application.xml`, del que ya hemos hablado
- `MANIFEST.MF` , que contiene información sobre la versión de `ant` usada para la empaquetación.
- `P1-ejb-cliente.war`, que es el cliente empaquetado.
- `P1-ejb.jar` , que es el servidor empaquetado.

Al comprobar los contenidos de `P1-ejb.jar` encontramos los `.class` correspondientes a los EJBs, listos para ser utilizados por el cliente.

En cambio, al examinar los contenidos de `P1-ejb-cliente.war`, vimos que a parte de los `.class` encontrados también existían ficheros de configuración del servicio web, dentro de `WEB-INF`.

## Ejercicio 3

Preparar los PCs con el esquema descrito y realizar el despliegue de la aplicación:

- Editar el archivo `build.properties` para que las propiedades `as.host.client` y `as.host.server` contengan la dirección IP del servidor de aplicaciones.
- Editar el archivo `postgresql.properties` para la propiedad `db.client.host` y `db.host` contengan las direcciones IP adecuadas para que el servidor de aplicaciones se conecte al postgresql, ambos estando en servidores diferentes.

Desplegar la aplicación de empresa  
ant desplegar

The screenshot shows the GlassFish administration console interface. The top bar indicates the user is 'admin' on 'domain1' with server version '10.1.3.2'. The left sidebar shows a tree view of the domain structure, with 'P1-ejb' selected under 'Applications'. The main panel displays the configuration for 'P1-ejb', including its status (Enabled), virtual servers (server), and deployment order (100). Below this, a table titled 'Modules and Components (9)' lists the deployed modules and their components.

Module Name	Engines	Component Name	Type	Action
P1-ejb-cliente.war	[web]	default	Servlet	Launch
P1-ejb-cliente.war		jsp	Servlet	
P1-ejb-cliente.war		DelPagos	Servlet	
P1-ejb-cliente.war		ProcesaPago	Servlet	
P1-ejb-cliente.war		GetPagos	Servlet	
P1-ejb-cliente.war		ComienzaPago	Servlet	
P1-ejb.jar	[ejb, weld]	VisaDAOBean	StatelessSessionBean	

Figura 1: Comprobar en la administración de Glassfish el despliegue de la aplicación

## Ejercicio 4

Comprobar el correcto funcionamiento de la aplicación mediante llamadas directas a través de las páginas `pago.html` y `testbd.jsp` (sin `directconnection`). Realice un pago. Lístelo. Elimínelo. Téngase en cuenta que la aplicación se habrá desplegado bajo la ruta `/P1-ejb-cliente`

No hubo ningún problema en las pruebas y comprobamos el estado de la base de datos a cada paso para asegurarnos de que funcionaba correctamente.

The screenshot shows a web browser window with the address bar displaying `10.1.3.2:8080/P1-ejb-cliente/procesapago`. Below the browser, a terminal window titled `4. ssh si2@10.1.3.1 (ssh)` displays the output of a payment process. On the left, text indicates the transaction details: `idTransaccion: 1`, `idComercio: 1`, `importe: 23.0`, `codRespuesta: 000`, and `idAutorizacion: 4`. A blue link `Volver al comercio` is also present. The terminal window shows a table with the following data:

idautorizacion	idtransaccion	codrespuesta	importe	idcomercio	numerotarjeta	fecha
4	1	000	23	1	1111 2222 3333 4444	2015-03-04 15:32:35.265286

Figura 2: Muestra de un pago realizado a través de EJBs

## Ejercicio 5

Modificar la aplicación VISA para soportar el campo saldo:

Archivo TarjetaBean.java:

- Añadir el atributo saldo y sus métodos de acceso:

```
private double saldo;
```

Archivo VisaDAOBean.java:

- Importar la definición de la excepción `EJBException` que debe lanzar el servlet para indicar que se debe realizar un rollback:

```
1 import javax.ejb.EJBException;
```

- Declarar un prepared statement para recuperar el saldo de la tarjeta de la base de datos.
- Declarar un prepared statement para insertar el nuevo saldo calculado en la base de datos.
- Modificar el método `realizaPago` con las siguientes acciones:
  - Recuperar el saldo de la tarjeta a través del prepared statement declarado anteriormente.
  - Comprobar si el saldo es mayor o igual que el importe de la operación. Si no lo es, retornar denegando el pago (`idAutorizacion=null` y `pago retornado=null`)
  - Si el saldo es suficiente, decrementarlo en el valor del importe del pago y actualizar el registro de la tarjeta para reflejar el nuevo saldo mediante el prepared statement declarado anteriormente.
  - Si lo anterior es correcto, ejecutar el proceso de inserción del pago y obtención del `idAutorizacion`, tal como se realizaba en la práctica anterior (este código ya debe estar programado y no es necesario modificarlo).
  - En caso de producirse cualquier error a lo largo del proceso (por ejemplo, si no se obtiene el `idAutorizacion` porque la transacción está duplicada), lanzar una excepción `EJBException` para retornar al cliente.
- Modificar el servlet `ProcesaPago` para que capture la posible interrupción `EJBException` lanzada por `realizaPago`, y, en caso de que se haya lanzado, devuelva la página de error mediante el método `enviaError` (recordar antes de retornar que se debe invalidar la sesión, si es que existe).

- Archivo TarjetaBean.java

Además de añadir saldo como atributo hemos añadido los métodos `getSaldo` y `setSaldo`.

- Archivo VisaDAOBean.java

Hemos creado las dos consultas y hemos modificado el método `realizaPago` para que ejecute las dos consultas anteriores usando los *prepared statements*.

Al contrario que las otras consultas que se ejecutan en `realizaPago`, estas no dan la opción de no usar el *prepared*, ya que solo hemos construido ese tipo de query. El resto de queries siguen utilizando ambos métodos.

Para lanzar la excepción en caso de que algo falle hemos añadido un `String` que explica la razón por la que ha fallado y se los mandamos al cliente junto con la `EJBException`.

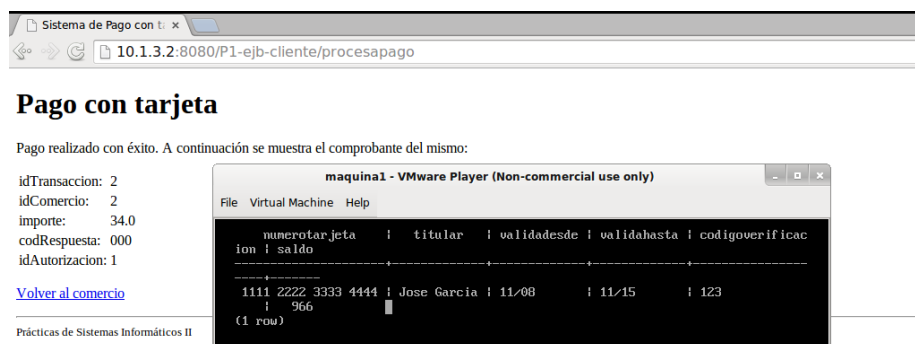
- Archivo ProcesaPago

Hemos capturado la excepción con un bloque `try / catch` que llama a `enviaError`.

## Ejercicio 6

Desplegar y probar la nueva aplicación creada.

- Probar a realizar pagos correctos. Comprobar que disminuye el saldo de las tarjetas sobre las que realice operaciones. Añadir a la memoria las evidencias obtenidas.
- Realice una operación con identificador de transacción y de comercio duplicados. Compruebe que el saldo de la tarjeta especificada en el pago no se ha variado.



The screenshot shows a web browser window with the address `10.1.3.2:8080/P1-ejb-cliente/procesapago`. The page title is "Sistema de Pago con t...". Below the browser window, there is a section titled "Pago con tarjeta" with the text "Pago realizado con éxito. A continuación se muestra el comprobante del mismo:". To the left of the terminal window, the following transaction details are listed:

```
idTransaccion: 2
idComercio: 2
importe: 34.0
codRespuesta: 000
idAutorizacion: 1
```

Below these details is a link [Volver al comercio](#). The terminal window, titled "maquina1 - VMware Player (Non-commercial use only)", shows a table with the following data:

numerosalida	titular	validadesde	validahasta	codigoverificac
1111 2222 3333 4444	Jose Garcia	11/08	11/15	123

The terminal also shows the text "(1 row)" and "(1 row)".

Figura 3: Pago realizado sobre la base de datos recién inicializada (Todas las tarjetas tenían saldo 1000)

## Pago con tarjeta

Fallo al insertar el pago

Prácticas de Sistemas Informáticos II

Figura 4: Resultado de un intento de pago con la misma id de transacción y de comercio

## Ejercicio 7

En la máquina virtual donde se encuentra el servidor de aplicaciones (10.X.Y.2), declare manualmente la factoría de conexiones empleando la consola de administración, tal y como se adjunta en la **Figura 4**.

Creamos manualmente a través de la interfaz de administración de Glassfish la factoría de mensajes aunque dudamos un poco al no coincidir completamente los nombres de los campos que hay en la captura de pantalla del enunciado con lo que veíamos nosotros. En concreto cambiaba la etiqueta del campo *Pool Name* por *JNDI Name*. Pudimos comprobar en ejercicios posteriores que era lo mismo ya que las referencias funcionaban.

**New JMS Connection Factory**  
The creation of a new Java Message Service (JMS) connection factory also creates a connector connection pool for the

**General Settings**

JNDI Name: *	jms/VisaConnectionFactory
Resource Type:	javax.jms.QueueConnectionFactory
Description:	Factoría de conexiones a la cola de pagos
Status:	<input checked="" type="checkbox"/> Enabled

## Ejercicio 8

En la máquina virtual donde se encuentra el servidor de aplicaciones (10.X.Y.2), declare manualmente la conexión empleando la consola de administración, tal y como se adjunta en la **Figura 5**

Realizando tareas similares a las del ejercicio 7 no tuvimos ningún problema creando la cola de mensajes.



## Ejercicio 9

- Modifique el fichero `sun-ejb-jar.xml` para que el MDB conecte adecuadamente a su connection factory
- Incluya en la clase `VisaCancelacionJMSBean`:
  - Consulta SQL necesaria para actualizar el código de respuesta a valor 999, de aquella autorización existente en la tabla de pagos cuyo `idAutorizacion` coincida con lo recibido por el mensaje.
  - Consulta SQL necesaria para rectificar el saldo de la tarjeta que realizó el pago
  - Método `onMessage()` que implemente ambas actualizaciones. Para ello tome de ejemplo el código SQL de ejercicios anteriores, de modo que se use un prepared statement que haga bind del `idAutorizacion` para cada mensaje recibido.

- Archivo `sun-ejb-jar.xml`

Bastó añadir el siguiente fragmento dentro del `ejb` declarado con los datos especificados en el ejercicio 7.

```
<mdb-connection-factory>
    <jndi-name>jms/VisaConnectionFactory</jndi-name>
</mdb-connection-factory>
```

- Archivo `VisaCancelacionJMSBean`

Construimos las queries de manera que solo hiciera falta la `id` de autorización para realizar ambas (pese a ser una sobre la tabla de pagos y la otra sobre la de tarjetas). Además, tuvimos en cuenta que se podría dar el caso de intentar cancelar un pago ya cancelado, así que la query de actualizar el código de respuesta no realiza ninguna actualización en caso de que el código ya haya sido actualizado.

Esto concuerda con la comprobación de errores que realizamos dentro del método. Después de recibir `idAutorizacion` y colocar su valor en la query mediante un *prepared statement*, ejecutamos la query y comprobamos cuantas filas se actualizaron. En ningún caso será mayor que uno el número ya que una de las condiciones es de igualdad en la clave primaria. Así que puede ser 1, en cuyo caso ha actualizado un pago correctamente, o 0 en cuyo caso no ha actualizado ningún pago por no existir o por que ha sido cancelado previamente.

En caso de que se haya cancelado con este mensaje un pago, se procede al reembolso. El cual se efectúa con una query que añade el coste del pago al saldo de la tarjeta que lo ha realizado mediante dos *subqueries*.

## Ejercicio 10

Implemente ambos métodos en el cliente proporcionado. Deje comentado el método de acceso por JNDI. Indique en la memoria de prácticas qué ventajas podrían tener uno u otro método.

El método que hemos usado al final, obtención de recursos JMS estáticos, tiene la ventaja de, en algunos casos, ser más eficiente que el método dinámico ya que las anotaciones que declaran que va a tener esa variable en tiempo de ejecución pueden prepararse una vez, cuando se instancie la clase en el sistema al arrancar la aplicación, mientras que la búsqueda dinámica tiene que ser llamada explícitamente, lo cual es su desventaja. En cambio el método dinámico proporciona más flexibilidad al sistema permitiendo que el nombre de la factoría de conexiones o de la cola pueda ser obtenida dinámicamente (de un archivo de configuración, de variables de entorno, de la entrada del usuario, etc...) o incluso cambiada en algún momento, al contrario que el método estático, en el que los nombres de los recursos deben saberse en tiempo de compilación al estar incluidos dentro de una anotación `@Resource`.

## Ejercicio 11

Automatice la creación de los recursos JMS (cola y connection factory) en el `build.xml` y `jms.xml`. Para ello, indique en `jms.properties` los nombres de ambos y el Physical Destination Name de la cola de acuerdo a los valores asignados en los ejercicios 7 y 8. Recuerde también asignar las direcciones IP adecuadas a las variables **as.host.mdb** (`build.properties`), **as.host.server** y **as.host.client** (`jms.properties`).

Borre desde la consola de administración de Glassfish la *connectionFactory* y la cola creadas manualmente y ejecute:

```
cd P1-jms
ant todo
```

Compruebe en la consola de administración del Glassfish que, efectivamente, los recursos se han creado automáticamente. Revise el fichero `jms.xml` y anote en la memoria de prácticas cuál es el comando equivalente para crear una cola JMS usando la herramienta `asadmin`.

No hubo que realizar cambios en ninguno de los archivos `xml` mencionados ya que ambos importan los archivos de propiedades.

Todas las direcciones ip que tuvimos que asignar fueron las de la máquina 2 ya que la base de datos permanece intacta y aislada en la 1. Así que cambiamos las variables `as.host.mdb`, `as.host.server` y `as.host.client` a `10.1.3.2`.

Además, en el archivo `jms.properties` asignamos las variables `jms.factoryname`, `jms.name` y `jms.physname` a los valores `jms/VisaConnectionFactory`, `jms/VisaPagosQueue` y `VisaPagosQueue` respectivamente.

## Ejercicio 12

**Importante:** Detenga la ejecución del MDB con la consola de administración para poder realizar satisfactoriamente el siguiente ejercicio (check de 'Enabled' en Applications/P1-jms-mdb y guardar los cambios).

Modifique el cliente, `VisaQueueMessageProducer.java`, implementando el envío de `args[0]` como mensaje de texto (consultar los apéndices). Ejecute el cliente en el PC del laboratorio mediante el comando:

```
1 /usr/local/glassfish-4.0/glassfish/bin/appclient -targetserver
  10.X.Y.Z -client
  dist/clientjms/P1-jms-clientjms.jar idAutorizacion
```

Donde `10.X.Y.Z` representa la dirección IP de la máquina virtual en cuyo servidor de aplicaciones se encuentra desplegado el MDB. Para garantizar que el comando funcione correctamente es necesario fijar la variable

```
(web console->Configurations->server-config->Java Message
  Service->JMS Hosts->default.JMS.host)
```

que toma el valor "localhost" por la dirección IP de dicha máquina virtual. El cambio se puede llevar a cabo desde la consola de administración. Será necesario reiniciar el servidor de aplicaciones para que surja efecto. Verifique el contenido de la cola ejecutando:

```
1 /usr/local/glassfish-4.0/glassfish/bin/appclient -targetserver
  10.X.Y.Z -client
  dist/clientjms/P1-jms-clientjms.jar -browse
```

Indique la salida del comando e inclúyala en la memoria de prácticas. A continuación, volver a habilitar la ejecución del MDB y realizar los siguientes pasos:

- Realice un pago con la aplicación web
- Obtenga evidencias de que se ha realizado
- Cancelelo con el cliente
- Obtenga evidencias de que se ha cancelado y de que el saldo se ha rectificado

Como se indicó en el mensaje posterior a la publicación de la práctica, ejecutamos el cliente jms en una de las máquinas virtuales. En nuestro caso elegimos la 10.1.3.1, ya que de esta manera la cola no estaría en la misma máquina que en la que se ejecutaba el comando.

Por supuesto, también configuramos la variable `default.JMS.host` (poniendo `10.1.3.2`) en glassfish para permitir la conexión desde el exterior (ya que previamente estaba definida como `localhost`).

Al haber desactivado el `jms-mdb`, al ejecutar el primer comando, colocamos

un mensaje en una cola de la que no hay nadie sacando mensajes. Esto nos permite, momentos después ejecutar el cliente `jms` otra vez, esta vez pidiendo que nos liste los mensajes en la cola. Con lo que pudimos comprobar que nuestro código para enviar mensajes funcionaba correctamente.

Como habíamos enviado el número 1 como mensaje, aprovechamos que en cuanto arrancáramos `jms-mdb` lo iba a procesar, y efectuamos un pago, que al ser el primero, tomaría la `id` de autorización 1. Efectivamente, cuando arrancamos el procesador de mensajes, se puso manos a la obra e invalidó el pago que habíamos realizado, devolviendo el dinero a la cuenta.

```
visa=# SELECT * FROM pago;
 idautorizacion | idtransaccion | codrespuesta | importe | idcomercio | numerotarjeta | fecha
-----+-----+-----+-----+-----+-----+-----
1 | 1 | 000 | 12 | 1 | 1111 2222 3333 4444 | 2015-03-04 17:23:01.783263
(1 row)
```

- (a) Query para comprobar que el pago se había realizado correctamente y ver que su identificador de autorización es, como esperábamos, 1

```
visa=# SELECT * FROM tarjeta WHERE numerotarjeta LIKE '1111%';
 numerotarjeta | titular | validadesde | validahasta | codigoverificacion | saldo
-----+-----+-----+-----+-----+-----
1111 2222 3333 4444 | Jose García | 11/08 | 11/15 | 123 | 988
(1 row)
```

- (b) Query para comprobar que el saldo de la tarjeta ha bajado como era esperado.

```
visa=# SELECT * FROM pago;
 idautorizacion | idtransaccion | codrespuesta | importe | idcomercio | numerotarjeta | fecha
-----+-----+-----+-----+-----+-----+-----
1 | 1 | 999 | 12 | 1 | 1111 2222 3333 4444 | 2015-03-04 17:23:01.783263
(1 row)
```

- (c) Una vez realizada la cancelación vemos como el resultado del pago ha cambiado a 999

```
visa=# SELECT * FROM tarjeta WHERE numerotarjeta LIKE '1111%';
 numerotarjeta | titular | validadesde | validahasta | codigoverificacion | saldo
-----+-----+-----+-----+-----+-----
1111 2222 3333 4444 | Jose García | 11/08 | 11/15 | 123 | 1000
(1 row)
```

- (d) Y también el saldo del cliente vuelve a la cantidad inicial: 1000

Figura 5