UAM UNIVERSIDAD AUTONOMA DE MADRID		Escuela Politécnica Superior Ingeniería Informática Prácticas de Sistemas Informáticos 2			
Grupo	2401	**Práctica**	1b	**Fecha**	4/03/2015
Alumno		Kasner Tourné, Cristina			
Alumno		Guridi Mateos, Guillermo			

Cuestión 1

`VisaDAOLocal.java`

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 Maquina 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

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

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

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

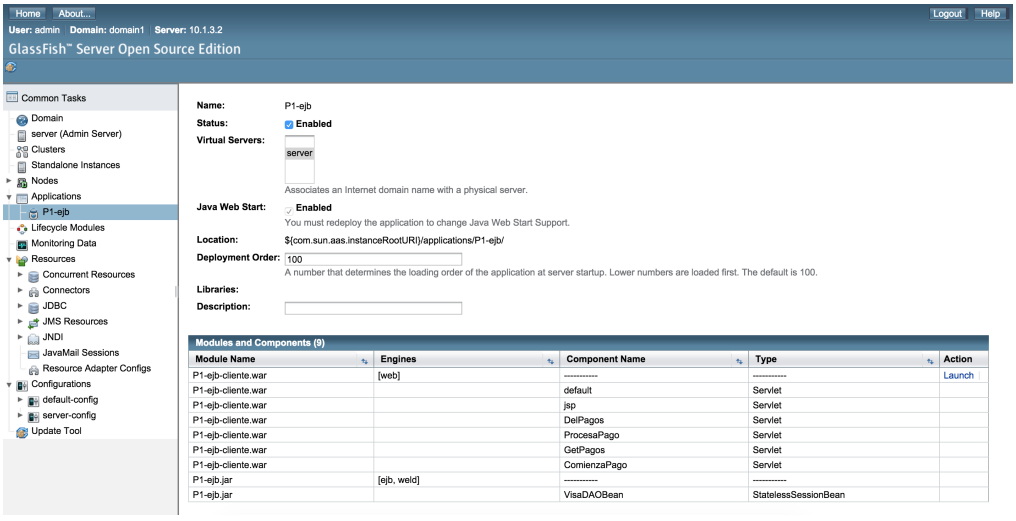

Figura 1: Comprobar en la administración de Glassfish el despliegue de la aplicación

Ejercicio 4

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.

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

Ejercicio 5

- 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 querys 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

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

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

Ejercicio7

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.

Ejercicio8

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

Ejercicio9

- 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 querys 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 actualizacion en caso de que el código ya haya sido actualizado.

  Esto concuerda con la comprobación de errores que realizamos dentro del método. Despues de recibir `idAutorizacion` y colocar su valor en la query mediante un *prepared statement*, ejecutamos la query y comprobamos cuantas filas a actualizado. 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 efectua con una query que añade el coste del pago al saldo de la tarjeta que lo ha realizado mediante dos *subquerys*.

Ejercicio10

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, mientra que la búsqueda dinámica tiene que ser llamada explicitamente, 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`.

Ejercicio11

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. (EXPLICAR PHYSNAME).

Ejercicio12

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 nestro código para enviar mensajes funcionaba correctamente.

Como habíamos enviado el número 1 como mensaje, aprovechamos que en cuanto arrancáramos `jms-mdb` lo íba 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.

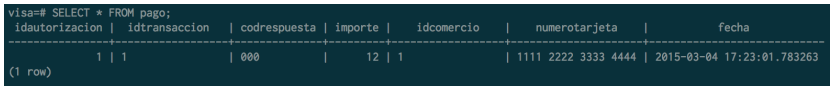

Figura 5: Query para comprobar que el pago se había realizado correctamente y ver que su identificador de autorización es, como esperábamos, 1

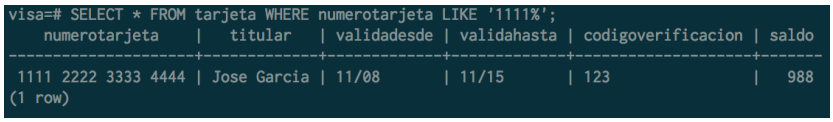

Figura 6: Query para comprobar que el saldo de la tarjeta ha bajado como era esperado.

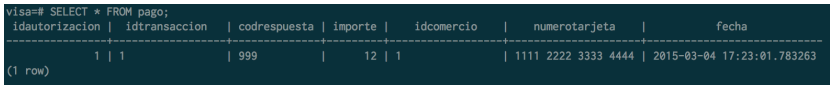

Figura 7: Una vez realizada la cancelación vemos como el resultado del pago ha cambiado a 999

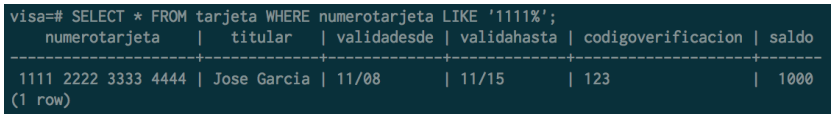

Figura 8: Y también el saldo del cliente vuelve a la cantidad inicial: 1000