|  | | Escuela Politécnica Superior  Ingeniería Informática  Prácticas de Sistemas Informáticos 2 | | | |
| --- | --- | --- | --- | --- | --- |
| **Grupo** | **2311** | **Práctica** | 1A | **Fecha** | 21/02/2022 |
| **Alumno/a** | | Cerrato, Sánchez, Daniel | | | |
| **Alumno/a** | | Garitagoitia, Romero, David T. | | | |

# Práctica 1A: Arquitectura de Java EE

## Ejercicio número 1:

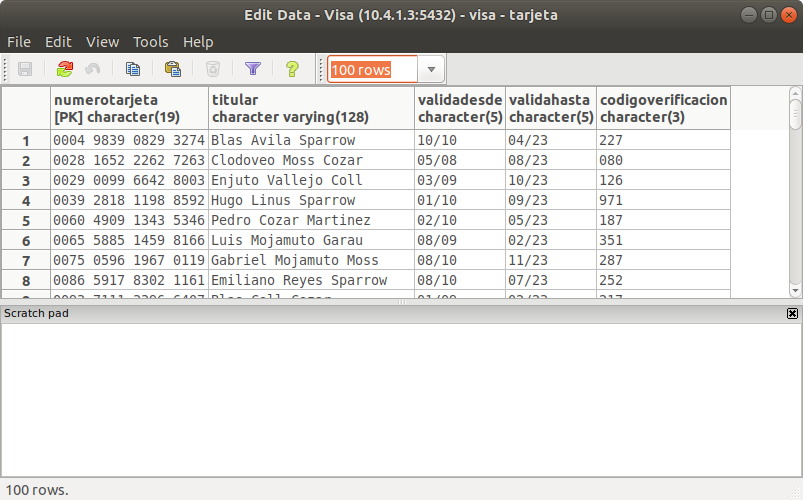
**Modifique los ficheros que considere necesarios en el proyecto para que se despliegue tanto la aplicación web como la base de datos contra la dirección asignada a la pareja de prácticas.**

Se han modificado los ficheros “build.properties” y “postgres.properties” para cambiar los valores de as.host, db.host y db.client.host con el valor de la IP obtenida para la máquina virtual de clase (10.250.1.86).

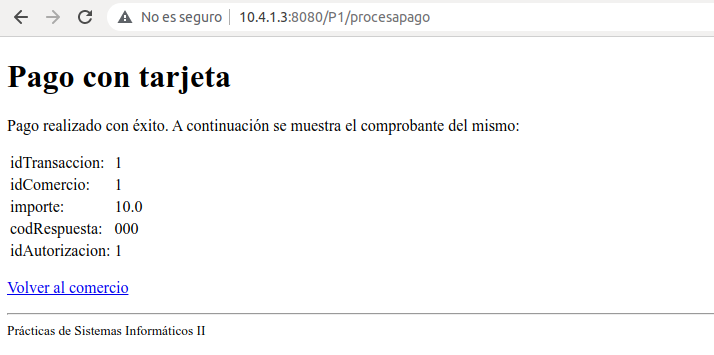
**Realice un pago contra la aplicación web empleando el navegador en la ruta** [**http://10.X.Y.Z:8080/P1**](http://10.x.y.z:8080/P1)

Lo primero que haremos será ver la información de las tarjetas desde el pgadmin para saber los

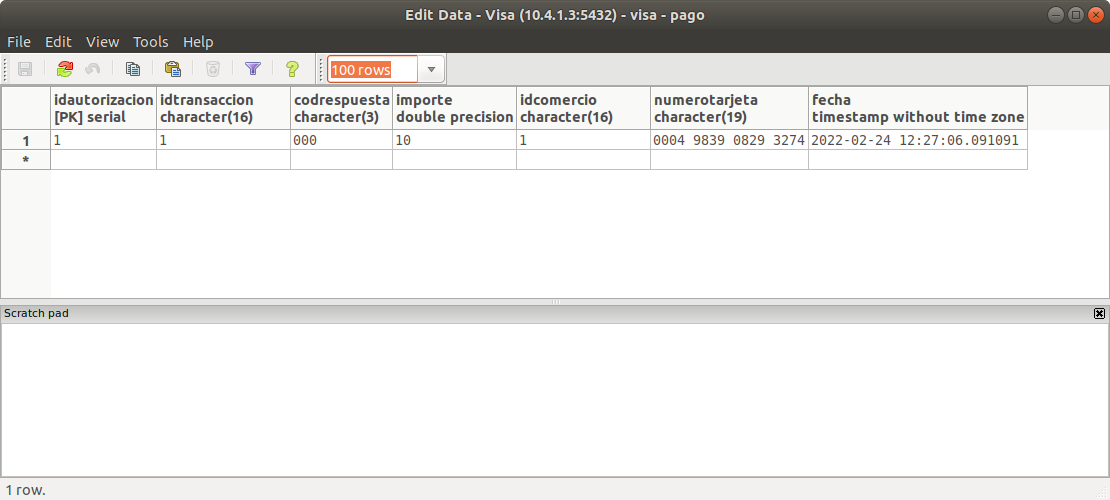
datos de alguna tarjeta válida, para ello nos conectamos a la base de datos visa con user alumnodb y la contraseña establecida.

Con esta información podemos ir a la ruta (emplearemos los del primer resultado).

****

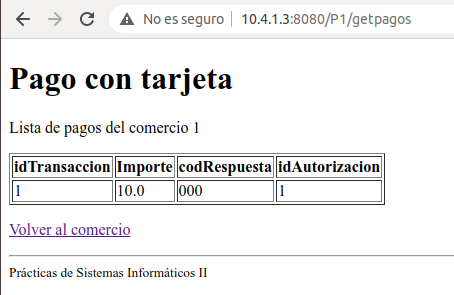


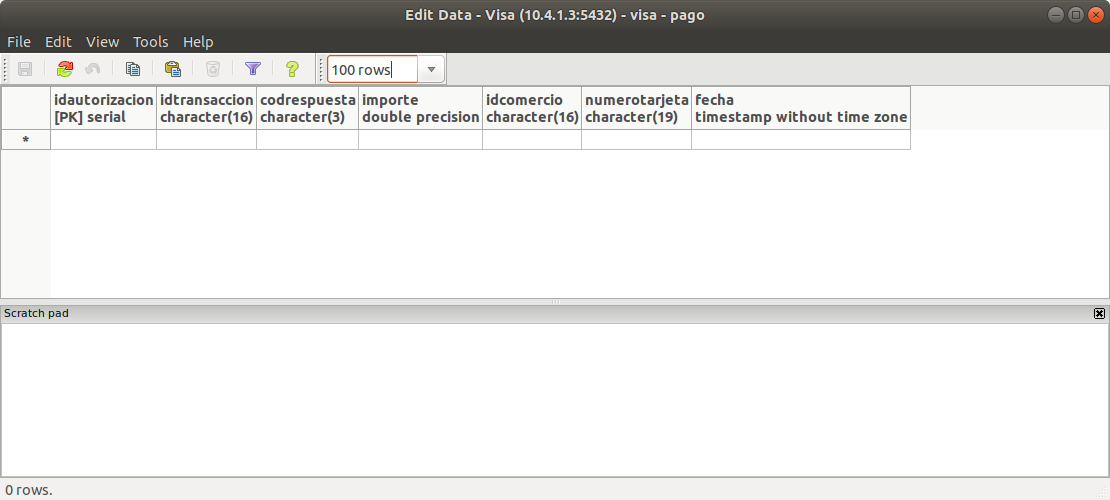
Como podemos ver, la transacción se realiza con éxito, esto se puede comprobar efectivamente al acceder a la tabla de pagos de la base de datos.



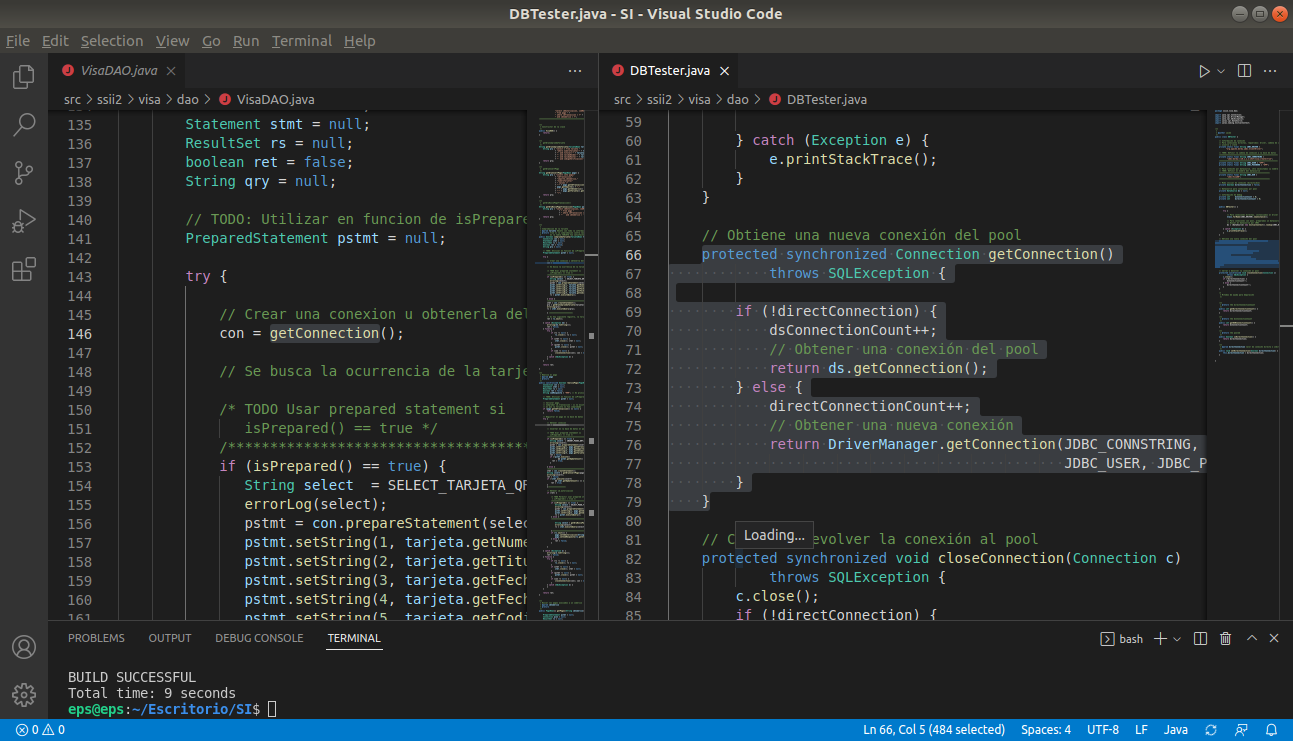
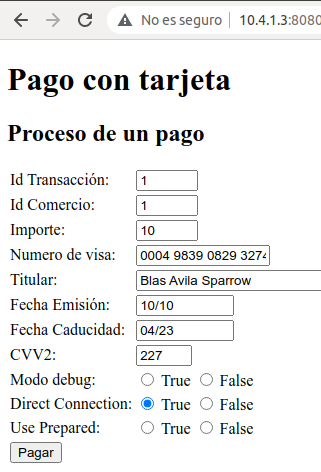
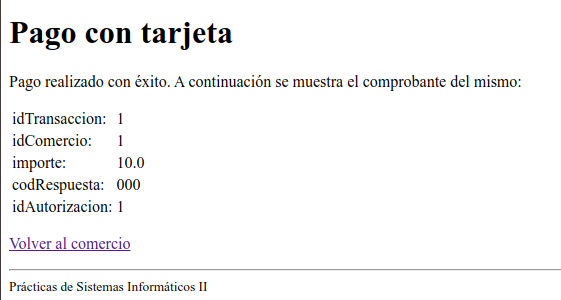
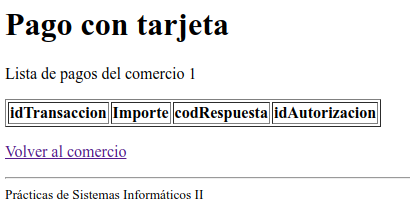
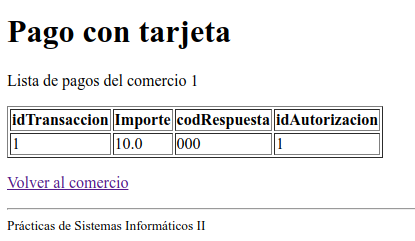
**Acceda a la página de pruebas extendida, http://10.X.Y.Z:8080/P1/testbd.jsp. Compruebe que la funcionalidad de listado de y borrado de pagos funciona correctamente. Elimine el pago anterior.**

Al acceder a la url e indicar que buscamos el comercio 1, podemos ver como efectivamente aparece dicho pago y el borrado se realiza correctamente.



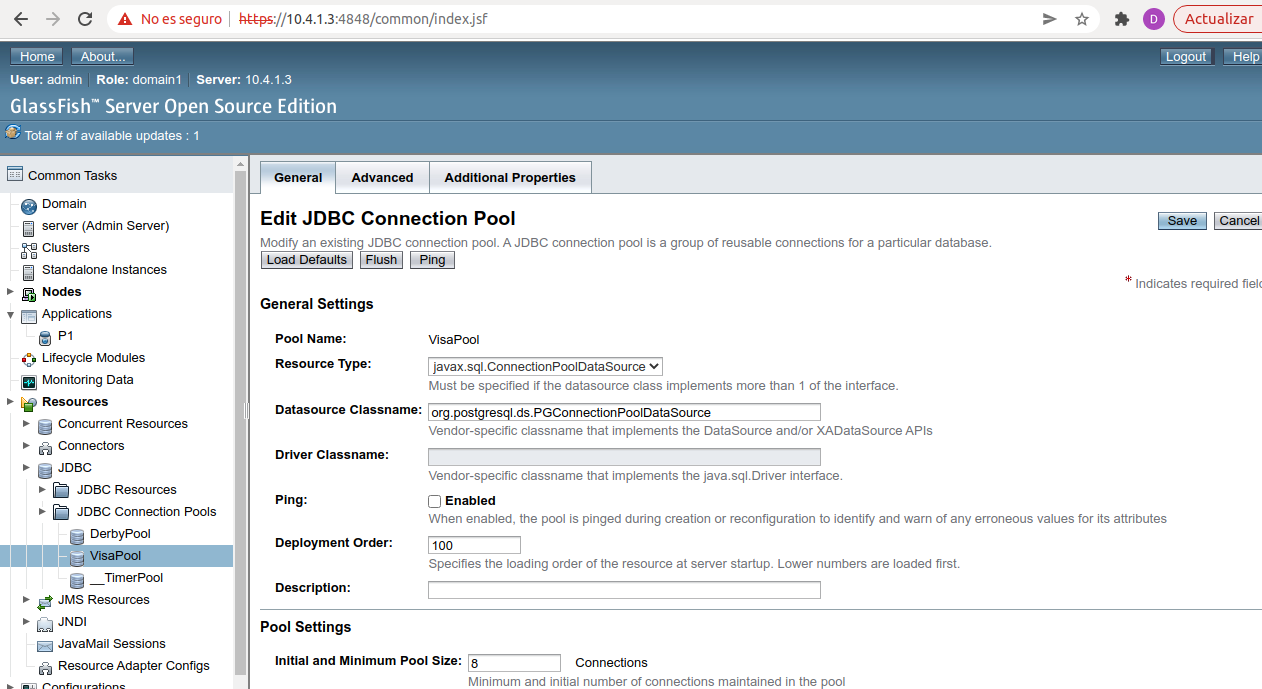
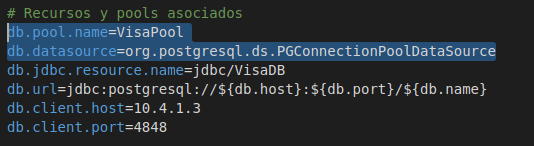
****

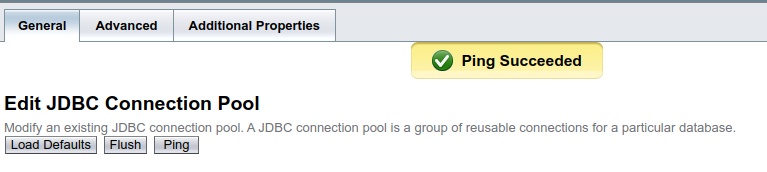
## Ejercicio número 2:

* 1. **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 posgresql, 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**
  2. Para este ejercicio debemos modificar alguna información para la conexión directa, para ello vamos al apéndice 10 sobre acceso a datos.
  3. El apartado 1 del apéndice habla sobre la conexión directa:
     1. Para conectar de forma directa lo primero que debemos hacer es instanciar el driver JDBC del gestor de bases de datos. Para ello bastará el siguiente fragmento de código (ver la clase DBTester.java): Class.forName("org.postgresql.Driver").newInstance(); A continuación, podemos utilizar el método estático de la clase DriverManager para obtener nuevas conexiones, para lo cual requeriremos conocer la cadena de conexión JDBC, usuario y clave: Connection c = DriverManager.getConnection ( "jdbc:postgresql://host:puerto/nombreBD”, “usuario”, “clave” );
  4. Como podemos ver en el código, la clase VisaDao llama a la función getConnection() que hereda de DBTester en ella se puede ver una separación para el caso de conexión directa en cuyo caso se vé como emplea unas string para la conexión, bastará con modificar esas strings como se indica en el apéndice.
  5. private static final String JDBC\_DRIVER =
  6. "org.postgresql.Driver";
  7. // TODO: Definir la cadena de conexion a la base de datos
  8. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
  9. private static final String JDBC\_CONNSTRING =
  10. "jdbc:postgresql://10.4.1.3:5432/visa";
  11. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
  12. private static final String JDBC\_USER = "alumnodb";
  13. private static final String JDBC\_PASSWORD = "\*\*\*\*";
      1. Vamos a comprobar que se puede realizar transacción adecuadamente.
      2. 
  14. Como se puede ver el pago se realiza con éxito sin ningún problema, cuando accedemos a la lista de pagos
  15. Después, procedemos al borrado de pagos idéntico al ejercicio anterior y efectivamente se borra con éxito.
  16. 

## Ejercicio número 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.**

Para el ping simplemente pulsamos en el botón de ping:



Revisamos los valores del pool setings:

Pool Settings

Initial and Minimum Pool Size: 8

Maximum Pool Size: 32

Pool Resize Quantity: 2

Idle Timeout: 300

Max Wait Time: 60000

En cuanto al impacto que pueden tener estos valores, el primero de ellos se asocia al número mínimo de hilos y por tanto los hilos con los que se inicializa lo que afecta al rendimiento de la app cuando se inicia.

Si es muy elevado el servicio tardará más en abrirse, pero si es muy bajo no ofrecerá el servicio de varias conexiones preparadas para la petición del cliente.

Por otro lado el máximo en caso de ser muy alto, cargará en exceso al servidor de peticiones simultáneas lo cual repercute directamente en el rendimiento pero si es muy bajo no podrá con todas las peticiones generando una cola de peticiones.

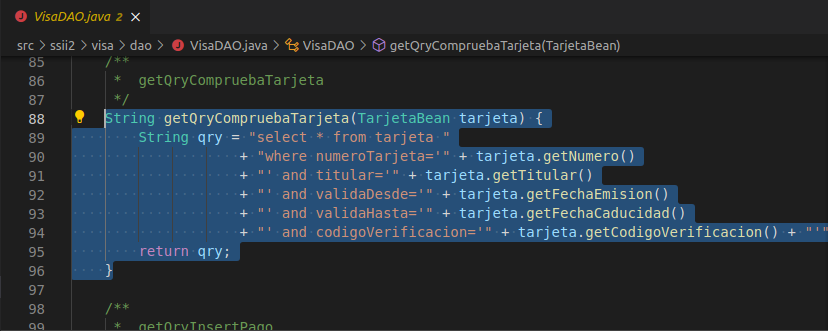
El pool resize quantity indica las conexiones que se eliminan al pasar el timeout por lo que un número muy elevado puede originar problemas.

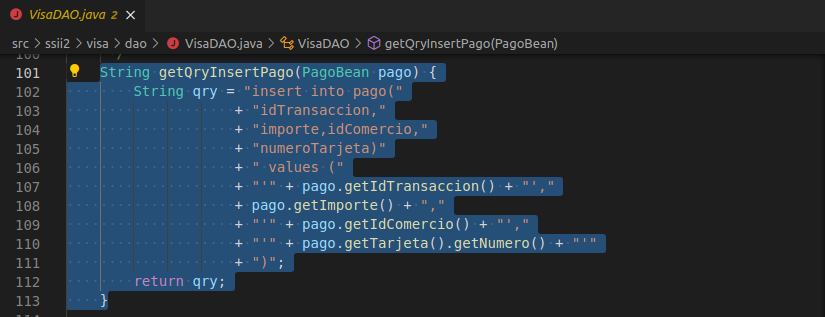
El timeout si es muy alto se tendrán en uso recursos que no son necesarios lo que genera una perdida en el rendimiento pero si es muy bajo se cerraran conexiones generando errores cuando se tarde algo en seguir con la petición.

Por último el max wait es el tiempo máximo que espera un cliente, si es muy alto daría problemas con el timeout y si es muy bajo originaría que se cortases consultas de conexión algo más lentas.

## Ejercicio número 4:

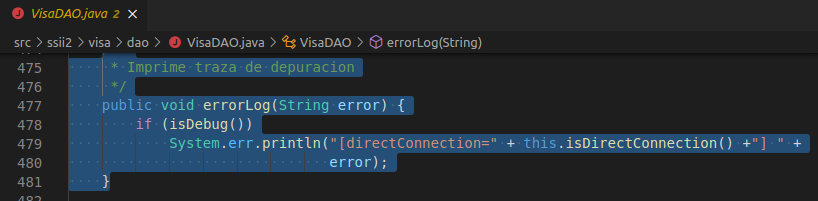
**Localice los siguientes fragmentos de código SQL dentro del proyecto proporcionado (P1-base) correspondientes a los siguientes procedimientos: - Consulta de si una tarjeta es válida. - Ejecución del pago. Incluya en la memoria de prácticas dichas consultas**

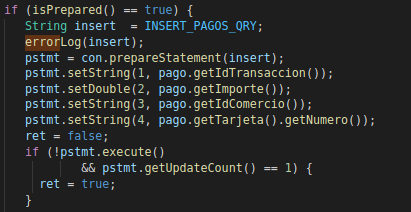
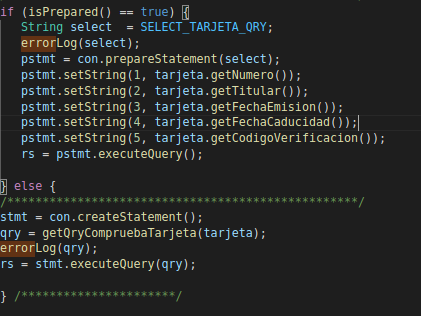
****



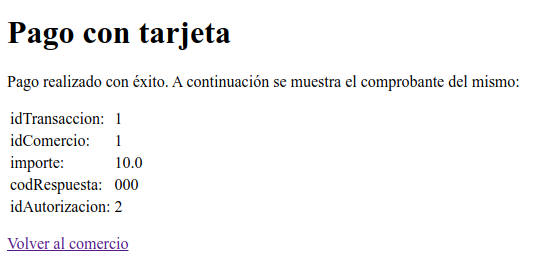
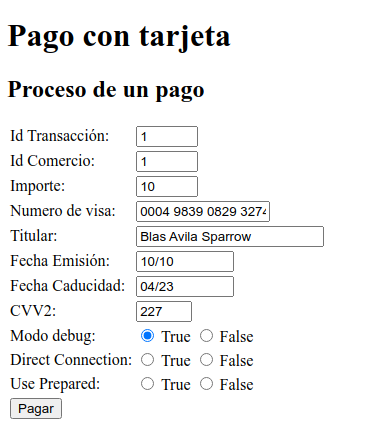
## Ejercicio número 5:

**Edite el fichero VisaDAO.java y localice el método errorLog. Compruebe en qué partes del código se escribe en log utilizando dicho método. 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 capturas de pantalla del log del servidor, accediento a él tanto desde el .terminal como del portal web.**

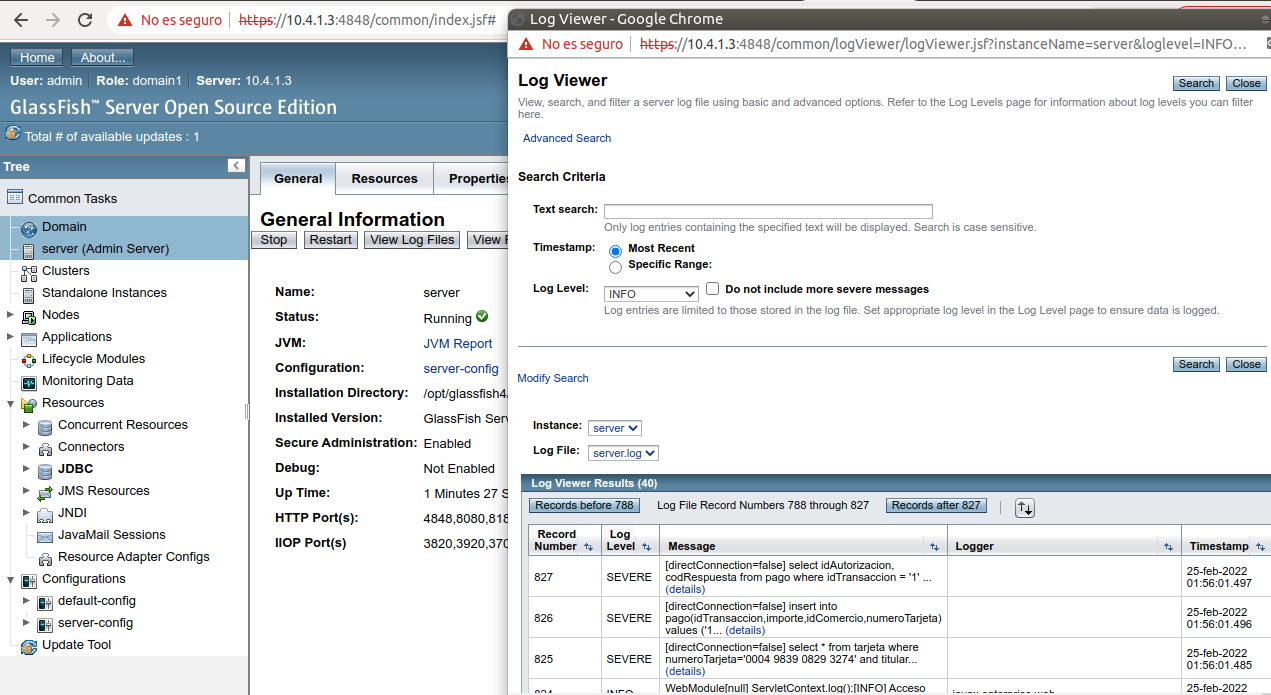
 El método errorLog escribe en el log en caso de estar activo el debug, se llama en partes del código como estas para informar del flujo de ejecución o para las excepciones capturadas del try catch.



Procedemos pues a realizar un pago en el modo debug



Se realiza el pago con éxito así que accedemos a la zona de admin para ver el log.



Aquí se pueden ver perfectamente los logs de comprobar tarjeta.

Para poder verlo desde terminal basta con un cat del fichero de logs

## Ejercicio número 6:

**Realícense las modificaciones necesarias en VisaDAOWS.java para que implemente de manera correcta un servicio web. Los siguientes métodos y todos sus parámetros deberán ser publicados como métodos del servicio. compruebaTarjeta() realizaPago() isDebug() / setDebug() (Nota: VisaDAO.java contiene dos métodos setDebug que reciben distintos argumentos. Solo uno de ellos podrá ser exportado como servicio web)3 . isPrepared() / setPrepared() Deberemos publicar así mismo: isDirectConnection() / setDirectConnection() que son métodos heredados de la clase DBTester 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?**

Para esto necesitamos modificar la clase VisaDAOWS para ello comenzamos colocando los nuevos import y anotaciones:

import javax.jws.WebMethod;

import javax.jws.WebParam;

import javax.jws.WebService;

@WebService(): indica que la clase Java implementa un servicio web.

@WebMethod(operationName = "nombreMetodo"): indica que el método Java que le sigue será exportado como un método público del servicio.

@WebParam(name = "nombreArgumento"): indica que el argumento que viene a continuación también es un argumento del método equivalente del servicio.

El webService simplemente lo colocaremos al inicio de la clase.

WebMethod lo colocaremos en cada método como puede ser compruebaTarjeta y los parámetros de estos iran con webParam.

finalmente para los isDirectConnection() empleamos los override y ya podemos ponerlos como webMethod:

@WebMethod(operationName = "isDirectConnection")

@Override

public boolean isDirectConnection() {

return super.isDirectConnection();

}

/\*\*

\* @param directConnection valor de conexión directa o indirecta

\*/

@WebMethod(operationName = "setDirectConnection")

@Override

public void setDirectConnection(@WebParam(name = "directConnection") boolean directConnection) {

super.setDirectConnection(directConnection);

}

Es necesario modificar el retorno de realizaPago() ya que ahora no basta únicamente con un booleano si no que se desea conocer información del pago, modificamos el retorno de los booleanos.

## Ejercicio número 7:

**Despliegue el servicio con la regla correspondiente en el build.xml. Acceda al WSDL remotamente con el navegador e inclúyalo en la memoria de la práctica (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 5 . Conteste a las siguientes preguntas: ¿En qué fichero están definidos los tipos de datos intercambiados con el webservice? ¿Qué tipos de datos predefinidos se usan? ¿Cuáles son los tipos de datos que se definen? ¿Qué etiqueta está asociada a los métodos invocados en el webservice? ¿Qué etiqueta describe los mensajes intercambiados en la invocación de los métodos del webservice? ¿En qué etiqueta se especifica el protocolo de comunicación con el webservice? ¿En qué etiqueta se especifica la URL a la que se deberá conectar un cliente para acceder al webservice?**

El fichero wdsl se encuentra en VisaDAOWSService-wdsl.xml

**wsdl.url=http://${as.host.server}:8080/${nombre}-ws/VisaDAOWSService?wsdl**

Los datos de intercambio se definen en el fichero XSD

Los tipos de datos predefinidos son xs:boolean, xs:string y xs:double

Los tipos son clases y métodos del sistema que se pueden llamar

Los métodos invocados por el web service utilizan la etiqueta <operation> mientras que el de los mensajes <message>

El protocolo de comunicación se especifica en <binding>

La url se especifica en <soap:address>

## Ejercicio número 8:

**Realícese las modificaciones necesarias en ProcesaPago.java para que implemente de manera correcta la llamada al servicio web mediante stubs estáticos. Téngase en cuenta que: El nuevo método realizaPago() ahora no devuelve un boolean, sino el propio objeto Pago modificado. Las llamadas remotas pueden generar nuevas excepciones que deberán ser tratadas en el código cliente. Incluye en la memoria una captura con dichas modificaciones.**

Para este apartado debemos modificar algunos aspectos de ProcesaPago.java

Para empezar pondremos los nuevos imports.

Las funciones que requieran de VisaDAOWS (processRequest) las modificaremos para emplear el siguiente orden.

**VisaDAOWSService service = new VisaDAOWSService();**

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

Para tener en cuenta que ahora realizaPago devuelve el pago, en la comprobación comprobamos que no sea null.

Como las llamadas remotas pueden generar excepciones bastará con envolverlas con un try catch.

y para poder facilitar el debugeo empleamos la funcion errorLog para que imprima el nombre de la excepción.

## Ejercicio número 9:

**Modifique la llamada al servicio para que la ruta (URL) 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.**

Lo primero que hacemos para este ejercicio es leer el apéndice 15.1 cómo se nos indica en el enunciado, en dicho apéndice se muestra lo siguiente.

El contexto es único e independiente de las sesiones de los usuarios. El contexto se obtiene mediante getServletContext() y permite a un servlet:

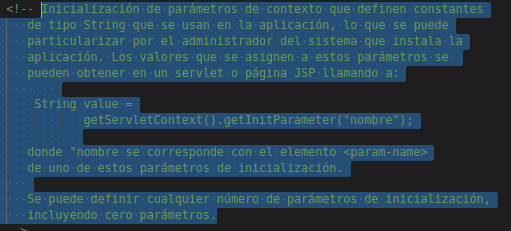
Pasar información a otros servlet: Cada servlet puede añadir objetos a ese contexto que son accesibles por el resto de los servlet.

Para ello, al igual que los dos casos anteriores, se crea un nuevo atributo que consta de un par nombre-valor donde el valor se corresponde con el objeto que se quiere añadir.

Escribir en los logs del servidor. Obtener parámetros de inicialización: Se pueden acceder a los parámetros de inicialización especificados en el archivo web.xml mediante el método getServletContext().getInitParameter("nombre-parámetro").

Como se nos indica es necesario emplear el comando

getServletContext().getInitParameter("nombre-parámetro").

**Leyendo las anotaciones del web.xml obtenemos la siguiente información:**

**Port tanto escribimos lo siguiente:**

**<context-param>**

**<param-name>dir</param-name>**

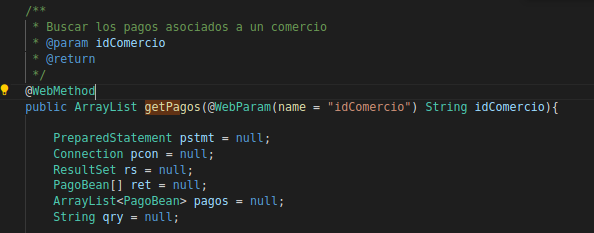
**<param-value>http://10.4.1.3:8080/P1-ws-ws/VisaDAOWSService</param-value>**

**</context-param>**

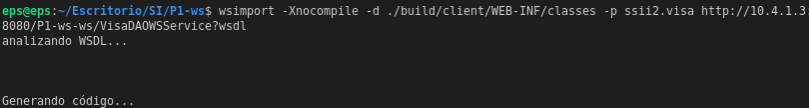
**y ahora simplemente modificamos el código de la siguiente forma:**

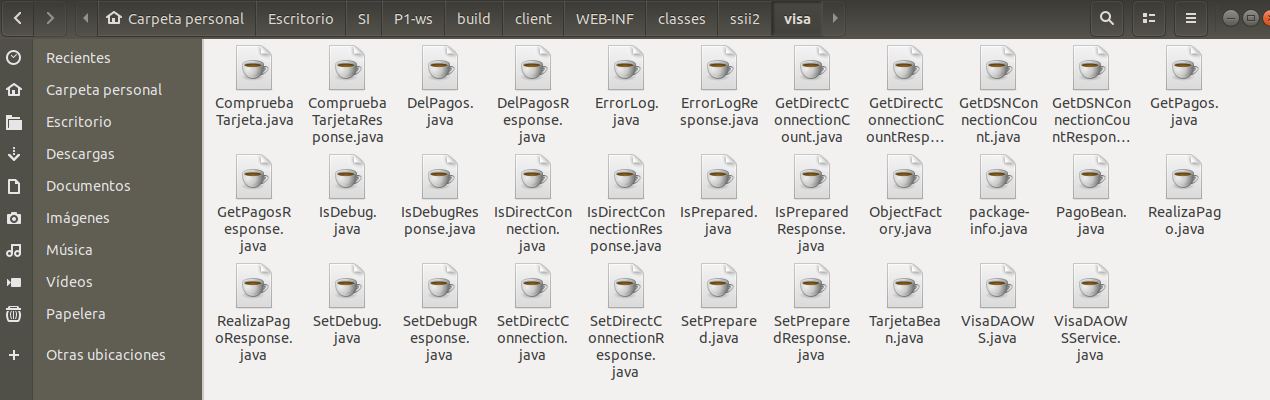
****

## Ejercicio número 10:

* 1. **Siguiendo el patrón de los cambios anteriores, adaptar las siguientes clases cliente para que toda la funcionalidad de la página de pruebas testbd.jsp se realice a través del servicio web. Esto afecta al menos a los siguientes recursos: Servlet DelPagos.java: la operación dao.delPagos() debe implementarse en el servicio web. Servlet GetPagos.java: la operación dao.getPagos() debe implementarse en el servicio web. Tenga en cuenta que no todos los tipos de datos son compatibles con JAXB (especifica como codificar clases java como documentos XML), por lo que es posible que tenga que modificar el valor de retorno de alguno de estos métodos. Los apéndices contienen más información. Más específicamente, se tiene que modificar la declaración actual del método getPagos(), que devuelve un PagoBean[], por: public ArrayList getPagos(@WebParam(name = "idComercio") String idComercio) Hay que tener en cuenta que la página listapagos.jsp espera recibir un array del tipo PagoBean[]. Por ello, es conveniente, una vez obtenida la respuesta, convertir el ArrayList a un array de tipo PagoBean[] utilizando el método toArray() de la clase ArrayList. Incluye en la memoria una captura con las adaptaciones realizadas.**
  2. Para esto, los métodos delPagos() y getPagos() del fichero visaDAOWS.java deben tener las etiquetas correspondientes como ya hemos visto en ejercicios anteriores**.**
  3. ****

## Ejercicio número 11:

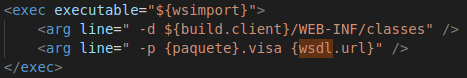
* 1. **Realice una importación manual del WSDL del servicio sobre el directorio de clases local. Anote en la memoria qué comando ha sido necesario ejecutar en la línea de comandos, qué clases han sido generadas y por qué. Téngase en cuenta que el servicio debe estar previamente desplegado.**
  2. ****



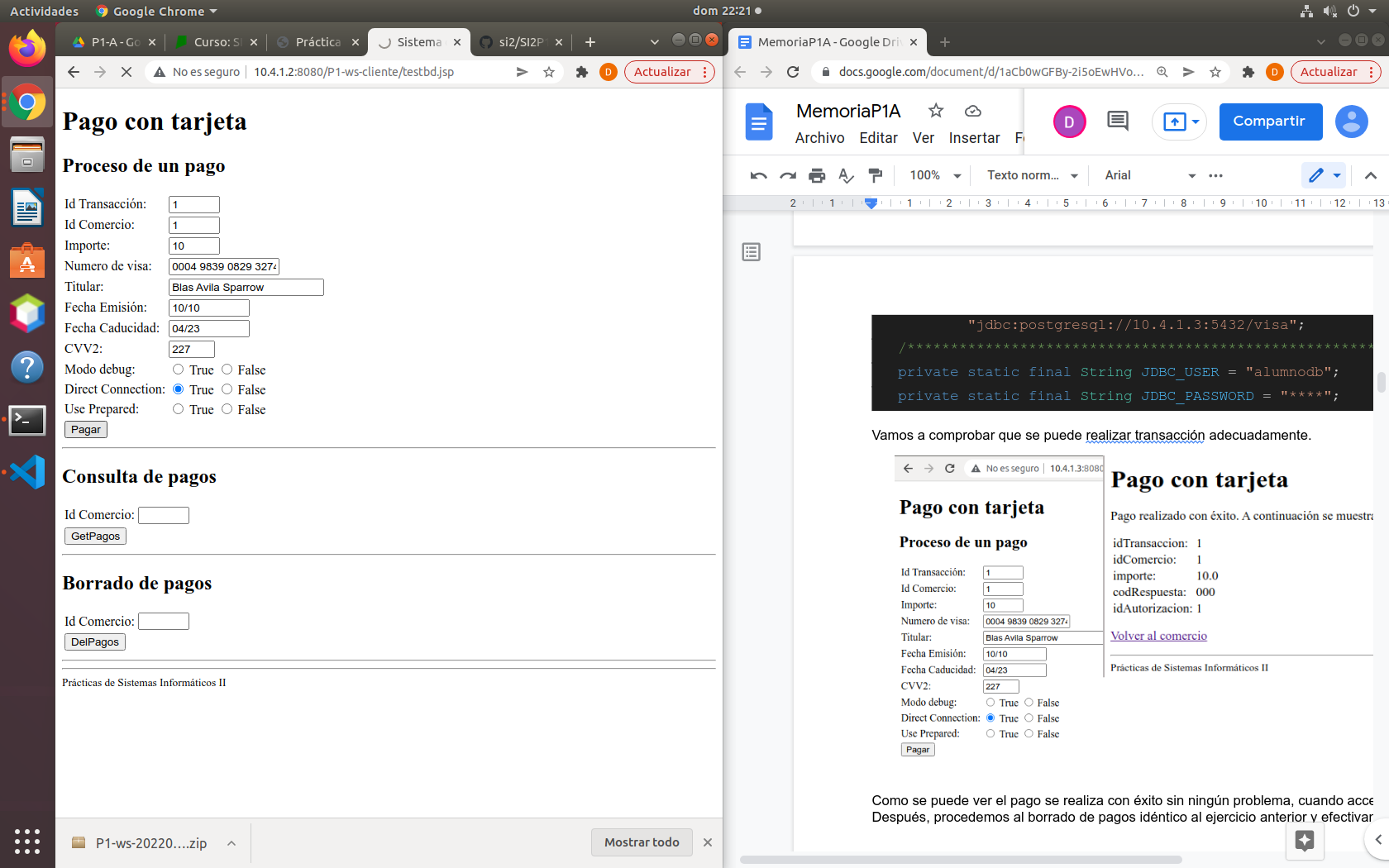
wsimport -Xnocompile -d ./build/client/WEB-INF/classes -p ssii2.visa <http://10.4.1.3:8080/P1-ws-ws/VisaDAOWSService?wsdl>

Se han generado todas las que tienen las etiquetas de web mostradas anteriormente.

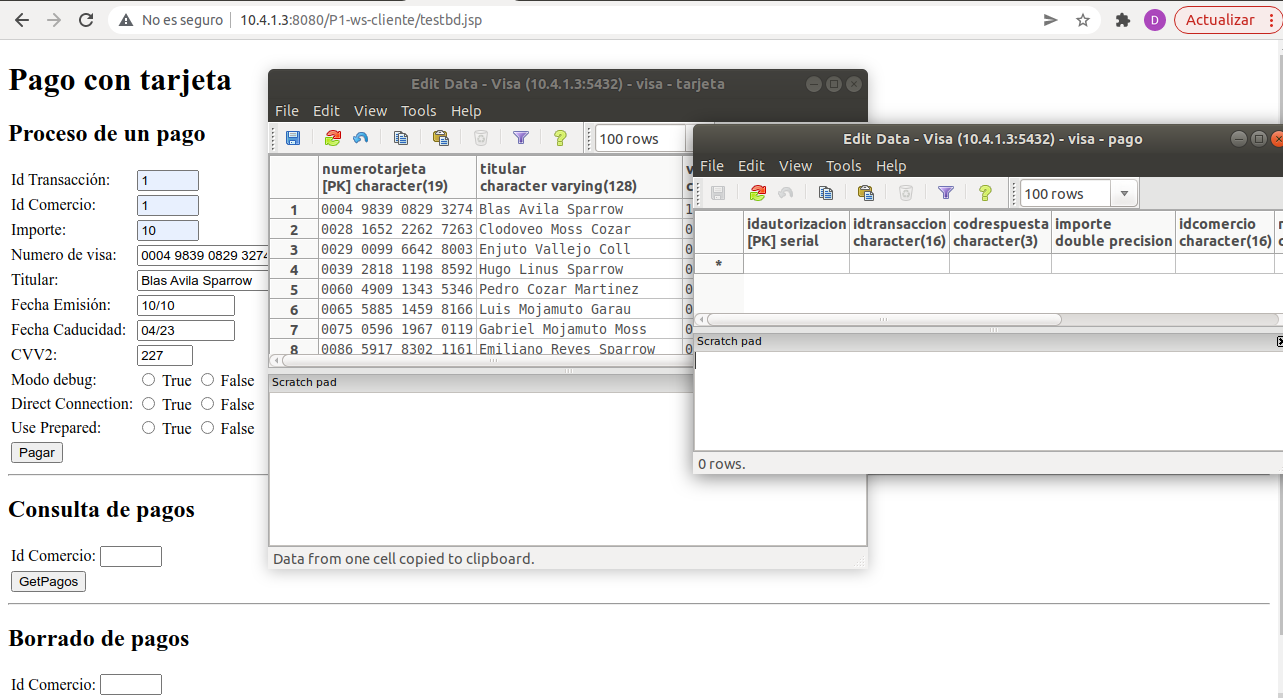
Los ficheros “normales” contienen la información de entrada del método y los ficheros “response” contienen la información de salida.

***Ejercicio número 12:***

***Ejercicio número 13:***

Según se puede apreciar en la url de la imagen, se ha podido acceder a la parte del cliente para poder rellenar el formulario del “testbd.jsp”. Además, se puede observar que se ha accedido desde la ip de la segunda máquina virtual (10.4.1.2). Esto demuestra que la separación del cliente y el servidor se ha hecho correctamente.

Como se puede ver, la base de datos de pagos está vacía y se han introducido correctamente los datos de la tarjeta de Blas Avila Sparrow

******

Como se puede ver en la siguiente imagen, el pago se realiza con éxito actualizándose de esta forma los valores de la base de datos.

## 

## 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 accedería al servlet “Comienza Pago”, donde pasaríamos a “formdatosvisa.jsp” y de ahí al servlet de “Procesa Pago”. Si los datos están bien introducidos, se crea la instancia de “VisaDAO” donde se comprueba la validez de los datos obtenidos.

Es aquí donde se comprobaría que la fecha de caducidad está expirada y se devuelve un error y se redirige a “error/muestraerror.jsp”.

**Cuestión 2. De los diferentes servlets (recuerde que las páginas jsp también se compilan a un servlet) que se usan en la aplicación, ¿podría indicar cuáles son los encargados de obtener la información sobre el pago con tarjeta cuando se usa pago.html para realizar el pago, y cuáles son los encargados de procesarla? ¿Qué información obtiene y procesa cada uno?**

Tanto “Comienza Pago” como “formdatosvisa.jsp” obtienen información sobre el pago, expresamente es “formdatosvisa.jsp” quien recoge la información sobre la tarjeta.

“Procesa Pago”, mediante la instanciación de “VisaDAO”, se encarga de procesar y validar los datos de la tarjeta.

Ambas partes recogen y comprueban, por su parte correspondiente: el número de tarjeta, el titular, la fecha de emisión y de caducidad y el código de verificación (CVV2).

**Cuestión 3. ¿Dónde se crea la instancia de la clase pago cuando se accede por pago.html? ¿Y cuándo se accede por testbd.jsp? Respecto a la información que manejan, ¿cómo la comparte entre los distintos servlets? ¿dónde se almacena? ¿dónde se crea ese almacén?**

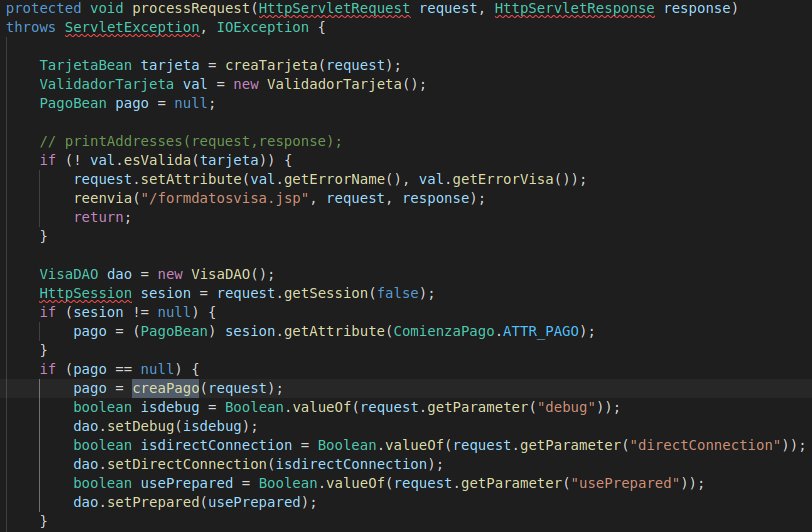
Cuando se accede por “pago.html”, la instancia se crea en el controlador de “ComienzaPago.java”:

 (imagen de pago.html)



Sin embargo, cuando se accede desde “testbd.jsp”, se llama directamente al controlador de “ProcesaPago.java”:

(imagen de testdb.jsp)



Esta última función es más extensa, pero para la justificación nos sirve con lo que se ve. Y es que, se puede observar como se comprueba si existe alguna instancia de “pago”. Esto se hace porque, si la request viene desde “pago.html” hasta “ProcesaPago”, entonces la instancia ya viene creada y se conoce que no se ha usado el “testbd.jsp”. Además, dentro de la sentencia if, se comprueban los parámetros de “debug”, “directConnection” y “usePrepared”, los cuales solo son accesibles usando “testbd.jsp”.

Las informaciones del pago y de la tarjeta se recogen, almacenan y comparten a través de la sesión, usando ATTR\_PAGO.

**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?**

Como mencionaba en la cuestión anterior, el pago funciona correctamente al usar “testbd.jsp” porque se hace una comprobación de la existencia de la instancia de pago al llegar a ese momento. Con esto se comprueba desde donde viene la request: si ha venido de “pago.html”, ha debido pasar por “ComienzaPago”, donde se ha debido de generar la instancia y por tanto se continúa de forma normal; sin embargo, si ha venido de “tesbd.jsp”, entonces es necesario crear una instancia de pago antes de continuar normalmente.

Así que las principales diferencias son:

1. En “pago.html” se recogen solo los datos del pago, después se crea la instancia de pago, se piden los datos de tarjeta y se accede a “ProcesaPago”. En “testbd.jsp”, se recoge todo de una sola vez, saltándose todo ese trayecto y accediendo directamente a “ProcesaPago”.
2. En “testbd.jsp” se pueden incluir opciones adicionales como “debug”, “directConnection” y “usePrepared”.