

SI 2

Práctica 1: 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.

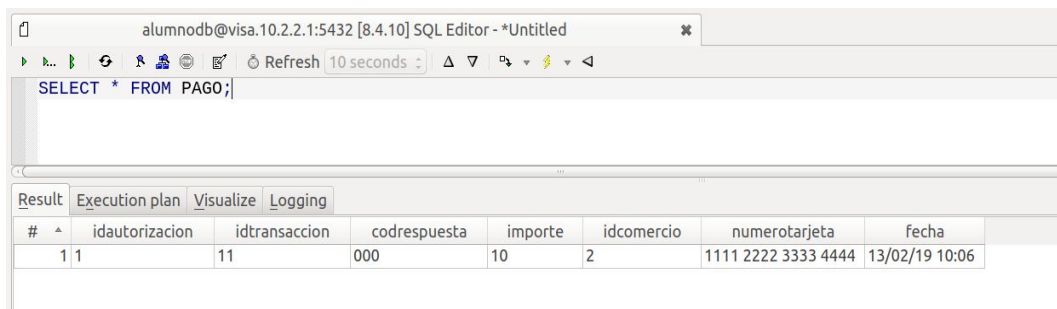
- Acceda a la página de pruebas extendida, <http://10.X.Y.Z:8080/P1/testbd.jsp>. Compruebe que la funcionalidad de listado de y borrado de pagos funciona correctamente. Elimine el pago anterior

En primer lugar lo que hemos hecho ha sido cambiar las direcciones IP necesarias para que la aplicación se pueda desplegar en los servidores correctos, es decir, los nuestros. Hemos modificado:

- Del fichero *build.properties*, la variable *as.host* para especificar la dirección IP del servidor, en este caso, la 10.2.2.1.
- Del fichero *postgresql.properties*, la variable *db.host* para especificar la dirección IP del servidor postgresql que contendrá nuestra base de datos, por lo que la IP es de nuevo 10.2.2.1.
- Por último del fichero *postgresql.properties*, la variable *db.client.host* para especificar el el servidor de aplicaciones dónde se van a crear los recursos (ya sea objeto DataSource o un pool de conexiones), que será la IP del cliente de la base de datos. En nuestro caso volverá a ser la misma, 10.2.2.1.

Una vez cambiados estos parámetros desplegamos nuestra aplicación ejecutando el build.xml y accediendo a la ruta <http://10.2.2.1:8080/P1> para realizar la operación de pago. Aunque previamente hemos podido observar que nuestra aplicación está correctamente desplegada al acceder a <http://10.2.2.1:4848> que es la consola de administración de Glassfish.

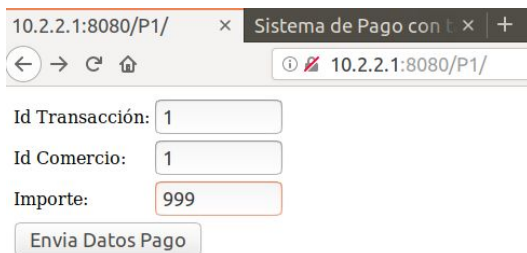
REALIZAR PAGO:



The screenshot shows a SQL Editor window with the query `SELECT * FROM PAGO;` and its result set. The result set contains one row with the following data:

#	idautorizacion	idtransaccion	codrespuesta	importe	idcomercio	numerotarjeta	fecha
1	1	11	000	10	2	1111 2222 3333 4444	13/02/19 10:06

Vemos que en la tabla de pagos de nuestra base de datos, usando la herramienta Tora sólo tenemos un pago (con id = 11). Vamos a efectuar un pago más y ver cómo se modifica la tabla de pagos.



The screenshot shows a web form titled "Sistema de Pago con tarjeta". It has input fields for "Id Transacción:" (value 1), "Id Comercio:" (value 1), and "Importe:" (value 999). There is a button labeled "Envía Datos Pago".

Insertamos el id de transacción, el id de comercio y el importe.



The screenshot shows a web form titled "Pago con tarjeta". It has input fields for "Numero de visa:" (value 1111 2222 3333 4444), "Titular:" (value Jose Garcia), "Fecha Emisión:" (value 11/09), "Fecha Caducidad:" (value 11/20), and "CVV2:" (value 123). There is a button labeled "Pagar". Below the form, there is a summary section showing the transaction details:

Id Transacción:	1
Id Comercio:	1
Importe:	999.0

Prácticas de Sistemas Informáticos II

Introducimos todos los datos de la tarjeta del usuario que desea efectuar el pago.

Sistema de Pago con tarjeta

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

idTransaccion: 1
idComercio: 1
importe: 999.0
codRespuesta: 000
idAutorizacion: 2

[Volver al comercio](#)

Prácticas de Sistemas Informáticos II

Comprobamos que al pulsar “Pagar”, el pago se ha realizado con éxito. Además se muestra un comprobante del mismo. Para asegurarnos de que se ha ejecutado la operación correctamente, comprobaremos que se ha añadido el pago a la tabla *pagos* de nuestra base de datos.

alumnodb@visa.10.2.2.1:5432 [8.4.10] SQL Editor - *Untitled

```
SELECT * FROM PAGOS;
```

#	idautorizacion	idtransaccion	codrespuesta	importe	idcomercio	numerotarjeta	fecha
1	1	11	000	10	2	1111 2222 3333 4444	13/02/19 10:06
2	2	1	000	999	1	1111 2222 3333 4444	13/02/19 10:14

Como podemos observar se ha ejecutado correctamente la operación, ya que se ha insertado una fila a la tabla con los datos de nuestro pago.

Ahora hemos de acceder a la página de pruebas extendida, cuya url en nuestro caso es <http://10.2.2.1:8080/P1/testbd.jsp> y comprobar que las operaciones de consulta y borrado de pago también se ejecutan correctamente.

CONSULTA DE PAGOS:

Vamos a realizar la consulta de un pago para comprobar que nos devuelve los datos del pago que estamos consultando.

Introducimos el identificador de comercio del pago que ya existía (antes de realizar el pago anterior) para comprobar que nos devuelve la información de dicho pago.

Sistema de Pago con tarjeta

Proceso de un pago

Id Transacción:

Id Comercio:

Importe:

Numero de visa:

Titular:

Fecha Emisión:

Fecha Caducidad:

CVV2:

Modo debug: ☒ True ☐ False

Direct Connection: ☒ True ☐ False

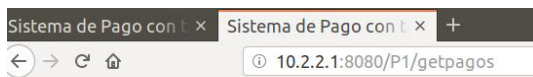
Use Prepared: ☒ True ☐ False

Consulta de pagos

Id Comercio:

Borrado de pagos

Id Comercio:



Pago con tarjeta

Lista de pagos del comercio 2

idTransaccion	Importe	codRespuesta	idAutorizacion
11	10.0	000	1

[Volver al comercio](#)

Prácticas de Sistemas Informáticos II

Como podemos observar nos ha devuelto los mismos datos que los que aparecen en la primera captura adjuntada al caso de realizar pago, por lo que la operación ha sido correcta.

BORRAR PAGO:

Vamos a ver que la acción de borrar un pago es correcta. Para ello vamos a borrar el pago que previamente hemos realizado.

#	idautorizacion	idtransaccion	codrespuesta	importe	idcomercio	numerotarjeta	fecha
1	1	11	000	10	2	1111 2222 3333 4444	13/02/19 10:06
2	2	1	000	999	1	1111 2222 3333 4444	13/02/19 10:14

Vamos a eliminar el pago con id de comercio 1.



Pago con tarjeta

Proceso de un pago

Id Transacción:

Id Comercio:

Importe:

Numero de visa:

Titular:

Fecha Emisión:

Fecha Caducidad:

CVV2:

Modo debug: ☒ True ☐ False

Direct Connection: ☒ True ☐ False

Use Prepared: ☒ True ☐ False

Consulta de pagos

Id Comercio:

Borrado de pagos

Id Comercio:

Prácticas de Sistemas Informáticos II

Introducimos el id de comercio 1, el cual queremos borrar de nuestra base de datos.



Pago con tarjeta

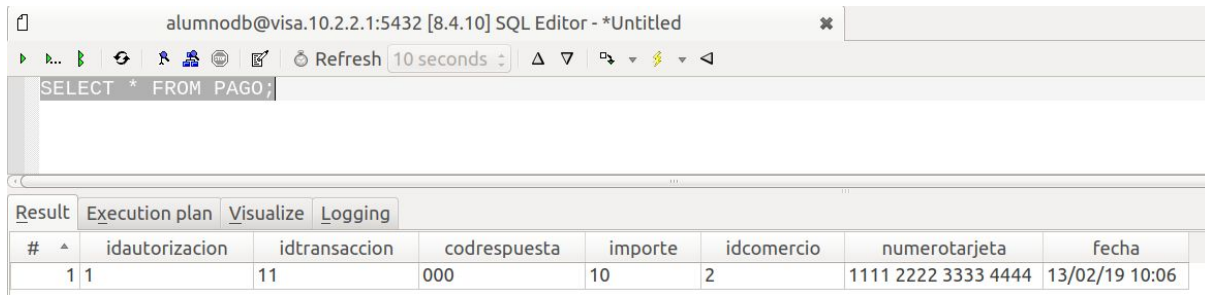
Se han borrado 1 pagos correctamente para el comercio 1

[Volver al comercio](#)

Prácticas de Sistemas Informáticos II

Tenemos confirmación por parte de la aplicación de que nuestro pago ya ha sido borrado.

Por último, comprobamos en la base de datos que el pago que acaba de ser eliminado ya no aparece en la tabla de *pagos*.



The screenshot shows a SQL Editor window titled 'alumnodb@visa.10.2.2.1:5432 [8.4.10] SQL Editor - *Untitled'. The query entered is 'SELECT * FROM PAGO;'. The results are displayed in a table with the following data:

#	idautorizacion	idtransaccion	codrespuesta	importe	idcomercio	numerotarjeta	fecha
1	1	11	000	10	2	1111 2222 3333 4444	13/02/19 10:06

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.

Añadimos las siguientes líneas a DBTester. Lo que estamos haciendo es:

1. Dar nombre al driver JDBC que vamos a usar para realizar la conexión directa.

```
private static final String JDBC_DRIVER = "org.postgresql.Driver";
Class.forName(JDBC_DRIVER).newInstance();
```

2. Usamos el método estático de la clase *DriverManager* para obtener nuevas conexiones, para lo cual modificaremos la cadena de conexión, el usuario y la clave con los valores de nuestra aplicación.

```
private static final String JDBC_CONNECTIONSTRING
="jdbc:postgresql://10.2.2.1:5432/visa";

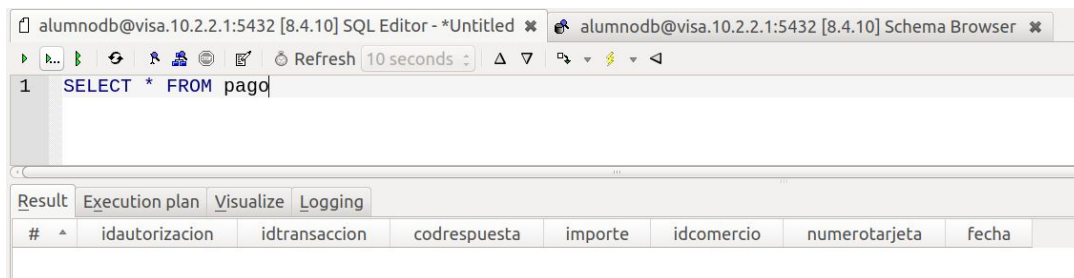
private static final String JDBC_USER = "alumnodb";

private static final String JDBC_PASSWORD = "";

DriverManager.getConnection(JDBC_CONNECTIONSTRING, JDBC_USER, JDBC_PASSWORD);
```

Y hechos estos cambios ahora vamos a comprobar que nuestra aplicación sigue funcionando correctamente pero usando la modalidad de conexión directa.

REALIZAR PAGO:



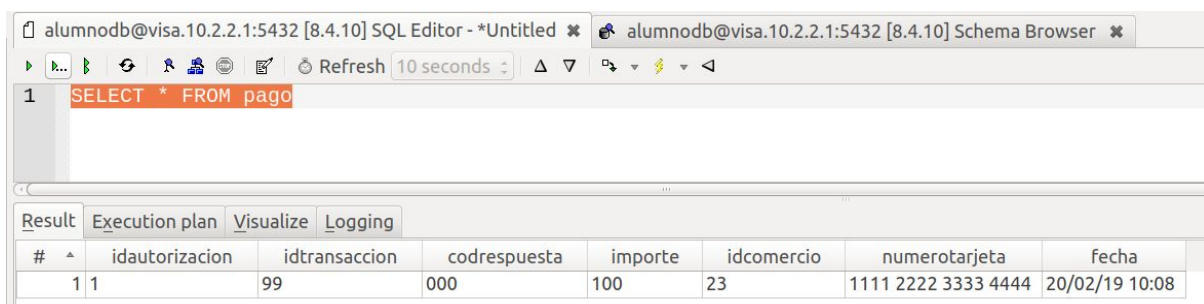
Vemos que la tabla está vacía, vamos a realizar un pago para comprobar que se modifica la tabla de *pagos*.

The screenshot shows the 'Sistema de Pago con tarjeta' web application. It has a 'Proceso de un pago' section with input fields for: Id Transacción (99), Id Comercio (23), Importe (100), Numero de visa (1111 2222 3333 4444), Titular (Jose Garcia), Fecha Emisión (11/09), Fecha Caducidad (11/20), CVV2 (123), and checkboxes for Modo debug, Direct Connection, and Use Prepared. There is a 'Pagar' button. Below this is a 'Consulta de pagos' section with an 'Id Comercio' field and a 'GetPagos' button. At the bottom is a 'Borrado de pagos' section with an 'Id Comercio' field and a 'DelPagos' button. The footer says 'Prácticas de Sistemas Informáticos II'.

Seleccionamos la opción de conexión directa e introducimos todos los datos de la tarjeta y pago. Así obtenemos la confirmación de pago realizado con éxito en la captura de abajo.



Si volvemos a hacer la consulta SQL en la base de datos, vemos que la tabla de *pagos* ya no está vacía si no que tiene el pago que acabamos de realizar usando conexión directa.



CONSULTAR Y BORRAR PAGO:

Si consultamos el pago que acabamos de hacer y luego lo eliminamos comprobaremos que ambas opciones se ejecutan correctamente (recordemos que el id de comercio usado es 23).



10.2.2.1:8080/P1/ x JDBC Resources x Sistema de Pago con tarjeta x +

← → ↻ 10.2.2.1:8080/P1/getpagos

Pago con tarjeta

Lista de pagos del comercio 23

idTransaccion	Importe	codRespuesta	idAutorizacion
99	100.0	000	1

[Volver al comercio](#)

Prácticas de Sistemas Informáticos II



10.2.2.1:8080/P1/ x JDBC Resources x Sistema de Pago con tarjeta x +

← → ↻ 10.2.2.1:8080/P1/delpagos

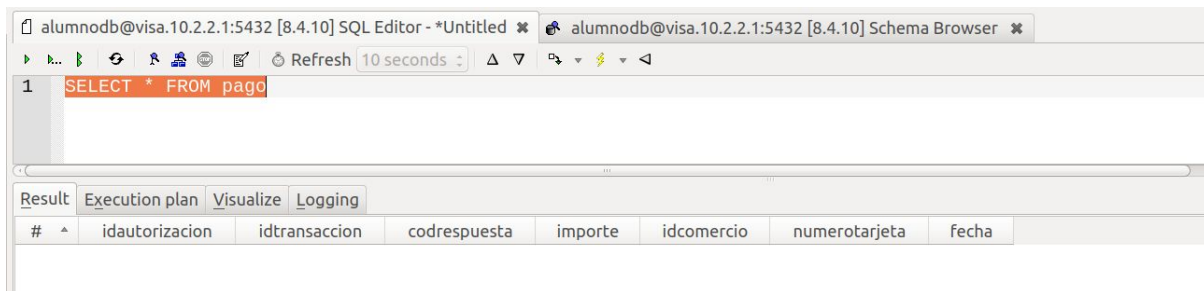
Pago con tarjeta

Se han borrado 1 pagos correctamente para el comercio 23

[Volver al comercio](#)

Prácticas de Sistemas Informáticos II

Si comprobamos la base de datos después de eliminar el pago vemos que la tabla vuelve a estar vacía.



alumnodb@visa.10.2.2.1:5432 [8.4.10] SQL Editor - *Untitled x

alumnodb@visa.10.2.2.1:5432 [8.4.10] Schema Browser x

Refresh 10 seconds

```
1 SELECT * FROM pago
```

Result Execution plan Visualize Logging

#	idautorizacion	idtransaccion	codrespuesta	importe	idcomercio	numerotarjeta	fecha
---	----------------	---------------	--------------	---------	------------	---------------	-------

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 fichero *postgresql.properties* vemos que tenemos una variable llamada *db.pool.name* que contiene el nombre del pool, en nuestro caso es *VisaPool*. Además también tenemos la variable *db.jdbc.resource.name* que contiene el nombre del recurso JDBC correspondiente al DataSource, que en nuestro caso será *jdbc/VisaDB*.

Una vez examinados estos nombres, si accedemos a la consola de Administración (<http://10.2.2.1:4848>) podemos comprobar en las imágenes adjuntas que los recursos JDBC y pool de conexiones están creados con sus respectivos nombres.

Recursos JDBC :

The screenshot shows the GlassFish Server administration console. The browser address bar indicates the URL is `https://10.2.2.1:4848/common/index.jsf`. The page title is "GlassFish™ Server Open Source Edition". The left sidebar contains a tree view of the system configuration, with "JDBC Resources" selected under the "Resources" category. The main content area is titled "JDBC Resources" and includes a description: "JDBC resources provide applications with a means to connect to a database." Below this, there is a table titled "Resources (3)" showing the configuration of three JDBC resources.

Select	JNDI Name	Logical JNDI Name	Enabled	Connection Pool	Description
<input type="checkbox"/>	jdbc/VisaDB		✓	VisaPool	
<input type="checkbox"/>	jdbc/_TimerPool		✓	_TimerPool	
<input type="checkbox"/>	jdbc/_default	java:comp/DefaultDataSource	✓	DerbyPool	

POOL DE CONEXIONES:

10.2.2.1:8080/P1/ x JDBC Connection P x Sistema de Pago con x +

https://10.2.2.1:4848/common/index.jsf

Home About

User: admin Role: domain1 Server: 10.2.2.1

GlassFish™ Server Open Source Edition

Total # of available updates : 1

Common Tasks

- Domain
 - server (Admin Server)
- Clusters
- Standalone Instances
- Nodes
- Applications
- Lifecycle Modules
- Monitoring Data
- Resources
 - Concurrent Resources
 - Connectors
 - JDBC
 - JDBC Resources
 - JDBC Connection Pools**
 - JMS Resources
 - JNDI
 - JavaMail Sessions
 - Resource Adapter Configs
 - Configurations
 - default-config
 - server-config
 - Update Tool

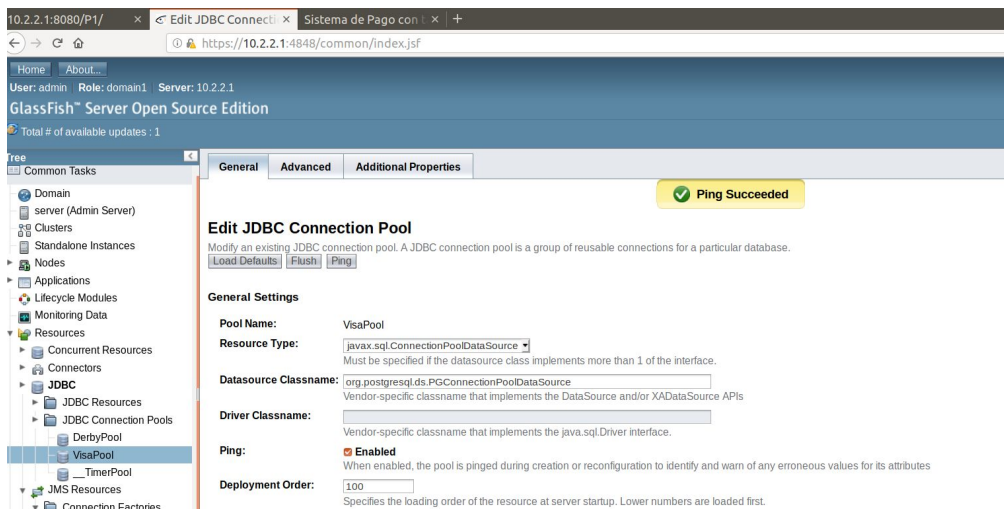
JDBC Connection Pools

To store, organize, and retrieve data, most applications use relational databases. Java EE applications access relational databases through the JDBC API. Before an application can access a database, it must get a connection.

Pools (3)

Select	Pool Name	Resource Type	Classname	Description
<input type="checkbox"/>	DerbyPool	javax.sql.DataSource	org.apache.derby.jdbc.ClientDataSource	
<input type="checkbox"/>	VisaPool	javax.sql.ConnectionPoolDataSource	org.postgresql.ds.PGConnectionPoolDataSource	
<input type="checkbox"/>	TimerPool	javax.sql.XADataSource	org.apache.derby.jdbc.EmbeddedXADataSource	

A continuación vamos a realizar un Ping JDBC a la base de datos para obtener los parámetros que explicaremos a continuación. Primero adjuntamos una imagen de que el ping se ha realizado con éxito:



Una vez realizado el ping, podemos observar una serie de valores en cuanto a las conexiones y al tiempo de ellas que vamos a explicar.

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

En cuanto al rendimiento de nuestra aplicación cabe destacar que como mínimo (Initial and Minimum Pool Size) tenemos 8 conexiones mantenidas en el pool esperando alguna petición de un cliente. Aumentar este número, provocaría un mayor consumo de recursos, no obstante, ahorraría tiempo en caso de esperar muchas conexiones simultaneas.

En cuanto al máximo (Maximum Pool Size) tenemos que son 32 conexiones como máximo las que pueden ser creadas para satisfacer las peticiones de los clientes. Un mayor número conlleva una mayor sobrecarga del servidor, no obstante, permitiría un mayor número de conexiones

El Pool Resize Quantity es dinámico, es decir, si existe una conexión que se encuentra en modo de “espera” y no es usada por ningún cliente antes de que se acabe su tiempo de espera, entonces las conexiones se destruyen de dos en dos. En caso de que llegase una petición de algún cliente antes de que se acabe el tiempo de espera, la conexión es reciclada de tal manera que no es necesario volver a crear una conexión nueva. El hecho de que se destruyan es debido a que si no hay ningún cliente que desee hacer una petición, no tiene sentido mantener conexiones abiertas, ya que estas suponen un elevado coste de memoria.

Con respecto al Idle Timeout, este es el tiempo de espera de conexión inactiva, es decir, cuando pasa ese tiempo el servidor quita del pool de conexiones dicha conexión. Si tenemos aplicaciones con una elevada concurrencia de clientes este tiempo nos es indiferente.

El *Max Wait Time* es el tiempo máximo que puede tardar el servidor en mandar al cliente un *timeout*, es decir, que en un tiempo de 60000 ms no ha sido posible ofrecerle ningún recurso. Estos dos últimos datos lo que hacen es controlar la relación entre la concurrencia de clientes y recursos del servidor, por lo tanto, aumentar su valor conllevaría el riesgo de estar malgastando recursos en conexiones que ya no lo necesitan.

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.

Incluya en la memoria de prácticas dichas consultas

- Consulta de si una tarjeta es válida.

```
/**
 * getQryCompruebaTarjeta
 */
String getQryCompruebaTarjeta(TarjetaBean tarjeta) {
    String qry = "select * from tarjeta "
        + "where numeroTarjeta='" + tarjeta.getNumero()
        + "' and titular='" + tarjeta.getTitular()
        + "' and validaDesde='" + tarjeta.getFechaEmision()
        + "' and validaHasta='" + tarjeta.getFechaCaducidad()
        + "' and codigoVerificacion='" + tarjeta.getCodigoVerificacion() + "'";

    return qry;
}
```

Dicha consulta se ejecuta en las líneas 166-169 del archivo VisaDAO.java

```
164         } else {
165             /*****
166             stmt = con.createStatement();
167             qry = getQryCompruebaTarjeta(tarjeta);
168             errorLog(qry);
169             rs = stmt.executeQuery(qry);
170
171             } *****/
```

- Ejecución del pago.

```
/**
 * getQryInsertPago
 */
String getQryInsertPago(PagoBean pago) {
    String qry = "insert into pago("
        + "idTransaccion,"
        + "importe,idComercio,"
        + "numeroTarjeta)"
        + " values ("
        + "'" + pago.getIdTransaccion() + "',"
        + pago.getImporte() + ","
        + "'" + pago.getIdComercio() + "',"
        + "'" + pago.getTarjeta().getNumero() + "'"
        + ")";

    return qry;
}
```

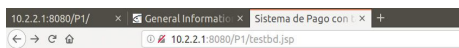
Dicha consulta se ejecuta entre las líneas 250-256 del archivo VisaDAO.java.

```
249      /*****
250      stmt = con.createStatement();
251      String insert = getQryInsertPago(pago);
252      errorLog(insert);
253      ret = false;
254      if (!stmt.execute(insert)
255          && stmt.getUpdateCount() == 1) {
256          ret = true;
```

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 es invocado cada vez que ejecutamos una consulta SQL a la base de datos de visa o cuando se capta una excepción, la sentencia `System.err.println("[directConnection=" + this.isDirectConnection() + "] " + error);` es la encargada de escribir en dicho log.

Ahora vamos a realizar un pago desde la página *testbd.jsp* con la opción de debug activada.



Pago con tarjeta

Proceso de un pago

Id Transacción:

Id Comercio:

Importe:

Numero de visa:

Titular:

Fecha Emisión:

Fecha Caducidad:

CVV2:

Modo debug: ☒ True ☐ False

Direct Connection: ☒ True ☐ False

Use Prepared: ☐ True ☒ False

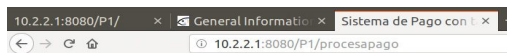
Consulta de pagos

Id Comercio:

Borrado de pagos

Id Comercio:

Realizamos el pago de manera que hayamos seleccionado la opción de debug y vemos que el pago se ha efectuado correctamente.



Pago con tarjeta

Pago realizado con éxito. A continuación se muestra el comprobante de

idTransaccion: 100
idComercio: 23
importe: 101.0
codRespuesta: 000
idAutorizacion: 22

[Volver al comercio](#)

Prácticas de Sistemas Informáticos II

Vamos a comprobar el server log:

Log Viewer - Mozilla Firefox (Private Browsing)

https://10.2.2.1:4848/common/logViewer/logViewer.jsf?instanceName=server&logLevel=INFO&viewResults=true

Log Viewer

View, search, and filter a server log file using basic and advanced options. Refer to the Log Levels page for information about log levels you can filter here.

Advanced Search

Search Criteria

Text search:

Only log entries containing the specified text will be displayed. Search is case sensitive.

Timestamp: ☒ Most Recent ☐ Specific Range:

Log Level: ☐ Do not include more severe messages

Log entries are limited to those stored in the log file. Set appropriate log level in the Log Level page to ensure data is logged.

Modify Search

Instance:

Log File:

Log Viewer Results (40)

Records before 319 | Log File Record Numbers 319 through 358 | Records after 358 |

Record Number	Log Level	Message	Logger	Timestamp	Name-Value Pairs
358	SEVERE	[directConnection=true] select idAutorizacion, codRespuesta from pago where idTransaccion = '100' ... (details)		Feb 20, 2019 10:50:36.192	[levelValue=1000, timeMillis=1550688636192]
357	SEVERE	[directConnection=true] insert into pago(idTransaccion,importe,idComercio,numeroTarjeta) values ('100' ... (details)		Feb 20, 2019 10:50:36.189	[levelValue=1000, timeMillis=1550688636189]
356	SEVERE	[directConnection=true] select * from tarjeta where numeroTarjeta='1111 2222 3333 4444' and titular= ... (details)		Feb 20, 2019 10:50:36.181	[levelValue=1000, timeMillis=1550688636181]
355	INFO	WebModule[nul] ServletContext.logg [INFO] Acceso correcto /procesapago/details	javax.enterprise.web	Feb 20, 2019 10:50:36.156	[levelValue=800, timeMillis=1550688636156]
354	INFO	WebModule[nul] ServletContext.logg [INFO] Acceso correcto /delpagos/details	javax.enterprise.web	Feb 20, 2019 10:09:26.660	[levelValue=800, timeMillis=1550688166660]
353	INFO	WebModule[nul] ServletContext.logg [INFO] Acceso correcto /getpagos/details	javax.enterprise.web	Feb 20, 2019 10:09:14.620	[levelValue=800, timeMillis=1550688154620]
352	INFO	visiting unvisited references(details)	javax.enterprise.system.tools.deployment.dsl	Feb 20, 2019 10:08:54.841	[levelValue=800, timeMillis=1550688134841]
351	INFO	WebModule[nul] ServletContext.logg [INFO] Acceso correcto /procesapago/details	javax.enterprise.web	Feb 20, 2019 10:08:54.663	[levelValue=800, timeMillis=1550688134663]
350	INFO	WebModule[nul] ServletContext.logg [INFO] Acceso correcto /testdb.jsp(details)	javax.enterprise.web	Feb 20, 2019 10:08:13.577	[levelValue=800, timeMillis=15506880973577]
349	INFO	WebModule[nul] ServletContext.logg [INFO] Acceso correcto /pago.html(details)	javax.enterprise.web	Feb 20, 2019 10:04:42.220	[levelValue=800, timeMillis=1550688042220]
348	INFO	WebModule[nul] ServletContext.logg [WARNING] Acceso No autorizado /testdb.jsp(details)	javax.enterprise.web	Feb 20, 2019 10:03:58.219	[levelValue=800, timeMillis=1550688020219]
347	INFO	P1 was successfully deployed in 124 milliseconds (details)	javax.enterprise.system.core	Feb 20, 2019 10:03:49.513	[levelValue=800, timeMillis=1550688020513]
346	INFO	Loading application [P1] at [P1](details)	javax.enterprise.web	Feb 20, 2019 10:03:49.498	[levelValue=800, timeMillis=1550688020498]
345	INFO	visiting unvisited references(details)	javax.enterprise.system.tools.deployment.common	Feb 20, 2019 10:03:49.439	[levelValue=800, timeMillis=1550688020439]
344	INFO	visiting unvisited references(details)	javax.enterprise.system.tools.deployment.common	Feb 20, 2019 10:03:49.437	[levelValue=800, timeMillis=1550688020437]
343	INFO	visiting unvisited references(details)	javax.enterprise.system.tools.deployment.common	Feb 20, 2019 10:03:49.403	[levelValue=800, timeMillis=1550688020403]
342	INFO	WebModule[nul] ServletContext.logg [WARNING] Acceso No autorizado /testdb.jsp(details)	javax.enterprise.web	Feb 20, 2019 10:02:22.886	[levelValue=800, timeMillis=1550687742886]
341	INFO	WebModule[nul] ServletContext.logg [WARNING] Acceso No autorizado /testdb.jsp(details)	javax.enterprise.web	Feb 20, 2019 10:02:16.651	[levelValue=800, timeMillis=1550687736651]

https://10.2.2.1:4848/common/logViewer/logEntryDetail.jsf?instanceName=server&logLevel=SEVERE&logFile=:

Log Entry Detail

Timestamp Feb 20, 2019 10:50:36.192
Log Level SEVERE
Logger
Name-Value Pairs {levelValue=1000, timeMillis=1550688636192}
Record Number 358
Message ID
Complete Message [directConnection=true] select idAutorizacion, codRespuesta from pago where idTransaccion = '100' and idComercio = '23'

Ejercicio 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).
- isPrepared() / setPrepared()

Para que estos métodos sean publicados como métodos del servicio hemos tenido que insertar los siguientes import y el resto de líneas de código correspondiente a cada método:

```
import javax.jws.WebMethod;  
import javax.jws.WebParam;  
import javax.jws.WebService;
```

Después, previo a la declaración de la clase, añadimos @WebService()

```
@WebService()  
public class VisaDAOWS extends DBTester {
```

A continuación, mostramos las correspondientes modificaciones en las declaraciones de los métodos en función de si tenían o no parámetros de entrada:

```
@WebMethod(operationName = "compruebaTarjeta")  
public boolean compruebaTarjeta(@WebParam(name = "tarjeta") TarjetaBean  
tarjeta) {  
  
@WebMethod(operationName = "realizaPago")  
public synchronized PagoBean realizaPago( @WebParam (name= "pago")  
PagoBean pago) {  
  
@WebMethod(operationName = "isDebug")  
public boolean isDebug() {  
  
@WebMethod(operationName = "setDebug")  
public void setDebug( @WebParam (name= "debug") boolean debug) {
```

```

@WebMethod(operationName = "isPrepared")
public boolean isPrepared() {

@WebMethod(operationName = "setPrepared")
public void setPrepared(@WebParam (name= "prepared") boolean prepared)

```

(Comentamos el método setDebug que no se usa para que no haya conflictos)

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.

```

@Override
@WebMethod(operationName = "isDirectConnection")
public boolean isDirectConnection() {

@Override
@WebMethod(operationName = "setDirectConnection")
public boolean isDirectConnection() {

```

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.

```

@WebMethod(operationName = "realizaPago")
public synchronized PagoBean realizaPago( @WebParam (name= "pago") PagoBean pago) {
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    .
    .
    .

```

Luego en vez de devolver false, devolvemos null en caso de error.

```

// Calcular pago.
// Comprobar id.transaccion - si no existe,
// es que la tarjeta no fue comprobada
if (pago.getIdTransaccion() == null) {
    return null;
}

```


. . .

Al final lo que hacemos es devolver el pago en sí (con los campos identificador de autorización y código de respuesta a null si hay algún error):

```
    return pago;
}
```

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?

Cuando realizamos una llamada a un web method, los parámetros de entrada se pasan por valor, no por referencia, por tanto, es necesario devolver el pago, o de lo contrario no seremos capaces de recoger el pago modificado.

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.

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<!--
  Published by JAX-WS RI (http://jax-ws.java.net). RI's version is Metro/2.3.2-b608 (trunk-7979; 2015-01-21T12:50:19+0000) JAXWS-RI/2.2.11-b150120.1832 JAXWS-APT/2.2.12 JAXB-RI/2.2.12-b141219.1637 JAXB-APT/2.2.13-b141020.1521 svn-revision#unknown.
-->
<!--
  Generated by JAX-WS RI (http://jax-ws.java.net). RI's version is Metro/2.3.2-b608 (trunk-7979; 2015-01-21T12:50:19+0000) JAXWS-RI/2.2.11-b150120.1832 JAXWS-APT/2.2.12 JAXB-RI/2.2.12-b141219.1637 JAXB-APT/2.2.13-b141020.1521 svn-revision#unknown.
-->
<definitions targetNamespace="http://dao.visa.ssi2/" name="VisaDAOSService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://dao.visa.ssi2/" schemaLocation="http://10.2.2.1:8080/P1-ws-ws/VisaDAOSService?xsd=1"/>
    </xsd:schema>
  </types>
  <message name="compruebaTarjeta">
    <part name="parameters" element="tns:compruebaTarjeta"/>
  </message>
  <message name="compruebaTarjetaResponse">
    <part name="parameters" element="tns:compruebaTarjetaResponse"/>
  </message>
  <message name="realizaPago">
    <part name="parameters" element="tns:realizaPago"/>
  </message>
  <message name="realizaPagoResponse">
    <part name="parameters" element="tns:realizaPagoResponse"/>
  </message>
  <message name="getPagos">
    <part name="parameters" element="tns:getPagos"/>
  </message>
  <message name="getPagosResponse">
    <part name="parameters" element="tns:getPagosResponse"/>
  </message>
  <message name="delPagos">
    <part name="parameters" element="tns:delPagos"/>
  </message>
  <message name="delPagosResponse">
    <part name="parameters" element="tns:delPagosResponse"/>
  </message>
  <message name="isPrepared">
    <part name="parameters" element="tns:isPrepared"/>
  </message>
  <message name="isPreparedResponse">
    <part name="parameters" element="tns:isPreparedResponse"/>
  </message>
  <message name="setPrepared">
    <part name="parameters" element="tns:setPrepared"/>
  </message>
```

Conteste a las siguientes preguntas:

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

En el WSDL, que es el fichero encargado de definir la interfaz pública a los servicios Web aparece un link donde están definidos los tipos básicos para la comunicación
`http://10.2.2.1:8080/P1-ws-ws/VisaDAOWSService?xsd=1`.

• ¿Qué tipos de datos predefinidos se usan?

Boolean, string y double.

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

Aquellos que comienzan con el prefijo *xs* que, al comienzo del fichero, define la ruta del documento donde se encuentran definidos los tipos.

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

La etiqueta usada es *portType*.

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

La etiqueta es *message*.

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

En la etiqueta *soap:binding* en el argumento *transport*.

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

En la etiqueta *soap:address* en el argumento *location*.

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.

Hacemos los *imports* correspondientes para que el servicio cliente pueda invocar servicio remoto mediante *stubs estáticos*:

```
//import ssii2.visa.dao.VisaDAO;
import ssii2.visa.VisaDAOWSService; // Stub generado automáticamente
import ssii2.visa.VisaDAOWS; // Stub generado automáticamente
import javax.xml.ws.WebServiceRef;
```

Posteriormente, hacemos la instanciación de la clase remota, mediante el objeto *VisaDAOWSService* que es una referencia al servicio remoto, para obtener así el *stub local*. Así podemos obtener el puerto del servicio remoto mediante la función *service.getVisaDAOWSPort()* guardando en el objeto *VisaDAOWS*.

```
try{
    VisaDAOWSService service = new VisaDAOWSService();
    dao = service.getVisaDAOWSPort ();
}
```

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.

En el fichero *web.xml* es necesario añadir un parámetro que contenga el valor de la ruta al servicio remoto, para que luego pueda ser extraída de este fichero usando el nombre de este parámetro.

```
<!--
<context-param>
    <param-name>webmaster</param-name>
    <param-value>myaddress@mycompany.com</param-value>
</context-param>
-->

<context-param>
    <param-name>urlCompleto</param-name>
    <param-value>http://10.2.2.1:8080/P1-ws-ws/VisaDAOWSService</param-value>
</context-param>
```

En el fichero *ProcesaPago.java* tenemos que modificar la ruta del servidor sin modificar la definición del servicio hecha previamente en el ejercicio 8 y que también se puede apreciar en la captura de abajo. Para ello, migramos dicho servicio a una ubicación física distinta usando la función *getServletContext().getInitParameter("urlCompleto")* donde hacemos referencia al parámetro de nuestro *web.xml* donde tenemos declarada la ruta, seguido de la función

```
bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, urlServRemoto).
```

```

VisaDAOWS dao = null; //Lo declaramos para que al compilar dao este visible para el resto del codigo

try{
    VisaDAOWSService service = new VisaDAOWSService();
    dao = service.getVisaDAOWSPort ();

    BindingProvider bp = (BindingProvider) dao;
    String urlServRemoto = getServletContext().getInitParameter("urlCompleto");
    bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,urlServRemoto);

}catch (Exception ee){
    errorLog(ee.toString());
}

```

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.

```

@WebMethod(operationName = "delPagos")
public int delPagos(@WebParam(name = "idComercio" )String idComercio) {

```

- Servlet GetPagos.java: la operación dao.getPagos() debe implementarse en el servicio web.

```

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

```

Tenga en cuenta que no todos los tipos de datos son compatibles con JAXB(especifica cómo 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<PagoBean> 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.

```

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

    PreparedStatement pstmt = null;
    Connection pcon = null;
    ResultSet rs = null;
    PagoBean[] ret = null;
    ArrayList<PagoBean> pagos = null;
    String qry = null;

    try {

```

. . .
(dentro del try)

```

    pagos = new ArrayList<PagoBean>();

    while (rs.next()) {
        TarjetaBean t = new TarjetaBean();
        PagoBean p = new PagoBean();
        p.setIdTransaccion(rs.getString("idTransaccion"));
        p.setIdComercio(rs.getString("idComercio"));
        p.setImporte(rs.getFloat("importe"));
        t.setNumero(rs.getString("numeroTarjeta"));
        p.setTarjeta(t);
        p.setCodRespuesta(rs.getString("codRespuesta"));
        p.setIdAutorizacion(String.valueOf(rs.getInt("idAutorizacion")));

        pagos.add(p);
    }

    ret = new PagoBean[pagos.size()];
    ret = pagos.toArray(ret);

```

En la imagen superior en la última línea vemos como usamos el método *toArray()* de la lista de *PagoBean* con el tamaño de la lista de pagos.

```

. . .
    return pagos;
}

```

En el cliente, debemos convertir el `ArrayList<PagoBean>` mediante las siguientes instrucciones:

```

List<PagoBean> arrayPagos = dao.getPagos(idComercio);
PagoBean[] pagos = arrayPagos.toArray(new PagoBean[arrayPagos.size()]);

```

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.

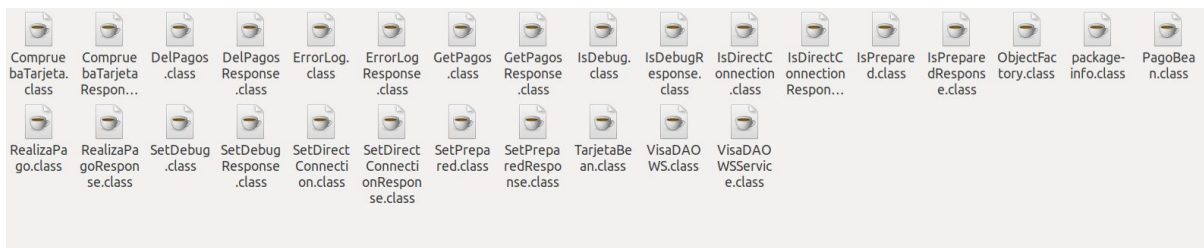
El comando ejecutado es el siguiente:

```
luis@luis-HP-Pavilion-x360:~/Desktop/UAM/SI2/SI2/P1/P1-ws$ wsimport -d
build/client/WEB-INF/classes -p ssii2.visa
http://10.2.2.1:8080/P1-ws-ws/VisaDAOWSService?wsdl
parsing WSDL...
```

Generating code...

Compiling code...

Las clases que se han generado son las siguientes:



Wsimport realiza un parseo al WSDL, generando los archivos necesarios para acceder los servicios publicados, con lo cual, las clases generadas son las que necesita el cliente para tener conocimiento de los web-methods, por tanto, necesita una clase para cada método, así como una clase para cada clase principal (como VisaDAOWS o TarjetaBean). Cabe destacar dos clases que se generan junto con los métodos, estas son ObjectFactory.class y package-info.class, la primera contiene los métodos de fábrica para cada interfaz de contenido Java y para el contenido Java generado en el paquete ssii2.visa. La segunda crea la estructura del directorio.

Junto con las clases de los métodos, se generan unas análogas terminadas en Response.class, estas reflejan las respuestas de los métodos.

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). Téngase en cuenta que:

- El raíz del directorio de salida del compilador para la parte cliente ya está definido en build.properties como \${build.client}/WEB-INF/classes
- El paquete Javaraíz(ssii2) ya está definido como \${paquete}
- La URL ya está definida como \${wsdl.url}


```
<target name="generar-stubs" depends="montar-jerarquia" description="Genera los stubs del cliente a partir del archivo WSDL">
  <exec executable="wsimport">
    <arg line=" -d ${build.client}/WEB-INF/classes" />
    <arg line=" -p ${paquete}.visa" />
    <arg line=" ${wsdl.url}" />
  </exec>
</target>
```

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.

```
as.user=admin
as.host.client=10.2.2.2
as.host.server=10.2.2.1
as.port=4848
as.passwordfile=${basedir}/p
```

Aquí observamos cómo las dos IPs tanto la de cliente como la del servidor son distintas.

• 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.

The screenshot displays a dual-pane environment. The left pane shows a terminal window with the command prompt 'alumnodb@visa.10.2.2.1:5432 [8.4.]' and a SQL query 'select * from pago;' being executed. Below the terminal, a table of results is visible with columns: #, idautorizacion, idtransaccion, codrespuesta, importe, idcomercio, numerotarjeta, and fecha. The right pane shows a web browser window titled 'Sistema de Pago con tarjeta - Mozilla Firefox (Private Browser)'. The browser displays the 'testbd.jsp' page, which contains a form for 'Pago con tarjeta' and sections for 'Consulta de pagos' and 'Borrado de pagos'. The 'Pago con tarjeta' section includes fields for transaction ID, merchant ID, amount, card number, cardholder name, and expiration date, along with radio buttons for debug mode and prepared statements. The 'Consulta de pagos' section has a merchant ID field and a 'GetPagos' button. The 'Borrado de pagos' section has a merchant ID field and a 'DelPagos' button. The browser's address bar shows the URL '10.2.2.2:8080/P1-ws-cliente/testbd.jsp?sessionId=a979f471340d0c21c13e1771b5ba'.

En esta imagen podemos ver cómo en el host servidor (IP: 10.2.2.1) tenemos abierto Tora con la base de datos y realizada una consulta viendo los pagos que ha efectuados hasta

el momento, es decir, ninguno (parte izquierda de la imagen). En el lado derecho tenemos el host cliente (IP: 10.2.2.1) con los campos necesarios para realizar un pago rellenos. Vamos a ver qué ocurre si el cliente efectúa el pago.

The screenshot shows a web browser window on the right and a SQL Editor window on the left. The browser window displays a confirmation message for a payment made with a card. The SQL Editor window shows a query result table with one row of data.

Pago con tarjeta

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

idTransaccion: 1
idComercio: 11
importe: 111.0
codRespuesta: 000
idAutorizacion: 6

[Volver al comercio](#)

Prácticas de Sistemas Informáticos II

SQL Editor: alumnodb@visa.10.2.2.1:5432 [8.4.10] SQL Editor - *Untitled

select * from pago

#	idautorizacion	idtransaccion	codrespuesta	importe	idcomercio	numerotarjeta	fecha
1	6	1	000	111	11	1111 2222 3333 4444	23/02/19 05:42

Row: 1 Col: 19

En la parte derecha de la imagen vemos que al cliente le ha llegado la información de que su pago ha sido realizado correctamente. En cuanto al servidor tenemos que al volver a realizar la misma consulta que antes (donde el resultado era una tabla vacía), ahora le aparece el pago con los datos que el cliente había introducido, por lo que el pago ha sido correctamente efectuado. Ahora vamos a comprobar el resto de operaciones.

The screenshot shows a web browser window on the right and a SQL Editor window on the left. The browser window displays a list of payments from a specific commerce. The SQL Editor window shows a query result table with one row of data.

Pago con tarjeta

Lista de pagos del comercio 11

idTransaccion	importe	codRespuesta	idAutorizacion
1	111.0	000	6

[Volver al comercio](#)

Prácticas de Sistemas Informáticos II

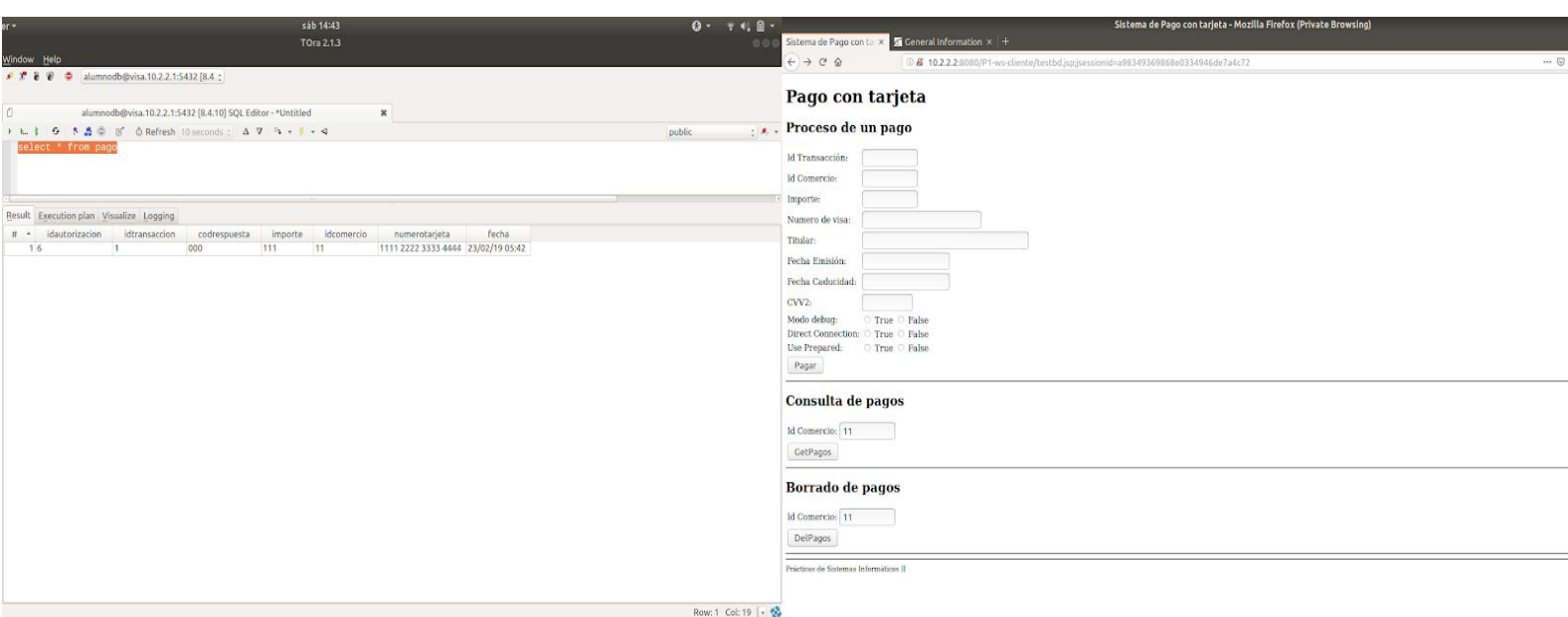
SQL Editor: alumnodb@visa.10.2.2.1:5432 [8.4.10] SQL Editor - *Untitled

select * from pago

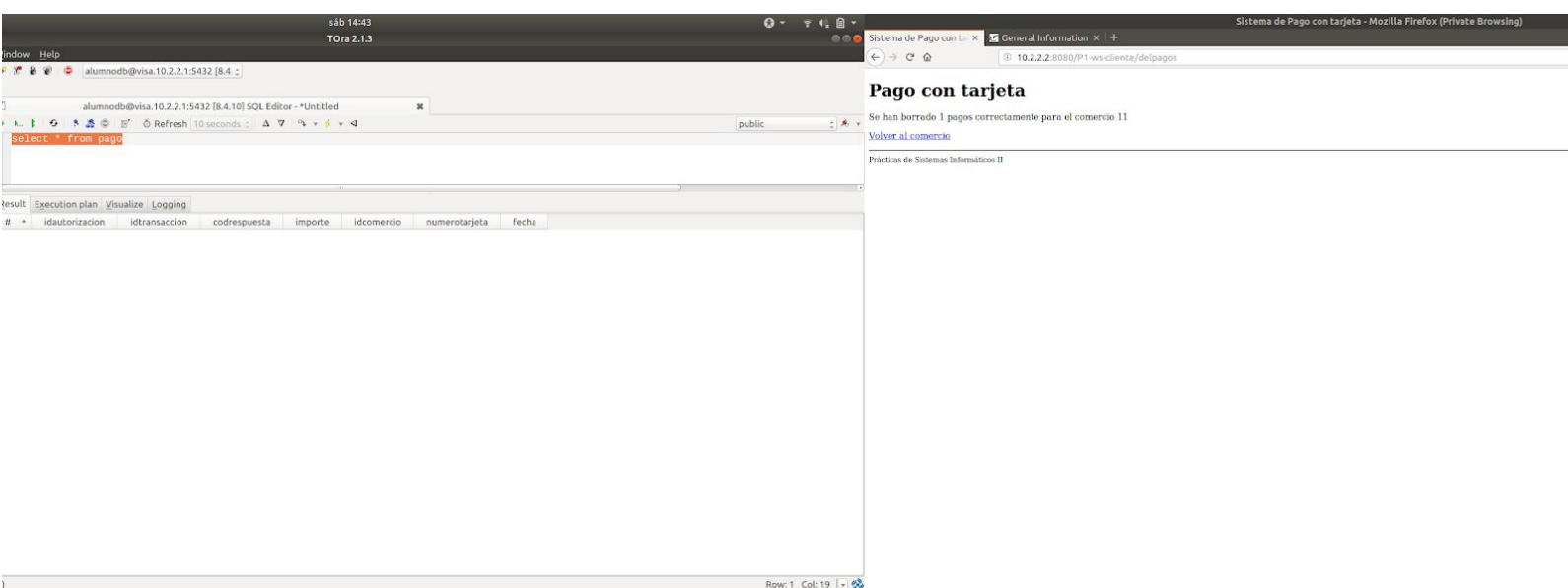
#	idautorizacion	idtransaccion	codrespuesta	importe	idcomercio	numerotarjeta	fecha
1	6	1	000	111	11	1111 2222 3333 4444	23/02/19 05:42

Row: 1 Col: 19

En cuanto a la operación de listar pagos el cliente obtiene correctamente los datos de la base de datos que reside en el host servidor. De igual manera, podemos comprobar que la consulta en la base de datos en el servidor sigue devolviendo el pago que se había ejecutado con anterioridad. Por último, comprobaremos la operación de borrado de pago.



En esta imagen podemos ver el estado inicial de la tabla de pagos en el servidor (parte izquierda con IP 10.2.2.1) donde la tabla de pagos continúa con el pago realizado antes. En cuanto a la pantalla del cliente podemos ver cómo ha introducido en el campo de id comercio el id del único pago que se encuentra en la tabla de *pagos* de la base de datos. Ahora veremos qué es lo que sucede cuando el cliente selecciona la opción de borrar.



Como observamos, al cliente le llega el mensaje de confirmación de pago borrado. En cuanto al servidor si volvemos a realizar la consulta para ver las filas que tiene la tabla *pagos* vemos que ya no hay ningún pago porque se ha borrado el pago con el id de comercio del único pago que había en la tabla. Concluimos así que se realizan las tres operaciones con éxito.

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.

Empezamos en la página *pago.html*, a continuación el servlet *comienzaPago*. Nos aparece un formulario, por lo que pasamos por *formdatosvisa.jsp*, pero como la fecha de caducidad de la tarjeta ha expirado, vamos a *error/muestraerror.jsp*

Cuestión 2. De los diferentes servlets que se usan en la aplicación, ¿podría indicar cuáles son los encargados de solicitar la información sobre el pago con tarjeta cuando se usa *pago.html* para realizar el pago?

El servlet *ComienzaPago* que valida los parámetros del nuevo pago (id de transacción y de comercio e importe) , crea una nueva sesión, y redirige la petición al formulario *formdatosvisa.jsp*.

Tras esto, se usa el servlet *ProcesaPago*, que se encarga de solicitar la información de la tarjeta.

Cuestión 3. Cuando se accede a *pago.html* para hacer el pago, ¿qué información solicita cada servlet? Respecto a la información que manejan, ¿cómo la comparten? ¿dónde se almacena?

ComienzaPago solicita el id de transacción, de comercio y de importe, tras esto, guarda la información del pago en la sesión (mediante el atributo *ATTR_PAGO*).

ProcesaPago recoge la información de la sesión y la complementa con toda la información que solicita de la tarjeta (número, titular, fecha de caducidad y emisión y cvv), que a su vez vuelve a guardar en la variable de sesión *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?

Cuando accedemos al pago desde `pago.html`, vemos como el formulario se envía al servlet *ComienzaPago* y este ya se encargará de continuar con *ProcesaPago*. No obstante, cuando utilizamos la página de prueba `testbd.jsp` su formulario se envía directamente a *ProcesaPago*.

Estas diferencias no provocan ningún mal funcionamiento del programa, ya que *ProcesaPago* lo primero que comprueba es si existe algún pago en la sesión *ATTR_PAGO*, si no existe (estamos en el caso de `testbd.jsp`), crea uno nuevo usando la petición, que contiene todos los atributos necesarios, tales como los identificadores de transacción y de comercio y el importe (*creaPago(request);*).