		Escuela Politécnica Superior Ingeniería Informática Prácticas de Sistemas Informáticos 2			
Grupo	2401	Práctica	1	Fecha	05/02/2019
Alumno/a	Cea Tassel, Claudia				
Alumno/a	Riera Gómez, Juan				

Práctica 1: Arquitectura de JAVA EE (Primera parte)

Ejercicio número 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.

Con el fin de desplegar tanto la aplicación web como la base de datos, hemos realizado alguna modificación en los archivos build.properties y postgresql.properties. En el primero hemos sustituido el valor de as.host por 10.1.3.1, que es la dirección asignada según nuestro grupo y número de pareja. En postgresql.properties hemos modificado los valores de db.host y db.client.host por la dirección mencionada anteriormente.

Una vez modificados los datos, entramos en la aplicación web cuya ruta es <http://10.1.3.1:8080/P1> y realizamos un pago. Para obtener los datos que vamos a utilizar al pagar, accedemos con psql a la base de datos visa, donde están almacenados los usuarios y los pagos. Utilizamos el comando: psql -U alumnodb visa y con una consulta obtenemos los datos de un usuario. Elegimos:

7557 9649 3955 5976 | Irene Avila Morales | 01/10 | 03/20 | 185

Pago con tarjeta

Numero de visa:	<input type="text" value="7557 9649 3955 5976"/>
Titular:	<input type="text" value="Irene Avila Morales"/>
Fecha Emisión:	<input type="text" value="01/10"/>
Fecha Caducidad:	<input type="text" value="03/20"/>
CVV2:	<input type="text" value="185"/>
<input type="button" value="Pagar"/>	

Id Transacción: 123
Id Comercion: 456
Importe: 666.0

Prácticas de Sistemas Informáticos II

Vemos que el pago se ha realizado correctamente:

Pago con tarjeta

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

idTransaccion: 123
idComercio: 456
importe: 666.0
codRespuesta: 000
idAutorizacion: 1

[Volver al comercio](#)

Prácticas de Sistemas Informáticos II

Comprobamos en la base de datos que el pago se ha registrado correctamente:

Archivo Editar Ver Buscar Terminal Ayuda						
idautorizacion	idtransaccion	codrespuesta	importe	idcomercio	numerotarjeta	fecha
1	123	000	666	456	7557 9649 3955 5976	2019-02-13 06:01:03.727667
(1 row)						

- 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. Incluya en la memoria de prácticas todos los pasos necesarios para resolver este ejercicio así como las evidencias obtenidas. Se pueden incluir por ejemplo capturas de pantalla.

Accedemos al pago con ID de transacción 123:

Pago con tarjeta

Lista de pagos del comercio 456

idTransaccion	Importe	codRespuesta	idAutorizacion
123	666.0	000	1

[Volver al comercio](#)

Prácticas de Sistemas Informáticos II

Borramos el pago:

Pago con tarjeta

Se han borrado 1 pagos correctamente para el comercio 456

[Volver al comercio](#)

Prácticas de Sistemas Informáticos II

Comprobamos que el pago se ha borrado correctamente:

Pago con tarjeta

Lista de pagos del comercio 456

idTransaccion	Importe	codRespuesta	idAutorizacion
---------------	---------	--------------	----------------

[Volver al comercio](#)

Prácticas de Sistemas Informáticos II

Ejercicio número 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

Hemos reemplazado la ruta de acceso a la base de datos, ya que antes intentaba conectarse a una de tipo derby, en una ip distinta a la nuestra, un puerto distinto y un usuario distinto.

Para ello hemos corregido algunas líneas del fichero DBTester.java, añadiendo como ruta de conexión a la base de datos: "jdbc:postgresql://10.1.3.1:5432/visa"

Como usuario: "alumnodb"

Y como contraseña, que no importa porque el usuario no tiene contraseña: "*****"

Hemos compilado después el servidor y lo hemos reiniciado. Seguidamente hemos realizado la consulta rellenando el formulario con los siguientes campos:

Pago con tarjeta

Proceso de un pago

Id Transacción:	<input type="text" value="2"/>
Id Comercio:	<input type="text" value="2"/>
Importe:	<input type="text" value="2"/>
Numero de visa:	<input type="text" value="6513 0633 4651 1154"/>
Titular:	<input type="text" value="Gabriel Locke Martinez"/>
Fecha Emisión:	<input type="text" value="04/08"/>
Fecha Caducidad:	<input type="text" value="02/20"/>
CVV2:	<input type="text" value="681"/>
Modo debug:	<input checked="" type="radio"/> True <input type="radio"/> False
Direct Connection:	<input checked="" type="radio"/> True <input type="radio"/> False
Use Prepared:	<input checked="" type="radio"/> True <input type="radio"/> False
<input type="button" value="Pagar"/>	

La respuesta del servidor ha sido la siguiente:

Pago con tarjeta

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

idTransaccion: 2
idComercio: 2
importe: 2.0
codRespuesta: 000
idAutorizacion: 1

[Volver al comercio](#)

Prácticas de Sistemas Informáticos II

Después hemos listado las transacciones asociadas al comercio:

Pago con tarjeta

Lista de pagos del comercio 2

idTransaccion	Importe	codRespuesta	idAutorizacion
2	2.0	000	1

[Volver al comercio](#)

Prácticas de Sistemas Informáticos II

Hemos borrado los pagos asociados con el comercio 2:

Pago con tarjeta

Se han borrado 1 pagos correctamente para el comercio 2

[Volver al comercio](#)

Prácticas de Sistemas Informáticos II

Y hemos accedido al listado otra vez, comprobando que efectivamente se ha borrado:

Pago con tarjeta

Lista de pagos del comercio 2

idTransaccion	Importe	codRespuesta	idAutorizacion

[Volver al comercio](#)

Prácticas de Sistemas Informáticos II

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.

Investigando el fichero de configuración, hemos averiguado que el nombre del recurso JDBC es "jdbc/VisaDB" y que el nombre del pool es "VisaPool".

Hemos comprobado después que tanto la base de datos como el Pool funcionaban correctamente entrando en la url de administración de glassfish "<https://10.1.3.1:4848>" y navegando hasta llegar a las siguientes páginas:

- Página asociada al servicio JDBC, donde se ve que aparece como "enabled":

JDBC Resources

JDBC resources provide applications with a means to connect to a database.

Resources (3)						
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="button" value="New..."/>	<input type="button" value="Delete"/>	<input type="button" value="Enable"/>	<input type="button" value="Disable"/>	
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		

- Página asociada al pool de conexiones, en la que hemos hecho un ping JDBC al pool, obteniendo el mensaje "ping succeeded" como se aprecia en la siguiente captura:



Edit JDBC Connection Pool

Modify an existing JDBC connection pool. A JDBC connection pool is a group of reusable connections for a particular

General Settings

Pool Name:

VisaPool

Resource Type:

javax.sql.ConnectionPoolDataSource

▼

- Desde la misma página, haciendo *scroll* hacia abajo, llegamos a ver los datos pedidos en el enunciado, que se observan en la siguiente imagen:

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 a cómo afectan estos parámetros:

- El Initial and Minimum Pool Size establece el número mínimo de procesos conectados concurrentemente con la base de datos, muy pocos procesos puede llevar a una saturación de cada uno de los mismos, pero demasiados procesos puede saturar a la base de datos.
- El pool máximo se encarga de mitigar la posible crecida de procesos en el pool, si empezamos en un punto medio y glassfish detecta que necesita más, puede abrir más conexiones y llegar a saturar la base de datos. Este parámetro se encarga de prevenir este escenario limitando el número máximo de procesos.
- Idle timeout establece la cantidad de tiempo que esperarán los procesos del pool sin acceder a la base de datos antes de cerrarse. Se considera que si un proceso supera este tiempo de espera, no merece la pena tenerlo abierto ya que está consumiendo recursos, y es mejor cerrarlo. Un tiempo demasiado corto puede por tanto llevar a un uso ineficiente de los recursos, mientras que un tiempo demasiado largo podría llevar a que durante una ráfaga de peticiones se abran nuevos pools y se cierren inmediatamente, cuando posiblemente llegue otra ráfaga similar pronto.
- El parámetro Max Wait Time nos indica cuántos milisegundos pasarán desde que se realiza una petición hasta que se declara en timeout. Si este parámetro es demasiado pequeño, se podría declarar falsos positivos (se dice que una petición es timeout, cuando habiendo esperado un poco más se habría procesado perfectamente y no habría habido problema) Sin embargo, un timeout demasiado largo puede llevar a esperas demasiado largas por parte del cliente cuando se producen bloqueos, pudiendo producirse live-locks e interbloqueos más largos.

Ejercicio número 4:

Localice los siguientes fragmentos de código SQL dentro del proyecto proporcionado (P1-base) correspondientes a los siguientes procedimientos:

El código SQL se encuentra en el fichero visaDAO.java y es el siguiente:

- 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;
}
```

- 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;
}
```

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.

La función errorLog se utiliza en los métodos compruebaTarjeta, getPagos y delPagos. Si realizamos un pago utilizando la página testbd.jsp con la opción de debug activada podemos observar como en el log se escribe información adicional tratada en la función compruebaTarjeta.

Este es nuestro archivo log:

Instance:
Log File:

Log Viewer Results (40)

Records before 108 | Log File Record Numbers 108 through 147 | Records after 147 |

Record Number	Log Level	Message	Logger	Timestamp	Name-Value Pairs
147	SEVERE	[directConnection=false] select idAutorizacion, codRespuesta from pago where idTransaccion = '1' ... (details)		25-feb-2019 11:35:34.371	{levelValue=1000, timeMillis=1551123334371}
146	SEVERE	[directConnection=false] insert into pago(idTransaccion,importe,idComercio,numeroTarjeta) values ('1... (details)		25-feb-2019 11:35:34.369	{levelValue=1000, timeMillis=1551123334369}
145	SEVERE	[directConnection=false] select * from tarjeta where numeroTarjeta='6513 0633 4651 1154' and titular... (details)		25-feb-2019 11:35:34.357	{levelValue=1000, timeMillis=1551123334357}
144	INFO	visiting unvisited references(details)	javax.enterprise.system.tools.deployment.dol	25-feb-2019 11:35:33.582	{levelValue=800, timeMillis=1551123333582}
143	INFO	WebModule[null] ServletContext.log():[INFO] Acceso correcto:/procesapago(details)	javax.enterprise.web	25-feb-2019 11:35:33.401	{levelValue=800, timeMillis=1551123333401}
142	INFO	WebModule[null] ServletContext.log():[INFO] Acceso correcto:/testbd.jsp(details)	javax.enterprise.web	25-feb-2019 11:33:47.865	{levelValue=800, timeMillis=1551123227865}
141	INFO	WebModule[null] ServletContext.log():[INFO] Acceso correcto:/comenzapago(details)	javax.enterprise.web	25-feb-2019 11:31:19.281	{levelValue=800, timeMillis=1551123079281}
140	INFO	Admin Console: Initializing Session Attributes...(details)	org.glassfish.admingui	25-feb-2019 11:29:49.948	{levelValue=800, timeMillis=1551122989948}
139	INFO	Redirecting to /index.jsf(details)	org.glassfish.admingui	25-feb-2019 11:29:49.841	{levelValue=800, timeMillis=1551122989841}

Esta es la información adicional que nos proporciona:

147	SEVERE	[directConnection=false] select idAutorizacion, codRespuesta from pago where idTransaccion = '1' ... (details)
146	SEVERE	[directConnection=false] insert into pago(idTransaccion,importe,idComercio,numeroTarjeta) values ('1... (details)
145	SEVERE	[directConnection=false] select * from tarjeta where numeroTarjeta='6513 0633 4651 1154' and titular... (details)

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()

Como observamos en la siguiente figura vemos que a comprueba tarjeta la hemos añadido como método web y su parámetro como parámetro web

```
*           en la tabla TARJETA fue satisfactoria, false en caso contrario */
@WebMethod(operationName = "compruebaTarjeta")
public boolean compruebaTarjeta(@WebParam(name = "tarjeta")TarjetaBean tarjeta) {
    Connection con = null;
    Statement stmt = null;
```

• realizaPago()

En esta función hemos editado la cabecera de forma similar a la función anterior:

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


Pero además hemos añadido y modificado líneas para que devuelva null en caso de error, como se nos pedía en el enunciado:

```
if (pago.getIdTransaccion() == null) {
    return null;
}

// Registrar el pago en la base de datos
try {

    // Obtener conexion
    con = getConnection();

    // Insertar en la base de datos el pago

    /* TODO Usar prepared statement si
    isPrepared() == true */
    /***/
    if (isPrepared() == true) {
        String insert = INSERT_PAGOS_QRY;
        errorLog(insert);
        pstmt = con.prepareStatement(insert);
        pstmt.setString(1, pago.getIdTransaccion());
        pstmt.setDouble(2, pago.getImporte());
        pstmt.setString(3, pago.getIdComercio());
        pstmt.setString(4, pago.getTarjeta().getNumero());
        ret = null;
        if (!pstmt.execute()
            && pstmt.getUpdateCount() == 1) {
            ret = pago;
        }
    }
    else {
        /***/
        stmt = con.createStatement();
        String insert = getQryInsertPago(pago);
        errorLog(insert);
        ret = null;
    }
}
```

```
// Obtener id.autorizacion
if (ret!=null) {

    /* TODO Permitir usar prepared statement si
    * isPrepared() = true */
    /***/
    if (isPrepared() == true) {
        String select = SELECT_PAGO_TRANSACCION_QRY;
        errorLog(select);
        pstmt = con.prepareStatement(select);
        pstmt.setString(1, pago.getIdTransaccion());
        pstmt.setString(2, pago.getIdComercio());
        rs = pstmt.executeQuery();
    }
    else {
        /***/

        String select = getQryBuscaPagoTransaccion(pago);
        errorLog(select);
        rs = stmt.executeQuery(select);

    }
    /***/
    if (rs.next()) {
        pago.setIdAutorizacion(String.valueOf(rs.getInt("idAutorizacion")));
        pago.setCodRespuesta(rs.getString("codRespuesta"));
    }
    else {
        ret = null;
    }
}

} catch (Exception e) {
    errorLog(e.toString());
    ret = null;
}
```

- **isDebug(), setDebug(), isPrepared(), setPrepared(), isDirectConnection(), setDirectConnection()**

En la siguiente figura observamos lo siguiente:

- Hemos incluido isPrepared() como método web
- Hemos añadido setPrepared() como método web y su parámetro 'prepared' como parámetro web
- Hemos añadido isDebug() como método web para su acceso remoto
- Hemos excluido de los métodos web la versión de setDebug() que recibía un boolean como argumento
- Hemos incluido en los métodos web la versión de setDebug() que recibe como argumento un texto "true" para el valor true y cualquier otra cadena para el valor false. Este argumento a su vez lo hemos añadido como parámetro web
- Hemos añadido override del método isDirectConnection(), que lo único que hace es llamar a su método padre, y lo hemos incluido en los métodos web.
- Hemos hecho lo mismo con set DirectConnection() con la salvedad de que su argumento de entrada se ha añadido también como parámetro web.

```
@WebMethod(operationName = "isPrepared")
public boolean isPrepared() {
    return prepared;
}

@WebMethod(operationName = "setPrepared")
public void setPrepared(@WebParam(name = "prepared")boolean prepared) {
    this.prepared = prepared;
}
/*****/

/**
 * @return the debug
 */
@WebMethod(operationName = "isDebug")
public boolean isDebug() {
    return debug;
}

/**
 * @param debug the debug to set
 */
@WebMethod(exclude = true)
//@WebMethod(operationName = "setDebug")
public void setDebug(@WebParam(name = "debug")boolean debug) {
    this.debug = debug;
}

/**
 * @param debug the debug to set
 */
//@WebMethod(exclude = true)
@WebMethod(operationName = "setDebug")
public void setDebug(@WebParam(name = "debug")String debug) {
    this.debug = (debug.equals("true"));
}

@WebMethod(operationName = "isDirectConnection")
@Override
public boolean isDirectConnection() {
    return super.isDirectConnection();
}

@WebMethod(operationName = "setDirectConnection")
@Override
public void setDirectConnection(@WebParam(name = "directConnection")boolean directConnection) {
    super.setDirectConnection(directConnection);
}

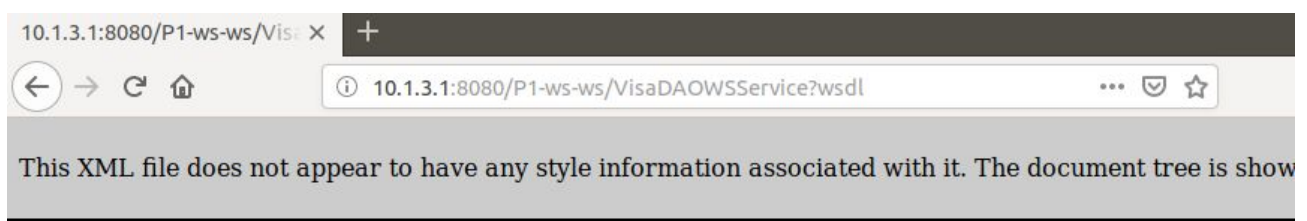
/****
```

- ¿Por qué se ha de alterar el parámetro de retorno del método realizaPago() para que devuelva el pago el lugar de un boolean?

Puesto que ahora nuestro servicio web utiliza SOAP, la aplicación se tendrá que ejecutar en dos máquinas: la del cliente y la del servidor. Esto significa que para que el cliente pueda ver los datos del pago que ha realizado, el servidor necesita pasarle el objeto del pago, que es el que almacena la información.

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:



```

10.1.3.1:8080/P1-ws-ws/Visa x +
10.1.3.1:8080/P1-ws-ws/VisaDAOWSService?wsdl
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+006
-->
<!--
  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+006
-->
<definitions targetNamespace="http://dao.visa.ssii2/" name="VisaDAOWSService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://dao.visa.ssii2/" schemaLocation="http://10.1.3.1:8080/P1-ws-
ws/VisaDAOWSService?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">

```

Observamos en la figura anterior que la dirección IP es la correcta, por su extensión, hemos incluido el fichero WSDL aparte en un archivo llamado wsdl.txt entregado junto con esta memoria.

En general observamos lo siguiente:

- Hay una correspondencia entre métodos y elementos XML “message”
- Por cada método hay dos mensajes (message), uno de petición que tiene el mismo nombre del método y otro de respuesta (response)
- Todos estos mensajes tienen un campo llamado “parameters” que tiene un atributo element que representa el paso de argumentos y de respuestas, como una estructura de serialización
- Por cada método Java se definen más abajo una operación, que tiene petición y respuesta, que se compone de los mensajes definidos más arriba

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

En nuestro caso en el siguiente fichero: <http://10.1.3.1:8080/P1-ws-ws/VisaDAOWSService?xsd=1>

Es decir, en el fichero que aparece en <types> dentro del WSDL

- ¿Qué tipos de datos predefinidos se usan?

Parámetros complejos que contienen estructuras de tipos simples

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

Los que aparecen en la primera lista del link de dos preguntas atrás, bajo la etiqueta <element>, son los tipos que sirven para el paso de mensajes:

```
<types>
  <xsd:schema>
    <xsd:import
      namespace="http://dao.visa.ssii2/"schemaLocation="http://10.1.3.1:8080/P1-ws-ws/VisaDAOWSService?xsd=1"/>
    </xsd:schema>
  </types>
```

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

```
<operation>
```

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

```
<message>
```

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

```
<binding>
```

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

```
<service> y dentro de ella <port> y dentro de ella a su vez <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.

Observamos en la siguiente figura cómo tiene en cuenta que ahora se devuelve un pago, y toma datos del mismo:

```
try {
    pago=dao.realizaPago(pago);
}
catch (javax.xml.ws.WebServiceException e){
    e.printStackTrace();
}

if (pago==null ||
    pago.getCodRespuesta()=="999") {
    enviaError(new Exception("Pago incorrecto"), request, response);
    return;
}

request.setAttribute(ComienzaPago.ATTR_PAGO, pago);
if (sesion != null) sesion.invalidate();
reenvia("/pagoexito.jsp", request, response);
return;
}
```


- Las llamadas remotas pueden generar nuevas excepciones que deberán ser tratadas en el código cliente.

Por otro lado aquí observamos dos lugares en los que hemos controlado las excepciones nuevas (aunque no son los únicos):

```
try{
    VisaDAOWSService service = new VisaDAOWSService();
    dao = service.getVisaDAOWSPort ();
    BindingProvider bp = (BindingProvider) dao;
    bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
        getServletContext().getInitParameter("serverAddress"));
}
catch (javax.xml.ws.WebServiceException e){
    e.printStackTrace();
}

try {
    pago=dao.realizaPago(pago);
}
catch (javax.xml.ws.WebServiceException e){
    e.printStackTrace();
}
```

Ejercicio número 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.

Esto es lo que hemos añadido al fichero web.xml:

```
<context-param>
    <param-name>serverAddress</param-name>
    <param-value>http://10.1.3.1:8080/P1-ws-ws/VisaDAOWSService</param-value>
</context-param>
```

Y así es como hemos obtenido el parámetro del contexto:

```
bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
    getServletContext().getInitParameter("serverAddress"));
```

Ejercicio número 10:

Siguiendo el patrón de los cambios anteriores, adaptar las siguientes clases cliente para que toda la funcionalidad de la página de pruebas testbd.jsp se realice a través del servicio web. Esto afecta al menos a los siguientes recursos:

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

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

Lo primero que hemos hecho ha sido añadir el siguiente set de instrucciones en todos los ficheros que necesiten conectarse al servidor:

```
try{
    VisaDAOWSService service = new VisaDAOWSService();
    dao = service.getVisaDAOWSPort ();
    BindingProvider bp = (BindingProvider) dao;
    bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
        getServletContext().getInitParameter("serverAddress"));
}
catch (javax.xml.ws.WebServiceException e){
    e.printStackTrace();
}
```

Después hemos cambiado en getPagos() la línea que llama al dao, para que espere recibir un ArrayList<PagoBean> y lo convierta a array:

```
/* Petición de los pagos para el comercio */
PagoBean[] pagos = dao.getPagos(idComercio).toArray(new PagoBean[0]);
```

Por el lado del servidor, getPagos() ahora devuelve un ArrayList<PagoBean> que es compatible con Java RMI:

```
return new ArrayList<PagoBean>(Arrays.asList(ret));
```

Ejercicio número 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.

Este ha sido el comando que hemos utilizado para la importación manual del WSDL:

```
wsimport -d build/client/WEB-INF/classes -p ssii2.visa
http://10.1.3.1:8080/P1-ws-ws/VisaDAOWSService?wsdl
```

Se han creado múltiples clases en el directorio indicado en el comando. Por cada método del WebService se han creado una clase que contiene la información del mensaje de petición y otra con el contenido de las respuestas. Por tanto dos clases por cada petición posible al servidor. Estas clases se corresponden además con los messages que aparecían en el documento WSDL

Ejercicio número 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 Java raíz (ssii2) ya está definido como \${paquete}
- La URL ya está definida como \${wsdl.url}

Hemos añadido lo siguiente al build.xml para que generar-stubs importe el wsdl automáticamente:

```
<exec executable="wsimport">
  <arg line="-d ${build.server}/WEB-INF/classes" />
  <arg line="-p ${paquete}.visa" />
  <arg line="${wsdl.url}" />
</exec>
```

Ejercicio número 13:

• Realice un despliegue de la aplicación completo en dos nodos tal y como se explica en la Figura 8. Habrá que tener en cuenta que ahora en el fichero build.properties hay que especificar la dirección IP del servidor de aplicaciones donde se desplegará la parte del cliente de la aplicación y la dirección IP del servidor de aplicaciones donde se desplegará la parte del servidor. Las variables as.host.client y as.host.server deberán contener esta información.

• Probar a realizar pagos correctos a través de la página testbd.jsp. Ejecutar las consultas SQL necesarias para comprobar que se realiza el pago. Anotar en la memoria práctica los resultados en forma de consulta SQL y resultados sobre la tabla de pagos.

Incluye evidencias en la memoria de la realización del ejercicio.

Hemos rellenado el formulario con los siguientes datos para crear un pago nuevo:

Pago con tarjeta

Proceso de un pago

Id Transacción:	<input type="text" value="9"/>
Id Comercio:	<input type="text" value="9"/>
Importe:	<input type="text" value="1"/>
Numero de visa:	<input type="text" value="6513 0633 4651 1154"/>
Titular:	<input type="text" value="Gabriel Locke Martinez"/>
Fecha Emisión:	<input type="text" value="04/08"/>
Fecha Caducidad:	<input type="text" value="02/20"/>
CVV2:	<input type="text" value="681"/>
Modo debug:	<input checked="" type="radio"/> True <input type="radio"/> False
Direct Connection:	<input checked="" type="radio"/> True <input type="radio"/> False
Use Prepared:	<input type="radio"/> True <input checked="" type="radio"/> False
<input type="button" value="Pagar"/>	

Y la respuesta ha sido satisfactoria:

Pago con tarjeta

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

idTransaccion: 9
idComercio: 9
importe: 1.0
codRespuesta: 000
idAutorizacion: 3

[Volver al comercio](#)

De hecho, si realizamos la petición del listado de pagos vemos que aparece:

Pago con tarjeta

Lista de pagos del comercio 9

idTransaccion	Importe	codRespuesta	idAutorizacion
9	1.0	000	3

[Volver al comercio](#)

Y si lo eliminamos:

Pago con tarjeta

Se han borrado 1 pagos correctamente para el comercio 9

[Volver al comercio](#)

Se borra correctamente, no aparece ya en la lista:

Pago con tarjeta

Lista de pagos del comercio 9

idTransaccion	Importe	codRespuesta	idAutorizacion
---------------	---------	--------------	----------------

[Volver al comercio](#)

Cuestión número 1:

Teniendo en cuenta el diagrama de la Figura 3, indicar las páginas html, jsp y servlets por los que se pasa para realizar un pago desde pago.html, pero en el caso de uso en que se introduce una tarjeta cuya fecha de caducidad ha expirado.

1. pago.html (HTML)
2. Comienza Pago (servlet)
3. formdatosvisa.jsp (JSP)
4. Procesa Pago (servlet)
5. formdatosvisa.jsp (JSP)
6. error/muestraerror.jsp (JSP)

Cuestión número 2:

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

Si nos centramos en los datos que se solicitan al usuario:

El servlet encargado de solicitar la información sobre el pago con tarjeta cuando se usa pago.html es "Comienza pago".

Si nos centramos en los datos que se solicitan a la página anterior:

Los servlets encargados de solicitar la información sobre el pago con tarjeta cuando se usa pago.html son "Comienza pago" y "Procesa pago".

Cuestión número 3:

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

Si nos centramos en los datos que se solicitan al usuario:

El servlet "Comienza pago" solicita los datos de la tarjeta de crédito: el número de la tarjeta, el nombre del titular, la fecha de inicio de su validez, la fecha de fin de su validez y el código verificación. Estos datos los muestra el servlet "Procesa pago", así como los datos de la compra: el id de la transacción, el id del comercio, el importe, el id de respuesta y el id de autorización.

Si nos centramos en los datos que se solicitan a la página anterior:

El servlet "Comienza pago" solicita el id de transacción, el id de comercio y el importe del pago a la página pago.html y el servlet "Procesa pago" solicita los datos de la tarjeta de crédito: número de la tarjeta, nombre del titular, fecha de inicio de su validez, fecha de fin de su validez y código verificación.

La información que manejan ambos servlets la comparten mediante una variable de sesión ATTR_PAGO, que almacena todos los datos referentes al pago.

Cuestión número 4:

Enumere las diferencias que existen en la invocación de servlets, a la hora de realizar el pago, cuando se utiliza la página de pruebas extendida testbd.jsp frente a cuando se usa pago.html. ¿Podría indicar por qué funciona correctamente el pago cuando se usa testbd.jsp a pesar de las diferencias observadas?

Al realizar el pago desde pago.html, el fichero HTML llama al servlet "Comienza pago" al enviar los datos: id de transacción, id de comercio e importe del pago. Este servlet solicita los datos del cliente para poder realizar el pago y envía toda la información al servlet "Procesa pago".

Por otro lado, si realizamos un pago desde la página de pruebas extendida testbd.jsp, se llama al servlet "Procesa pago" directamente, pues es este el que solicita tanto los datos del comercio como los datos del cliente en un único paso.

Ambas formas de pago funcionan ya que en el servlet "Procesa pago" se comprueba si hay una sesión iniciada y si hay datos en la variable ATTR_PAGO. Esto significa que coge los datos cuando existen (caso pago.html) o crea un pago con todos los datos a rellenar en el formulario cuando no existen (caso testbd.jsp).