Práctica 1: Arquitectura de JAVA EE (Segunda parte)

Índice

1. Objetivos

El objetivo fundamental de esta segunda parte de la práctica 1 es continuar profundizando en elementos de la arquitectura JAVA EE. Se pretenden alcanzar los siguientes subobjetivos:

- Lógica de negocio: EJBs *session stateless* locales
- Gestores de colas y API JMS para la comunicación entre aplicaciones JAVA EE
- *Message Driven Beans*, o MDBs

Se asume que el alumno ya tiene finalizada la primera parte de la práctica 1.

La dedicación estimada para esta práctica es de 4 horas presenciales y 4 horas no presenciales por estudiante.

2. Material entregado

El material entregado en esta práctica se puede descargar de la página de web del laboratorio. Consta de los siguientes archivos:

- *P1-ejb-base.tgz: Proyecto de ejemplo con estructura para EJB Session Stateless* (con interfaz local)
- *P1-ejb-transaccional-base.tgz: Modificaciones para comprobar la transaccionalidad de un EJB*
- *P1-jms-base.tgz: Proyecto de ejemplo con cliente JMS y servidor MDB para envío de mensajes*

3. Entorno

El entorno de realización de la práctica es el indicado en la práctica 0. Para realizar las prácticas en el laboratorio se podrá utilizar sólo la partición de Linux.

4. Componentes de Glassfish de esta práctica

En la Figura 1, se puede ver la arquitectura de una instancia del servidor Glassfish. En este práctica van a entrar en juego los elementos que se muestran enmarcados en color rojo: El *EJB container*, encargado de la ejecución y ciclo de vida de componentes EJB, y el *Java Message Service* es el componente middleware encargado de intermediar con los proveedores de mensajería, como el gestor de colas incluido en el propio Glassfish, para la intercomunicación de aplicaciones JAVA EE.

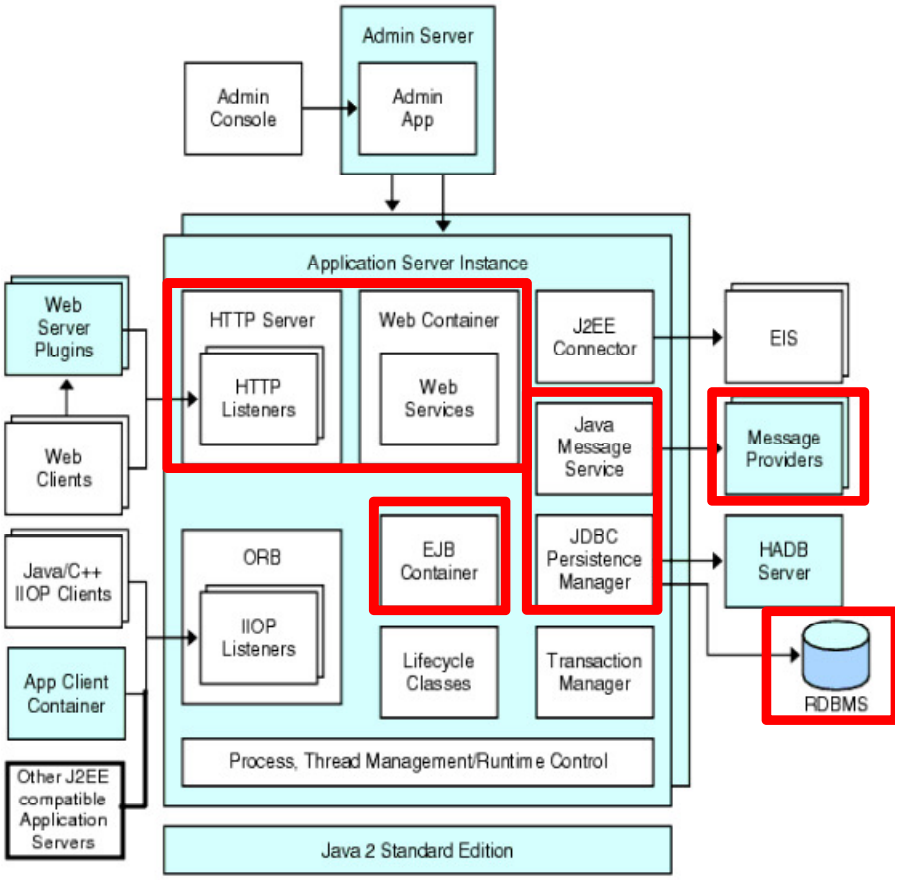

Figura 1 Módulos de la arquitectura de Glassfish relacionados con el contenido de la práctica

5. Primera parte: EJBs *stateless* locales

Antes de comenzar a realizar esta parte de la práctica se recomienda al alumno que lea el apéndice sobre EJBs, para conocer en mayor detalle estos componentes y poder comprender las distintas acciones que se llevarán a cabo a lo largo de la práctica.

El objetivo de este apartado será convertir la clase VisaDAOWS realizado en la primera parte de la práctica 1 en un EJB *stateless* al que se accederá a través de su interfaz local. Esto quiere decir que ambos, cliente y servidor, residirán en el mismo servidor. Existe la posibilidad de usar la interfaz remota, para distribuir este EJB en un servidor distinto, que se llevará a cabo en el siguiente ejercicio.

Para conseguir esto, se mostrará también cómo desplegar aplicaciones web que contienen tanto servlets, JSPs y clases java como EJBs, generando un archivo empaquetado de *Enterprise Application* (.ear)

5.1 Material de partida

El alumno dispone del archivo P1-ejb-base.tgz, que contiene la siguiente estructura de directorios y archivos:

P1-ejb-base	
|-- build.xml / build.properties	Archivo ant para construir P1-ejb
|-- conf	
|-- application	Configuración de la aplicación conjunta
|-- META-INF	
|-- application.xml	Descriptor de despliegue de la aplicación
|-- server	
|-- META-INF	
|-- MANIFEST.MF	
|-- sun-ejb-jar.xml	Descriptor de despliegue del EJB
|-- src	
|-- server	Fuentes del servidor
|-- ssii2	
|-- visa	
|-- VisaDAOLocal.java	Interfaz local para el EJB VisaDAOBean

5.1.1 Combinar proyectos P1-ws y P1-ejb-base como P1-ejb

A continuación el alumno combinará el árbol de directorios de P1-ws y P1-ejb-base de la siguiente forma:

Importante: Todas las modificaciones añadidas en apartados anteriores sobre el build.xml se perderán al descomprimir el nuevo contenido.

- Tomará P1-ws como punto de partida. Se deberá copiar el directorio como P1-ejb
  - cp –r P1-ws P1-ejb
- Sobre este directorio se descomprimirá el fichero P1-ejb-base.tgz, de esta forma:
  - cd P1-ejb
  - tar -xzvf /ruta/a/P1-ejb-base.tgz
- A continuación eliminaremos el directorio (ya innecesario) conf/serverws
- Por último, ejecutaremos la regla de limpieza para eliminar otros restos del webservice:
  - ant limpiar-todo

Para seguir la nomenclatura estándar de J2EE para los archivos y clases que implementan EJBs, **renombraremos nuestra clase `VisaDAOWS` a `VisaDAOBean`**, haciendo el mismo cambio de nombre en el archivo que la contiene. Asimismo, es necesario renombrar el constructor (es idéntico y ha de coincidir con el nombre de la clase). **Se eliminarán las anotaciones** introducidas en VisaDAOWS en la primera parte de la práctica para evitar exportar los métodos como un servicio web.

5.2 Creación de la aplicación Visa con EJB local

Para crear una aplicación Visa que utilice un EJB local se realizan distintos pasos. Todos ellos se encuentran automatizados mediante *ant*, gobernado por el archivo de configuración *build.xml*. Dicho archivo contiene los siguientes objetivos:

Para generar el servidor:

1. compilar-servidor
2. empaquetar-servidor

Y para generar el cliente:

3. compilar-cliente
4. empaquetar-cliente

Tras realizar estos pasos, ya se puede empaquetar la aplicación final, que contiene tanto el cliente como el servidor, y desplegarla en el entorno elegido

5. empaquetar-aplicacion
6. desplegar

A continuación se van a ir detallando cada uno de los pasos, indicándose todas las operaciones que se deben realizar para lograr que funcionen correctamente.

5.2.1 Codificar y compilar el servidor

El primer paso para obtener el EJB es codificar las clases relativas al EJB, los posibles interfaces y la implementación de los mismos. Para ello se emplean un conjunto de anotaciones, que serán empleadas posteriormente en el proceso de generación de código.

Con el código de la práctica se entrega el archivo VisaDAOLocal.java, que contiene la definición de la interfaz local que implementará dicho EJB.

Cuestión 1: Editar el archivo *VisaDAOLocal.java* y comprobar la definición de dicha interfaz. Anote sus comentarios en la memoria.

Tras esto es necesario definir como EJB sesión *stateless* el bean VisaDAOBean. Para ello, en el archivo VisaDAOBean.java introducimos las siguientes modificaciones:

Ejercicio 1: Introduzca las siguientes modificaciones en el *bean* VisaDAOBean para convertirlo en un EJB *stateless* con interfaz local:

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

```
...
import javax.ejb.Stateless;
...

@Stateless(mappedName="VisaDAOBean")
public class VisaDAOBean extends DBTester implements VisaDAOLocal {
...
```

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

Para la compilación de las clases del VisaDAOBean se ejecutará el programa *ant* con el objetivo *compilar-servidor*:

```
ant compilar-servidor
```

5.2.2 Empaquetar el servidor

El siguiente paso consiste en empaquetar los archivos del servicio en un archivo .jar. Para esto se ejecutará nuevamente la herramienta ant con el objetivo *empaquetar-servidor*:

```
ant empaquetar-servidor
```

Esto generará un archivo dentro del directorio **dist/server** llamado **P1-ejb.jar**.

Dentro del archivo empaquetado se ha introducido el descriptor del EJB, proporcionado en el código de la práctica en el archivo `conf/server/META-INF/sun-ejb-jar.xml`. Este archivo es muy simple, ya que todas las características del despliegue del EJB han sido generadas automáticamente por el proceso de compilación y empaquetado del código gracias a las anotaciones incluidas en el código fuente. En versiones anteriores de la especificación EJB, todas las características del despliegue debían darse de modo explícito dentro de este descriptor.

5.2.3 Codificar y compilar el cliente (web)

En este paso prepararemos el cliente web para que acceda al EJB local que se acaba de crear.

Ejercicio 2: Modificar el *servlet* ProcesaPago para que acceda al EJB local. Para ello, modificar el archivo ProcesaPago.java de la siguiente manera:

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

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

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

```
~~import ssii2.visa.VisaDAOWSService; // Stub generado automáticamente~~
~~import ssii2.visa.VisaDAOWS; // Stub generado automáticamente~~
```

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

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

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

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

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

Para la compilación del cliente se ejecutará el programa *ant* con el objetivo *compilar-cliente*:

```
ant compilar-cliente
```

Notar que, en este caso, la compilación del cliente incluye en el *classpath* las clases previamente compiladas en el servidor, ya que alguna de ellas (PagoBean, TarjetaBean) las necesita utilizar como tales para su operación. Dado que son locales, el acceso a las mismas debe ser directo. Esto se puede comprobar dentro del archivo *build.xml*:

```
<path id="compile.client.classpath">
   <pathelement location="${as.lib}/javaee.jar"/>
   <pathelement location="${build.server}"/>
</path>
```

5.2.4 Empaquetar el cliente (web)

Una vez compilado el cliente se puede empaquetar en un .war, como hace el objetivo *empaquetar-cliente*:

```
ant empaquetar-cliente
```

Con esto se obtiene un archivo llamado **P1-ejb-cliente.war** en el subdirectorio *dist*, el cual formará parte de la aplicación web.

5.2.4.1 Empaquetar la aplicación (web)

Tras compilar y empaquetar la parte web de la aplicación (cliente) y el EJB que utiliza (servidor), el siguiente paso es empaquetar ambos formando una **aplicación de empresa**, con extensión **ear**. Un Archivo de Empresa (Enterprise Archive, o EAR), es un formato de fichero usado en Java EE para empaquetar uno o más módulos en un archivo único de manera que el despliegue de varios módulos en un servidor de aplicaciones se realice simultánea y coherentemente. También contienen ficheros XML (descriptores) que describen como hacer el despliegue de dichos módulos (en el directorio de metainformación, META-INF). En relación al empaquetamiento, un fichero ear es un fichero jar estándar, con la extensión cambiada.

Este paso se logra mediante el objetivo *empaquetar-aplicacion*, contenido en el archivo build.xml.

```
<target name="empaquetar-aplicacion" description="Genera un .ear de toda la
aplicación">
  <delete file="${dist}/${ear}" />
  <ear destfile="${dist}/${ear}" appxml="${conf.application}/META-
INF/application.xml">
     <fileset dir="${dist.client}" includes="*.war" />
     <fileset dir="${dist.server}" includes="*.jar" />
  </ear>
</target>
```

La aplicación de empresa necesita un descriptor de despliegue, que en este caso es el archivo application.xml suministrado en el código de la práctica.

Cuestión 2: Editar el archivo *application.xml* y comprobar su contenido. Verifique el contenido de todos los archivos .jar / .war / .ear que se han construido hasta el momento (empleando el comando jar –tvf). Anote sus comentarios en la memoria.

5.2.4.2 Desplegar la aplicación completa

Tras realizar el empaquetado de la aplicación, vamos a proceder a desplegarla en el entorno que se muestra en la Figura 2, que es el mismo que se utilizó en la primera parte de la práctica 1:

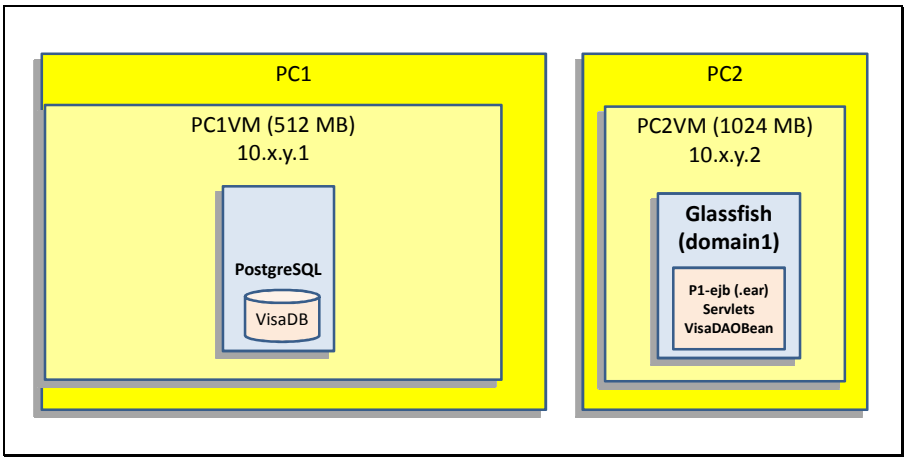

Figura 2 Entorno de despliegue de la aplicación

Sobre el servidor de aplicaciones residente en PC2VM se desplegará la aplicación, accediendo a la base de datos que residirá en PC1VM.

Ejercicio 3: Preparar los PCs con el esquema descrito y realizar el despliegue de la aplicación:

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

Desplegar la aplicación de empresa

```
ant desplegar
```

Tras el despliegue, comprobar en la aplicación de administración de Glassfish en PC2VM el despliegue de la aplicación. En este caso, la aplicación no se ha desplegado, como en las prácticas anteriores, como *Web Application*, sino que se encuentra bajo el apartado de *Enterprise Applications*, como muestra la Figura 3:

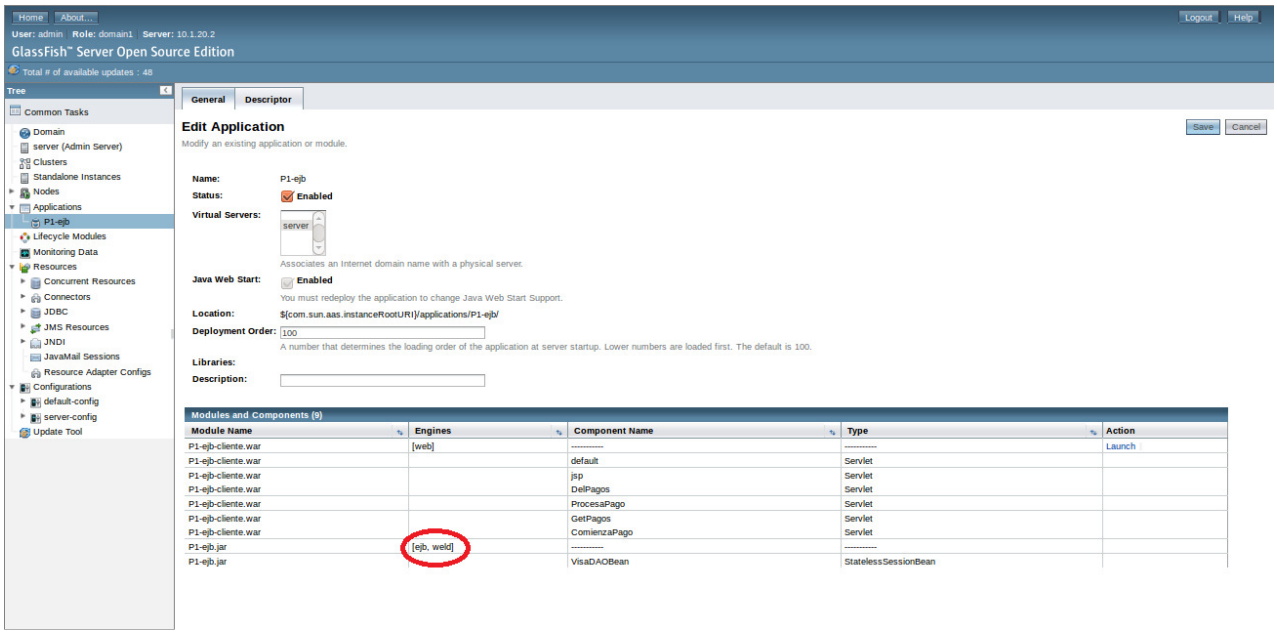

Figura 3 Aplicación de administración de Glassfish en PCVM2 tras el despliegue.

Ejercicio 4: Comprobar el correcto funcionamiento de la aplicación mediante llamadas directas a través de las páginas *pago.html* y *testbd.jsp (sin directconnection)*. Realice un pago. Lístelo. Elimínelo. Téngase en cuenta que la aplicación se habrá desplegado bajo la rulta /P1-ejb-cliente

6. Segunda parte: EJBs *stateless* con invocación remota

El objetivo de este apartado es implementar un cliente remoto del Enterprise Java Bean de sesión sin estado utilizado en P1-ejb. Como material de partida se utilizará P1-ejb para desarrollar el servidor. P1-base se usará para desarrollar el cliente remoto de los EJBs.

6.1 Creación de la aplicación Visa con EJB remoto

Para crear una aplicación Visa que permita la invocación remota de los métodos de los EJB se han de realizar distintos pasos. A continuación se van a ir detallando cada uno de los pasos, indicándose todas las operaciones que se deben realizar.

6.1.1 Habilitar la invocación remota en el EJB en la P1-ejb

Se partirá del código P1-ejb utilizado previamente, copiándolo al directorio P1-ejb-servidor-remoto:

```
cp -pr P1-ejb P1-ejb-servidor-remoto
```

Definir la interfaz remota del EJB: Este paso consiste en copiar VisaDAOLocal.java a VisaDAORemote.java, cambiar el nombre de la interfaz a VisaDAORemote y cambiar la anotación @Local por @Remote.

Añadir import javax.ejb.Remote; y quitar import javax.ejb.Local

Hacer que VisaDAOBean implemente ambas interfaces, la local y la remota.

6.1.2 Marcar como serializables los objetos intercambiados entre cliente y servidor

Se deberá marcar como serializables PagoBean y TarjetaBean. Para ello se editarán los ficheros PagoBean.java y TarjetaBean.java y se marcarán los objetos correspondientes como implementadores de la interfaz java.io.Serializable.

Ejercicio 5: Realizar los cambios indicados en P1-ejb-servidor-remoteo y preparar los PCs con el esquema de máquinas virtuales indicado y compilar, empaquetar y desplegar de nuevo la aplicación P1-ejb como servidor de EJB remotos de forma similar a la realizada en el Ejercicio 3 con la Figura 2 como entorno de desplieque. Esta aplicación tendrá que desplegarse en la máquina virtual del PC2.

Se recomienda replegar la aplicación anterior (EJB local) antes de desplegar ésta.

6.1.3 Construir el cliente remoto de EJB partiendo de P1-base

Copiar P1-base a P1-ejb-cliente-remoto y cambiar el nombre de la aplicación en P1-ejb-cliente-remoto/build.properties a P1-ejb-cliente-remoto.

Eliminar el directorio ssii2/visa/dao pues ahora toda la lógica se invocará de forma remota.

Se deberá marcar como serializables PagoBean y TarjetaBean. Para ello se editarán los ficheros PagoBean.java y TarjetaBean.java y se marcarán los objetos correspondientes como implementadores de la interfaz java.io.Serializable.

Copiar la interfaz remota VisaDAORemote.java de P1-ejb-servidor-remoto a P1-ejb-cliente-remoto/src/ssii2/visa.

En los servlets que invocan la lógica de negocio (procesapago.java, getpagos.java y delpagos.java) eliminar la declaración de VisaDAO dao, e insertar una referencia al objeto remoto obtenida por inyección. Para ello se hará algo parecido a lo hecho en P1-ejb. Es decir, se declarará un atributo de cada servlet de la forma:

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

Habrá que importar

```
import javax.ejb.EJB;
import ssii2.visa.VisaDAORemote;
```

Crear el un fichero glassfish-web.xml en web/WEB-INF que especificará como se resolverán las referencias a los EJBs remotos y se invocarán los métodos remotos. En este caso usaremos IIOP. Para ello, editar dicho fichero e introducir las líneas:

```
<?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE glassfish-web-app PUBLIC "-//GlassFish.org//DTD
  GlassFish Application Server 3.1 Servlet 3.0//EN"
  "http://glassfish.org/dtds/glassfish-web-app_3_0-1.dtd">
  <glassfish-web-app>
  <ejb-ref>
          <ejb-ref-name>VisaDAOBean</ejb-ref-name>
         <jndi-name>corbaname:iiop:10.x.y.2:3700#java:global/P1-ejb/P1-
ejb/VisaDAOBean!ssii2.visa.VisaDAORemote</jndi-name>
  </ejb-ref>
  </glassfish-web-app>
```

Lo que sigue a corbaname:iiop:10..x.y.2:3700# es el nombre JNDI que se asigna al EJB en P1-ejb-servidor-remoto, al hacer el despliegue. Se puede observar en el log de la MV donde se hace el despliegue de P1-ejb-servidor-remoto. En este caso indicamos también la interfaz que utilizaremos (un EJB puede tener varias). La entrada ejb-ref-name hace referencia al nombre del EJB, como se ha indicado en la anotación @EJB. La dirección 10.x.y.2 sería la dirección de la máquina virtual donde estaría el servidor EJB. Es decir, donde se ha desplegado P1-ejb. El puerto 3700 es el puerto IIOP de glassfish.

Cambiar las direcciones IP especificadas en los ficheros P1-ejb-cliente-remoto/build.properties y P1-ejb-cliente-remoto/posgresql.properties para que apunten a otra MV, por ejemplo la 10.x.y.1, y desplegar. Conectarse a 10.x.y.1:8080/P1-ejb-cliente-remoto y probar a hacer un pago.

Ejercicio 6: Realizar los cambios comentados en la aplicación P1-base para convertirla en P1-EJB-cliente-remoto y compilar, empaquetar y desplegar de nuevo la aplicación en otra máquina virtual distinta a la de la aplicación servidor, es decir, esta aplicación cliente estará desplegada en la MV del PC1 tal y como se muestra en la Figura 2.Conectarse a la aplicación cliente y probar a realizar un pago. Comprobar los resultados e incluir en la memoria evidencias de que el pago ha sido realizado de forma correcta.

7. Tercera parte: Transaccionalidad en los EJBs

En esta parte de la práctica se va a trabajar con la razón principal para utilizar EJBs en lugar de cualquier otro componente para realizar funciones de lógica de negocio (*Model* en terminología MVC): su funcionamiento transaccional.

En la configuración de los EJBs que estamos manejando, cada llamada de un cliente a un método de un EJB de sesión es una transacción. La llamada es, por tanto, atómica:

- Si el método termina correctamente, todas las acciones que ha realizado se validan (es decir, se realiza un *commit*)
- Si el método no termina correctamente, lo que se manifiesta porque lanza una excepción de sistema, todas las acciones que ha realizado se deshacen (es decir, se realiza un *rollback*).

En el modelo de EJB que se ha mostrado en las partes anteriores de la práctica, el proceso de gestión de la transaccionalidad se realiza por parte del contenedor del EJB. Es el modelo más sencillo de trabajo, aunque no el único.

En esta parte de la práctica se va a modificar ligeramente el contenido del método *realizaPago* para que se pueda comprobar su funcionamiento transaccional.

Antes de comenzar la ejecución este ejercicio, elimine la configuración de la base de datos y repliegue las aplicaciones P1-ejb cliente y servidor, para garantizar el correcto despliegue del código a realizar a continuación:

```
ant replegar
ant unsetup-db
```

La arquitectura de despliegue de esta parte de la práctica es la misma que se empleó para la parte primera (ver Figura 2)

7.1 Material de partida

El alumno dispone del archivo P1-ejb-transaccional-base.tgz, que contiene la siguiente estructura de archivos:

P1-ejb-transaccional-base	
|-- sql	
|-- create.sql	**Creación de la base de datos – nueva columna TARJETA.saldo**
|-- drop.sql	Borrado de la base de datos
|-- insert.sql	Inserción de registros de tarjetas

7.1.1 Combinar P1-ejb y P1-ejb-transaccional

A continuación el alumno combinará el árbol de directorios de P1-ejb y P1-ejb-transaccional de la siguiente forma:

- Copiamos el directorio del proyecto anterior con el nuevo nombre: P1-ejb-transaccional
  - cp –r P1-ejb P1-ejb-transaccional
- Sobre este directorio se descomprimirá el fichero P1-ejb-transaccional-base.tgz, de esta forma:
  - cd P1-ejb-transaccional
  - tar -xzvf /ruta/a/P1-ejb-transaccional-base.tgz
- Por último, ejecutaremos la regla de limpieza para eliminar otros restos anteriores:
  - ant limpiar-todo

7.2 Modificación de la aplicación VISA

Con el fin de demostrar la transaccionalidad del EJB `VisaDAOBean`, se ha modificado la base de datos VISA, añadiendo un campo más en la tabla de tarjetas: el saldo asociado a la cuenta. Este campo ya se define en el archivo create.sql proporcionado en esta parte de la práctica:

```
saldo double precision not null
```

El archivo `insert.sql` también se ha actualizado para que se carguen todos los registros de la tabla con un valor por defecto en dicho campo igual a 1000.0.

Para utilizar este campo en la aplicación VISA se realizarán las siguientes modificaciones:

Ejercicio 7: Modificar la aplicación VISA para soportar el campo saldo:

Archivo TarjetaBean.java:

- Añadir el atributo saldo y sus métodos de acceso:
  ```
  private double saldo;
  ```

Archivo VisaDAOBean.java:

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

Ejercicio 8: Desplegar y probar la nueva aplicación creada.

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

8. Cuarta parte: MDBs y colas de mensajes

8.1 Envío de mensajes: JMS

La comunicación de procesos a través de mecanismos RPC empleados hasta ahora, tales como los servicios web o EJBs, tienen la particularidad de que son *síncronos*. Esto quiere decir que el proceso llamante esperará hasta que la operación haya sido recibida y procesada por el servidor y no continuará su flujo de ejecución hasta haberse completado toda la operación y retornado el control desde el servidor.

Esto puede ser conveniente en muchos casos, pero no siempre es la mejor manera de comunicar un cliente y un servidor. Por ejemplo, si el lado cliente tiene que procesar muchas peticiones seguidas puede relegarse su procesamiento desacoplando el cliente y el servidor mediante el envío de mensajes a un "destino de mensaje" intermedio. De este modo el retraso en el procesamiento de una petición afecta en menor medida a las siguientes.

La *Java Messaging* API (JMS) proporciona un mecanismo para el envío de mensajes entre aplicaciones Java EE. Ellas no se comunican *directamente*, en su lugar los *productores de mensajes* envían mensajes a un "destino" y los *consumidores de mensajes* reciben de ese destino.

Los "destinos de mensajes" pueden ser:

- Colas de mensajes (*message queue* o *MQ*): Cuando la comunicación es *punto a punto* (PTP).

- Tema de mensaje (*message topic*) cuando se emplea comunicación mediante *publicación/subscripción*

En este apartado vamos a introducir el uso del primer mecanismo, colas de mensaje con el fin de añadir un nuevo nivel de complejidad

8.2 El gestor de colas

Glassfish tiene un gestor de colas integrado en el servidor de aplicaciones. No obstante, es posible usar JMS para conectar con gestores de cola externos tales como WebSphere MQ u otros.

La semántica de una cola de mensajes es:

- **Operación SEND / PUT**: El proceso cliente escribe un mensaje en una cola de mensajes, sin esperar a que el servidor atienda la petición. Puede inmediatamente atender nuevas operaciones.
- **Operación RECEIVE / GET**: El proceso servidor lee un mensaje de la cola. Procesa este mensaje y escribe su respuesta (si procede) en otra cola.

8.2.1 Creación de la factoría de conexión

Una factoría de conexión o *connection factory* es una clase Java especializada en la instanciación de conexiones a destinos JMS, así como su reutilización en forma de *pools*. Puede emplearse tanto para conexiones a colas de mensajes o temas de mensajes, aunque en este ejemplo vamos a crear únicamente colas.

Para crearla, accederemos a la consola de administración -> *Resources* -> JMS Resources -> *Connection Factories*

Pulsando el botón "*New JMS Connection Factory*" accederemos a una pantalla como la de la **Figura 4**. Deberemos rellenar los siguientes campos:

- **Pool name** (o nombre del pool): Estableceremos un nombre que, preferentemente, empiece por jms/nombreDelPool para facilitar su localización por JNDI
- **Resource Type:** Tipo de recurso. Emplearemos javax.jms.QueueConnectionFactory puesto que queremos crear conexiones a *colas de mensajes*. Otros posibles valores son: *TopicConnectionFactory* o el más genérico *ConnectionFactory* que permitirá dar a este *connection factory* ambos usos.
- **Pool settings:** Diversos parámetros del pool tales como el número de conexiones, valor máximo, incremento de conexiones etc que estudiaremos más adelante en la práctica de rendimiento.

Figura 4 Creación de una *Connection Factory*

Ejercicio 9: En la máquina virtual donde se encuentra el servidor de aplicaciones (10.X.Y.2), declare manualmente la factoría de conexiones empleando la consola de administración, tal y como se adjunta en la **Figura 4**.

8.2.2 Creación de la cola de mensajes

Para crear un *destination resource* de tipo "cola de mensajes" accederemos a la opción Resources -> JMS Resources -> Destination resources

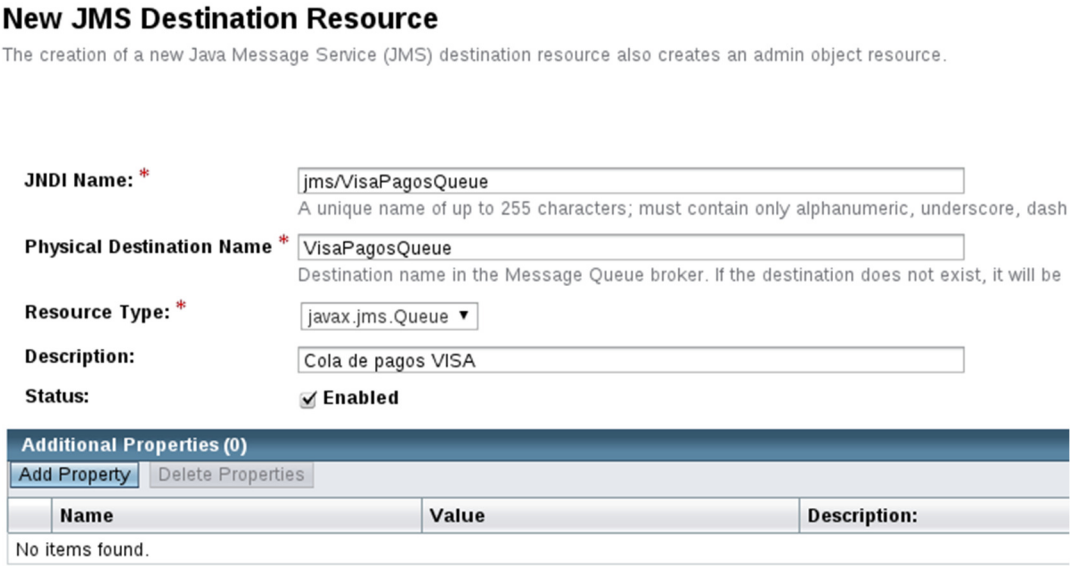

Figura 5 Creación de una *Cola de Mensajes*

Pulsando el botón "*New JMS Destination Resource*" accederemos a una pantalla como la de la **Figura 5**. Deberemos rellenar los siguientes campos:

- **JNDI name**: Estableceremos un nombre que, preferentemente, empiece por jms/nombreDeLaCola para facilitar su localización por JNDI
- **Physical destination resource:** Nombre del recurso (de la cola) físico. Estamos usando el gestor de colas incluido en glassfish, por lo que se puede emplear el mismo nombre que el del recurso JNDI (sin el jms/), pero en situaciones en que usemos un gestor de colas externo pudiera ser que la nomenclatura en los recursos físicos estuviese más restringida.

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

8.2.3 Envío de mensajes

Para codificar el envío de mensajes a través de una cola, es necesario incluir:

- Anotación para indicar la *factory* y la *cola*

  @Resource(mappedName = "jms/NombreDeLaConnectionFactory")
  @Resource(mappedName = "jms/NombreDeLaCola")

- **connectionFactory.createConnection()** : Empleado para obtener una conexión del pool.
- **connection.createSession()** : A partir de la conexión, se crea una sesión que permitirá conectar al destino.
- **session.createProducer()** : A partir de la sesión, se crea un "productor de mensajes" que es el que realmente envía los mensajes
- **session.createTextMessage()** : Crea el mensaje de texto que será enviado. Otros de los tipos de mensajes pueden ser creados son: *ObjectMessage* (creará mensajes adecuados para objetos que implementen la interfaz *serializable*)
- **messageProducer.send()** : Envía el mensaje

Consulte el apéndice 12.1 para un ejemplo completo de código de envío de mensajes

8.2.4 Recepción de mensajes

La recepción de mensajes es un proceso simétrico al anterior y hará uso de las mismas anotaciones y similares métodos para recibirlos. A diferencia del anterior, deberá ejecutarse:

- **session.createConsumer()** : A partir de la sesión, se crea un "consumidor de mensajes" que es el que realmente recibe los mensajes
- **messageConsumer.receive()** : Realiza la recepción del mensaje, de tipos similares a los ya descritos.

Consulte el apéndice 12.2 para un ejemplo completo de código de recepción de mensajes.

No obstante, en esta práctica, nuestro MDB recibe automáticamente los mensajes, invocando el servidor de aplicaciones el método OnMessage de nuestro MDB. Así pues, no es necesario el uso de los métodos anteriores.

8.3 VISA: Cancelación de un pago

En este apartado vamos a incluir la posibilidad de que un agente externo realice la cancelación de un pago mediante el envío de un mensaje. Lo que estamos es "simulando" que el titular de la tarjeta no reconozca el pago, operación que normalmente se hace a posteriori. Nuestra aplicación deberá recibir los mensajes de cancelación vía JMS, para modificar el registro correspondiente al pago a través de la página web. Para implementar este apartado:

- Desarrollaremos un MDB, VisaCancelacionJMSBean. Esta clase debe implementar un *destino de mensaje JMS* correspondiente a la cola creada en el apartado anterior.
- El objeto quedará a la espera de mensajes de cancelación, de tipo TextMessage La cancelación incluye un *identificador de la autorización* que habrá sido previamente convertido a texto para su envío. El receptor deberá traducirlo al tipo correspondiente y actualizar el registro correspondiente de la base de datos empleando una nueva consulta SQL (UPDATE) sobre la tabla de pagos. Deberá modificar el código de respuesta para que sea "999" (Cancelado por el usuario).

8.3.1 Material de partida

El alumno dispone del archivo P1-jms-base.tgz, que descomprimirá para conseguir la siguiente estructura de archivos:

P1-jms	
|-- build.xml / build.properties	Archivo ant para desplegar
|-- jms.xml /jms.properties	Fichero ant para automatizar la creación de los recursos JMS (cola y connection factory) del proyecto
|-- conf	
|-- mdb	Configuración servidor para MDB
|-- META-INF	
|-- sun-ejb-jar.xml	Descriptor de despliegue del EJB / MDB
|-- src	
|-- clientjms	Fuentes del cliente
|-- ssii2/VisaQueueMessageProducer.java	Completar según instrucciones que siguen
|-- mdb	Fuentes del servidor
|-- ssii2/visa/VisaCancelacionJMSBean.java	Completar según instrucciones que siguen

El proyecto proporcionado contiene dos paquetes:

- **clientjms**: Clase VisaQueueMessageProducer.java que permitirá la generación de mensajes contra la cola usando la línea de comandos.
- **mdb**: Clase VisaCancelacionJMSBean.java que permitirá la cancelación de mensajes vía mensajes JMS generados desde el programa anterior.

8.3.2 Preparación del MDB

8.3.2.1 sun-ejb-jar.xml

Los MDBs son EJBs, por lo que el fichero META-INF/sun-ejb-jar.xml también debe existir.

En éste deberemos indicar el nombre JNDI de la *connection factory* que se va a emplear:

```
<ejb>
  <ejb-name>VisaCancelacionJMSBean</ejb-name>
  <mdb-connection-factory>
    <jndi-name>NombreDeLaConnectionFactory</jndi-name>
  </mdb-connection-factory>
</ejb>
```

8.3.2.2 Código del MDB

El fichero VisaCancelacionJMSBean.java contiene un ejemplo del contenido esperado del MDB.

El método **onMessage()** debe implementar la lógica de qué debe hacer el objeto cuando llegue un mensaje. Este método se invocará de forma automática al colocarse un mensaje en la cola a la que esté suscrito el MDB. El método deberá obtener el mensaje y actuar en consecuencia según la función esperada.

Ejercicio 11:

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

8.3.3 Preparación del cliente

Existen principalmente dos maneras para que el cliente pueda localizar la cola de mensajes: mediante anotaciones estáticas o mediante la búsqueda del recurso JNDI

8.3.3.1 Recursos JMS estáticos

En el primer método, los miembros privados connectionFactory y queue deben estar anotados con los nombres de los recursos, por ejemplo, así:

```
@Resource(mappedName = "jms/NombreDeLaConnectionFactory")
private static ConnectionFactory connectionFactory;
@Resource(mappedName = "jms/NombreDeLaCola")
private static Queue queue;
```

Tiene el inconveniente de que el nombre hay que conocerlo en tiempo de compilación.

8.3.3.2 Recursos JMS dinámicos

En este segundo método, el programa puede obtener dinámicamente los recursos a través de una búsqueda explícita JNDI:

```
InitialContext jndi = new InitialContext();
connectionFactory = (ConnectionFactory)jndi.lookup("jms/NombreDeLaConnectionFactory");
queue = (Queue)jndi.lookup("jms/NombreDeLaCola");
```

Ejercicio 12: Implemente ambos métodos en el cliente proporcionado. Deje comentado el método de acceso por JNDI. Indique en la memoria de prácticas qué ventajas podrían tener uno u otro método.

8.3.4 Compilación

El proyecto proporcionado contiene un fichero build.xml que implementa la compilación y creación de recursos. Ambos proyectos (cliente y mdb) se compilan mediante la ejecución de "ant todo"

Ejercicio 13: Automatice la creación de los recursos JMS (cola y *connection factory*) en el build.xml y jms.xml. Para ello, indique en jms.properties los nombres de ambos y el Physical Destination Name de la cola de acuerdo a los valores asignados en los ejercicios 7 y 8. Recuerde también asignar las direcciones IP adecuadas a las variables ***as.host.mdb*** (build.properties) y ***as.host.server*** (jms.properties).

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

```
cd P1-jms
```

```
ant todo
```

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

8.3.5 Ejecución

El MDB, como hemos explicado anteriormente, será ejecutado ante el evento de llegada de un mensaje. Para probar el ciclo completo realizaremos a continuación:

- Un pago con la aplicación web
- Verificación de que el pago se ha realizado
- Cancelación con el cliente proporcionado

Ejercicio 14: Importante: Detenga la ejecución del MDB con la consola de administración para poder realizar satisfactoriamente el siguiente ejercicio (check de 'Enabled' en Applications/P1-jms-mdb y guardar los cambios).

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

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

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

(web console->Configurations->server-config->Java Message Service->JMS Hosts->default_JMS_host)

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

Verifique el contenido de la cola ejecutando:

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

Indique la salida del comando e inclúyala en la memoria de prácticas.

A continuación, volver a habilitar la ejecución del MDB y realizar los siguientes pasos:

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

Al realizar este ejercicio en los laboratorios surge un error indicando que no es posible resolver el nombre del host local a una dirección IP. Esto se debe a que no hay una entrada con dicho nombre en el fichero /etc/hosts asociado a una dirección IP. Como dicho fichero no se puede editar, la solución es ejecutar el cliente de colas de mensajes desde la máquina virtual 1, para que se conecte a la máquina virtual 2. Basta con copiar el .jar del cliente a la máquina virtual, iniciar sesión de forma remota. Indicar donde se encuentra la versión 8 de java exportando la variable JAVA_HOME y ejecutar el cliente de colas con appclient desde la máquina virtual 1. Para ello, hay que ejecutar la siguiente secuencia de comandos:

Desde PC1 host:

```
$ scp dist/clientjms/P1-jms-clientjms.jar si2@10.X.Y.1:/tmp
```

Desde la máquina virtual 10.X.Y.1:

```
si2@si2srv01:~$  export JAVA_HOME=/usr/lib/jvm/java-8-oracle/
```

```
si2@si2srv01:~$ /opt/glassfish4/glassfish/bin/appclient -targetserver 10.X.Y.2
-client /tmp/P1-jms-clientjms.jar <idAutorizacion>
```

9. Entrega

La fecha de entrega de esta práctica es la semana del 13 al 17 de Marzo de 2016, antes del comienzo de la clase de prácticas correspondiente.

La **entrega de los resultados de esta práctica se regirá por las normas generales expuestas durante la presentación de la asignatura**. El incumplimiento de estas normas conllevará a considerar que la práctica no ha sido entregada en tiempo. Esto implica que se tendrá que volver a enviar correctamente y que se aplicará la penalización por retraso pertinente.

A modo de resumen, para esta práctica se espera encontrar dentro del archivo SI2P1B_<grupo>_pareja>.zip (ejemplo: SI2P1B_2311_1.zip):

- Informe técnico
- P1-ejb con todas las modificaciones que hayan sido necesarias para el EJB.
- P1-ejb-servidor-remoto con todas las modificaciones que hayan sido necesarias para el servidor remoto de EJB.
- P1-ejb-cliente-remoto con todas las modificaciones que hayan sido necesarias para el cliente remoto de EJB.
- P1-ejb-transaccional con todas las modificaciones que hayan sido necesarias para el EJB.
- P1-jms con todas las modificaciones que hayan sido necesarias para el MDB.

10. Bibliografía recomendada

- *Transparencias sobre Javascript* (http://arantxa.ii.uam.es/%7Esi2lab/doc/Javascript_ssii2.ppt.pdf)
- *Transparencias sobre JSTL* (http://arantxa.ii.uam.es/%7Esi2lab/doc/JSTL_ssii2.ppt.pdf)
- *Transparencias sobre mecanismos de filtrado en servlets* (http://arantxa.ii.uam.es/%7Esi2lab/doc/*Servlets*_ssii2.ppt.pdf)
- *API Java EE 6 (http://download.oracle.com/javaee/7/api/)*
- *Java EE 6 Tutorial*: http://download.oracle.com/javaee/7/tutorial/doc/

Apéndices

11. APÉNDICE I: EJBs[1]

11.1 Introducción

Los Enterprise JavaBeans EJB son una de las API de J2EE (ahora JEE 7), y su especificación detalla cómo los servidores de aplicaciones proveen objetos (EJBs) desde el lado del servidor, así como los papeles jugados por el contenedor de EJB y los propios EJBs.

Los EJB proporcionan un modelo de componentes distribuido estándar del lado del servidor. El objetivo es dotar al programador de un modelo que le permita abstraerse de los problemas generales de una aplicación empresarial (concurrencia, transacciones, persistencia, seguridad, etc.) para centrarse en el desarrollo de la lógica de negocio en sí. El hecho de estar basado en componentes permite que éstos sean flexibles y sobre todo reutilizables.

No hay que confundir los *Enterprise JavaBeans* con los **JavaBeans**. Los JavaBeans también son un modelo de componentes creado para la construcción de aplicaciones, pero no pueden utilizarse en entornos de objetos distribuidos al no soportar nativamente la invocación remota (RMI).

11.2 Ventajas de los Enterprise Beans

Los *enterprise beans* simplifican el desarrollo de grandes aplicaciones distribuidas, por varios motivos:

- **Contenedor EJB – Desarrollador EJBs**: El contenedor EJB proporciona servicios a los *enterprise beans*, facilitando de este modo al desarrollador la implementación de la lógica de negocio. Entre estos servicios de los que el contenedor EJB es responsable encontramos, por ejemplo, la gestión de la transaccionalidad y la seguridad.
- **Cliente EJB – EJB**: Los EJBs **encapsulan la lógica de negocio** liberando de este modo a los clientes del EJB de dicha tarea. Los desarrolladores de la capa de cliente se pueden así enfocar en la presentación, sin tener que codificar funciones complejas de lógica de la aplicación o el acceso a datos. De este modo obtenemos clientes ligeros, aspecto fundamental hoy en día en dispositivos móviles (por poner un ejemplo).
- **Portabilidad de los EJBs**: El empaquetador de la aplicación puede construir aplicaciones simplemente a base de "conectar" beans preexistentes. Estas aplicaciones pueden ejecutarse en cualquier servidor Java EE siempre que use APIs estándar.

11.3 Cuándo utilizar Enterprise Beans

Se puede considerar el uso de *enterprise beans* si la aplicación tiene alguno de los siguientes requerimientos:

- **Escalabilidad**: para acomodar un número creciente de usuarios, una solución posible es distribuir los componentes en varios servidores. Los EJBs no solo pueden ejecutarse en máquinas distintas, sino que su localización es transparente al cliente.
- **Transaccionalidad**: Para asegurar la integridad de los datos, se puede requerir el uso de transacciones para el acceso concurrente a objetos compartidos. Los EJBs soportan transacciones, por lo que este acceso seguro queda garantizado.
- **Clientes ligeros**: Con pocas líneas de código, los clientes pueden localizar fácilmente enterprise beans.

[1] Extraído y compilado fundamentalmente de www.wikipedia.com y www.oracle.com.

11.4 Tipos de EJBs

Existen dos tipos de EJBs:

- **EJB de Sesión (Session EJBs):** gestionan el flujo de la información en el servidor. Generalmente sirven a los clientes como una fachada de los servicios proporcionados por otros componentes disponibles en el servidor. Puede haber dos tipos:
  - ***Con estado (stateful).*** En un bean de sesión con estado, las variables de instancia del bean almacenan datos específicos obtenidos durante la conexión con el cliente. Cada bean de sesión con estado, por tanto, almacena el estado conversacional de un cliente que interactúa con el bean. Este estado conversacional se modifica conforme el cliente va realizando llamadas a los métodos de negocio del bean. El estado conversacional no se guarda cuando el cliente termina la sesión.
  - ***Sin estado (stateless).*** Los beans de sesión sin estado son objetos distribuidos que carecen de estado asociado permitiendo por tanto que se los acceda concurrentemente. No se garantiza que los contenidos de las variables de instancia se conserven entre llamadas al método. Y un EJB sin estado puede implementar un Web Service (hay anotaciones que directamente lo permiten).
- **EJB dirigidos por mensajes (Message-driven EJBs):** son los únicos beans con funcionamiento asíncrono. Usando el *Java Messaging System* (JMS), se suscriben a un tema (*topic*) o a una cola (*queue*) y se activan al recibir un mensaje dirigido a dicho tema o cola. No requieren de su instanciación por parte del cliente.

Hasta hace poco existía otro tipo de EJB, los llamados **EJB de Entidad (Entity EJBs).** Su objetivo era encapsular los objetos del lado del servidor que almacena los datos. Los EJB de entidad presentan la característica fundamental de la persistencia, bien gestionada por el contenedor (CMP, *Container Managed Persistence,* el contenedor se encarga de almacenar y recuperar los datos del objeto de entidad mediante el mapeo o vinculación de las columnas de una tabla de la base de datos con los atributos del objeto), bien gestionada por el *bean* (BMP, *Bean Managed Persistence,* el propio objeto entidad se encarga, mediante una base de datos u otro mecanismo, de almacenar y recuperar los datos a los que se refiere, por lo cual, la responsabilidad de implementar los mecanismos de persistencia es del programador). A partir de la documentación de java para **JEE 5.0**, los *entity beans* desaparecen ya que son remplazados por **JPA** (***Java Persistence API***).

Los EJBs de sesión, tanto con estado como sin estado **pueden ser accedidos vía una interfaz Local** (para el acceso sobre la misma JVM) o **Remota** (sobre diferentes JVMs).

11.5 ¿Qué es un EJB de Sesión?

Un EJB de sesión representa a un cliente único en el interior del servidor de aplicaciones. Para acceder a una aplicación que se instala en el servidor, el cliente invoca los métodos del EJB de sesión. El EJB de sesión realiza un trabajo para su cliente, aislándole de la complejidad del servicio prestado, mediante la ejecución de las tareas de negocio encapsuladas dentro del servidor.

Como su nombre indica, un EJB de sesión es similar a una sesión interactiva. Un EJB de sesión no se comparte, sino que tiene un único cliente, de la misma manera que una sesión interactiva sólo puede tener un único usuario. Al igual que una sesión interactiva, un EJB de sesión no es persistente (es decir, sus datos no se guardan o persisten en una base o repositorio de datos). Cuando el cliente termina, su bean de sesión también termina, y no está asociado desde ese momento con el cliente.

11.6 ¿Cuándo usar EJBs de Sesión?

En general, se debe utilizar un EJB de sesión, si se dan las siguientes circunstancias:

- Cuando en cualquier momento dado, sólo un cliente tiene acceso a la instancia del *bean*.

- El estado del *bean* no es persistente, existente sólo por un corto período de tiempo (tal vez un par de horas).
- El *bean* implementa un servicio web.

Para mejorar el rendimiento, puede elegir un *bean* de sesión sin estado, si tiene alguno de estos rasgos:

- El estado del *bean* no tiene datos para un cliente específico.
- En una invocación de método único, el *bean* realiza una tarea genérica para todos los clientes. Por ejemplo, podría usar un bean de sesión sin estado para enviar un correo electrónico que confirma un pedido en línea.

11.7 Definiendo el acceso del cliente al EJB de Sesión mediante interfaces

Un cliente puede acceder a un EJB de sesión sólo a través de los métodos definidos en la interfaz de negocio del *bean*. La interfaz de negocio define la *vista* del cliente de un EJB. Todos los demás aspectos de detalle (implementaciones de método y de configuración de implementación) están ocultos al cliente.

El correcto diseño de los interfaces simplifica el desarrollo y mantenimiento de las aplicaciones Java EE. No sólo las interfaces correctas protegen a los clientes de cualquier complejidad en el nivel de EJB, sino que también permiten que se pueda cambiar la implementación detallada interna de los mismos sin afectar a los clientes. Por ejemplo, si se cambiara un EJB sin estado a ser un EJB de sesión con estado, no se tendrá que modificar el código de cliente. Pero si se tuviera que cambiar las definiciones de método en la interfaz, entonces casi con seguridad podría tener que modificar el código de cliente también. Por lo tanto, es importante que el diseño de las interfaces se haga con cuidado para aislar a sus clientes de posibles cambios en los *beans*.

Los *beans* de sesión pueden tener más de una interfaz de negocio. Así mismo suelen implementar la interfaz o interfaces de negocio, aunque no están obligados a ello.

Cuando se diseña una aplicación Java EE, una de las primeras decisiones a tomar es el tipo de acceso de cliente permitido por los *beans* de empresa: **remoto, local, o vía servicio web**.

11.7.1 Clientes Remotos

Un cliente remoto de un EJB tiene las siguientes características:

- Puede funcionar en un servidor y una máquina virtual Java (JVM) diferente a la del EJB que invoca.
- Puede ser un componente web, un cliente de aplicación, u otro EJB.
- Para un cliente remoto, la ubicación del EJB es transparente.

Para crear un *bean* de empresa que permita el acceso remoto, debe realizar las siguientes tareas:

- Comentar la interfaz del *bean* con la anotación **@Remote**:

```
@Remote
      public interface InterfaceName { ... }
```

- Comentar la clase del *bean* con la anotación @Remote, especificando la interfaz o interfaces de negocio:

```
@Remote(InterfaceName.class)
      public class BeanName implements InterfaceName { ... }
```

- La interfaz remota define los métodos y el ciclo de vida de negocio que son específicos del *bean*.

11.7.2 Clientes Locales

Un cliente local tiene las siguientes características:

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

La interfaz local de negocio define los métodos de negocio y el ciclo de vida del *bean*. Si la interfaz de negocio del *bean* no está comentada con las anotaciones @Local o @Remote, y la clase del *bean* no especifica la interfaz usando las anotaciones @Local o @Remote, la interfaz de negocio por defecto es interfaz local. Para construir un EJB que permita solo acceso local, se puede, aunque no es obligatorio, realizar una de las siguientes opciones:

- Comentar la interfaz de negocio del bean con @Local, por ejemplo:

```
@Local
      public interface InterfaceName { ... }
```

- Especificar la interfaz comentando la clase del bean con @Local y especificando el nombre de la interfaz, por ejemplo:

```
@Local(InterfaceName.class)
      public class BeanName implements InterfaceName { ... }
```

11.7.3 Decidiendo acceso local o remoto

El permitir acceso local o remoto a un *bean* depende de los siguientes factores.

- Grado de acoplamiento a los *bean* relacionados: Si dependen unos de otros fuertemente, estarán bastante acoplados y deberían accederse localmente entre ellos. Por ejemplo, si un *bean* de sesión que gestiona los procesos de pedidos de cliente llama a un *bean* de sesión que genera los correos electrónicos de confirmación al cliente, estos beans están estrechamente unidos. Los beans fuertemente acoplados son buenos candidatos para el acceso local. Debido a que encajan entre sí como una unidad lógica, por lo general se llaman entre sí con frecuencia y se beneficiarían de un mayor rendimiento con el acceso local.
- Tipo de cliente: Si el bean de empresa es invocado por clientes de aplicación, se debe permitir el acceso remoto. En un entorno de producción, estos clientes casi siempre se ejecutan en máquinas diferentes que el servidor de aplicaciones donde corren los EJBs. Si los clientes de un bean de empresa son componentes web u otros beans de empresa, el tipo de acceso depende de cómo se quiera distribuir sus componentes.
- La distribución de componentes: Las aplicaciones Java EE son escalables debido a que sus componentes del lado del servidor puede ser distribuidos a través de múltiples máquinas. En una aplicación distribuida, por ejemplo, los componentes web se puede ejecutar en un servidor diferente del de los EJBs que acceden. En este escenario distribuido, los beans de empresa deben permitir el acceso remoto.
- Rendimiento: Debido a factores como la latencia de la red, las llamadas remotas pueden ser más lentas que las llamadas locales. Por otro lado, si distribuye componentes entre los diferentes servidores, es posible mejorar el rendimiento general de la aplicación. Ambas afirmaciones son generalizaciones; el rendimiento real puede variar en diferentes entornos operativos. Sin embargo, se debe tener en cuenta cómo el diseño de una aplicación puede afectar a su rendimiento.

Si no se está seguro de qué tipo de acceso se debe dar a un *bean* de empresa, se ha de elegir el acceso remoto. Esta decisión da más flexibilidad, al poder distribuir en el futuro los componentes.

Aunque es poco frecuente, es posible que un EJB pueda permitir el acceso remoto y local. Si este es el caso, bien la interfaz de negocio del *bean* debe ser diseñada expresamente como una interfaz de negocio comentada con las anotaciones @Local o @Remote, bien la clase del bean debe designar explícitamente

las interfaces de negocio utilizando las anotaciones @Local y @Remote. La misma interfaz de negocio no puede ser local y remota al mismo tiempo.

11.7.4 Clientes de Servicios Web

Un cliente de servicios web puede acceder a una aplicación Java EE de dos maneras. En primer lugar, el cliente puede acceder a un servicio web creado con JAX-WS (tal y como hemos explicado anteriormente). En segundo lugar, puede invocar los métodos de negocio de un bean de sesión sin estado.

Siempre que se utilizan los protocolos correctos (SOAP, HTTP, WSDL), cualquier cliente de servicios web, puede acceder a un EJB de sesión sin estado, si el cliente está escrito en el lenguaje de programación Java. El cliente ni siquiera "sabe" la tecnología que implementa el servicio: EJB de sesión sin estado, JAX-WS, o alguna otra tecnología. Además, los *beans* de empresa y los componentes Web pueden ser clientes de servicios web. Esta flexibilidad le permite integrar las aplicaciones Java EE con los servicios web fácilmente.

Un cliente de servicios web accede a un *bean* de sesión sin estado a través de la clase de implementación del punto final (*endpoint*) del servicio web del *bean*. Por defecto, todos los métodos públicos en la clase del *bean* son accesibles para los clientes de servicios web. La anotación @WebMethod se puede utilizar para personalizar el comportamiento de los métodos de servicios web. Si la anotación @WebMethod se utiliza para comentar los métodos de la clase bean, sólo los métodos comentados con @WebMethod están expuestos a los clientes de servicios web.

12. Apéndice II: JMS

12.1 Envío de mensajes

El siguiente código Java ilustra cómo enviar mensajes a una cola.

```
@Resource(mappedName = "jms/NombreDeLaConnectionFactory")
  private static ConnectionFactory connectionFactory;
  @Resource(mappedName = "jms/NombreDeLaQueue")
  private static Queue queue;
  public void enviarMensaje()
  {
    MessageProducer messageProducer;
    TextMessage textMessage;
    try
    {
      Connection connection = connectionFactory.createConnection();
      Session session = connection.createSession(false,
        Session.AUTO_ACKNOWLEDGE);
      messageProducer = session.createProducer(queue);
      textMessage = session.createTextMessage();

      textMessage.setText("Esto es un mensaje en formato texto");
      System.out.println("Enviando el siguiente mensaje: "
        + textMessage.getText());
```

```
      messageProducer.send(textMessage);

      textMessage.setText("Adios!");
      System.out.println("Enviando el siguiente mensaje: "
        + textMessage.getText());
      messageProducer.send(textMessage);
      messageProducer.close();
      session.close();
      connection.close();
    }
    catch (JMSException e)
    {
      e.printStackTrace();
    }
}
```

12.2 Recepción de mensajes

El siguiente código Java ilustra cómo recibir mensajes a la cola.

```
connection = connectionFactory.createConnection();
Session session = connection.createSession(false,
        Session.AUTO_ACKNOWLEDGE);
messageConsumer = session.createConsumer(queue);
connection.start();
while (!goodByeReceived) {
        System.out.println("Esperando mensajes...");
        textMessage = (TextMessage) messageConsumer.receive();
        if (textMessage != null)
        {
          System.out.print("Recibido el mensaje: ");
          System.out.println(textMessage.getText());
          System.out.println();
        }
        if (textMessage.getText() != null
          && textMessage.getText().equals("Adios!")) {
          goodByeReceived = true;
        }
      }
      messageConsumer.close();
      session.close();
      connection.close();
```