Ciclo Formativo DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Módulo 6 Acceso a Datos

Unidad Formativa 4

INICIATIVA Y COORDINACIÓN
IFP Innovación en Formación Profesional

Supervisión editorial y metodológica:
Departamento de Producto de Planeta Formación

Supervisión técnica y pedagógica:
Departamento de Enseñanza de **IFP** Innovación en Formación Profesional

Módulo: Acceso a Datos
UF 4 / Desarrollo de Aplicaciones Multiplataforma

MÓDULO 6

Unidad Formativa 4

Acceso a Datos

Esquema de contenido

OBJETIVOS

- Valorar las ventajas e inconvenientes de utilizar conectores.
- Establecer conexiones, modificaciones y consultas sobre una base de datos usando conectores.
- Gestionar transacciones mediante conectores.
- Conocer los conceptos de componente (propiedades y atributos); eventos (asociación de acciones a eventos); e introspección (reflexión).
- Realizar componentes de acceso a datos.

INTRODUCCIÓN

Se llama conector al conjunto de clases encargadas de implementar la interfaz de programación de aplicaciones (API) y facilitar, con ello, el acceso a una base de datos. Para poder conectarse a una base de datos y lanzar consultas, una aplicación siempre necesita tener un conector asociado.

Desde los lenguajes propios de los sistemas gestores de bases de datos se pueden gestionar los datos mediante lenguajes de consulta y manipulación propios de esos sistemas. Sin embargo, cuando se quiere acceder a los datos desde lenguajes de programación de una misma manera con independencia del sistema gestor que contenga los datos, entonces es necesario utilizar conectores que faciliten estas operaciones. Los conectores dan al programador una manera homogénea de acceder a cualquier sistema gestor (preferiblemente relacional u objeto-relacional).

En este capítulo se hace una primera introducción a los conceptos básicos que hay detrás de los conectores. Seguidamente se muestran ejemplos sobre cómo realizar las operaciones básicas sobre una base de datos MySQL usando conectores con Java.

1. EL DESFASE OBJETO-RELACIONAL

El problema del desfase objeto-relacional consiste en la diferencia de aspectos que existen entre la programación orientada a objetos, con la que se desarrollan aplicaciones, y la base de datos, con las que se almacena la información. Estos aspectos se pueden presentar relacionados cuando:

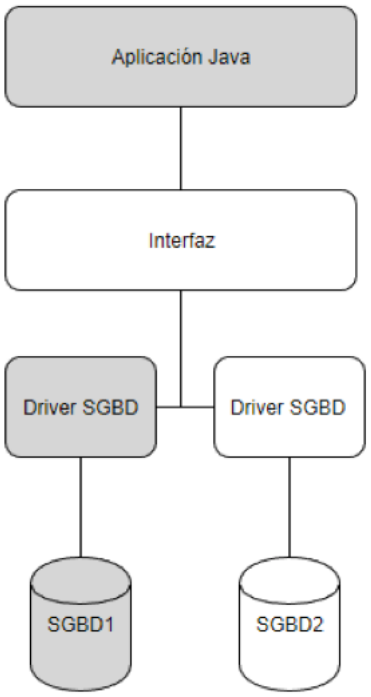
- Se realizan actividades de programación, donde el programador debe conocer el lenguaje de programación orientada a objetos (POO) y el lenguaje de acceso a datos.
- Se especifican los tipos de datos. En las bases de datos relacionales siempre hay restricciones en cuanto a los tipos de datos que se pueden usar, suelen ser tipos simples, mientras que en la programación orientada a objetos se utilizan tipos de datos complejos.
- En el proceso de elaboración del software se realiza una traducción del modelo orientado a objetos al modelo entidad-relación (ER) puesto que el primero maneja objetos y el segundo maneja tablas y tuplas o filas, lo que implica que el desarrollador tenga que diseñar dos diagramas diferentes para el diseño de la aplicación.

La discrepancia objeto-relacional surge porque en el modelo relacional se trata con relaciones y conjuntos debido a su naturaleza matemática. Sin embargo, en el modelo de programación orientada a objetos se trabaja con objetos y las asociaciones entre ellos. Al problema se le denomina desfase objeto-relacional, o sea, el conjunto de dificultades técnicas que aparecen cuando una base de datos relacional se usa conjuntamente con un programa escrito con lenguajes de programación orientada a objetos.

En el Capítulo 3 se volverá a hacer hincapié sobre esta idea, destacando los problemas del desfase objeto-relacional en programación y mostrando la solución que son los sistemas gestores orientados a objetos (SGBDOO) para solventar este problema.

2. PROTOCOLOS DE ACCESO A BASES DE DATOS: CONECTORES

Muchos servidores de bases de datos utilizan protocolos de comunicación específicos que facilitan el acceso a los mismos, lo que obliga a aprender un lenguaje nuevo para trabajar con cada uno de ellos. Es posible reducir esa diversidad de protocolos mediante alguna interfaz de alto nivel que ofrezca al programador una serie de métodos para acceder a la base de datos.



Estas interfaces de alto nivel ofrecen facilidades para:

- Establecer una conexión a una base de datos.
- Ejecutar consultas sobre una base de datos.
- Procesar los resultados de las consultas realizadas.

Las tecnologías disponibles, aunque muy diversas, abstraen la complejidad subyacente de cada producto y proporcionan una interfaz común, basada en el lenguaje de consulta estructurado (SQL), para el acceso homogéneo a los datos. Algunos ejemplos representativos son JDBC (Java Data Base Connectivity) de Sun y ODBC (Open Data Base Connectivity) de Microsoft.

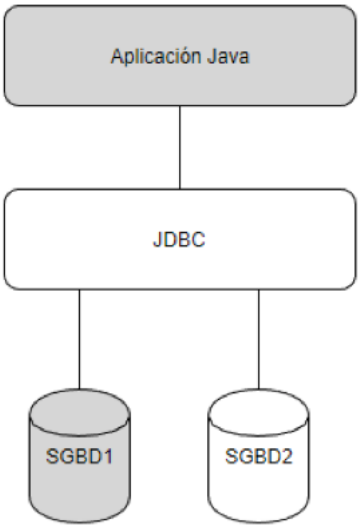
Al conjunto de clases encargadas de implementar la interfaz de programación de aplicaciones (API) y facilitar, con ello, el acceso a una base de datos se le denomina conector o driver. Para poder conectarse a una base de datos y lanzar consultas, una aplicación siempre necesita tener un conector asociado.

Cuando se construye una aplicación de base de datos, el conector oculta los detalles específicos de cada base de datos, de modo que el programador solo debe preocuparse de los aspectos relacionados con su aplicación, olvidándose de otras consideraciones. La mayoría de los fabricantes ofrecen conectores para acceder a sus bases de datos.

Un ejemplo de conector muy extendido es el mencionado con anterioridad, el conector JDBC. Este conector es una capa software intermedia situada entre los programas Java y los sistemas de gestión de bases

de datos relacionales que utilizan SQL. Dicha capa es independiente de la plataforma y del gestor de bases de datos utilizado.

Con el conector JDBC no hay que escribir un programa para acceder, por ejemplo, a una base de datos Access y otro programa distinto para acceder a una base de datos Oracle, etc., sino que se puede escribir un único programa utilizando el API JDBC, y es ese programa el que se encarga de enviar las consultas a la base de datos utilizada en cada caso.



Otro ejemplo de conectores el conector de Microsoft ODBC. La diferencia entre JDBC y ODBC está en que ODBC tiene una interfaz C. En este sentido, ODBC es simplemente otra opción respecto a JDBC, ya que la mayoría de sistemas gestores de bases de datos disponen de drivers para trabajar con ODBC y JDBC. Además, también existe en Java un driver JDBC-ODBC, para convertir llamadas JDBC a ODBC y poder acceder a bases de datos que ya tienen un driver ODBC y todavía no tienen un conector JDBC.

Seguidamente, y para ser coherentes con el resto de contenidos de este libro, el capítulo se centrará en el conector JDBC, sus componentes y principales características.

2.1. COMPONENETES JDBC

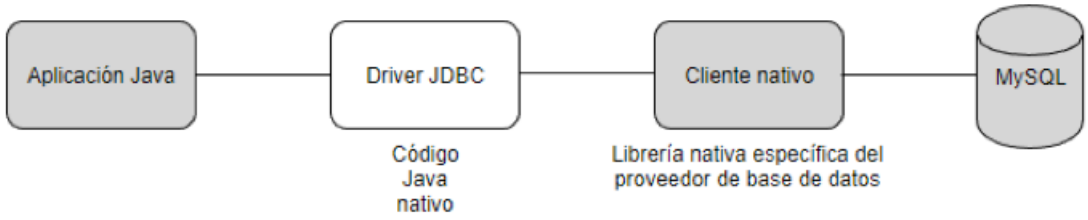
El conector JDBC incluye cuatro componentes principales:

- La propia APIJDBC, que facilita el acceso desde el lenguaje de programación Java a bases de datos relacionales y permite que se puedan ejecutar sentencias de consulta en la base de datos. Dicha API está disponible en los paquetes java.sql y javax.sql, Java Standard Edition (Java SE) / Java. Enterprise Edition (Java EE) respectivamente.
- El gestor del conector JDBC (drivermanager), que conecta una aplicación Java con el driver correcto de JDBC. Se puede realizar por conexión directa (DriverManager) o a través de un pool de conexiones, vía DataSource.
- La suite de pruebas JDBC, encargada de comprobar si un conector (driver cumple con los requisitos JDBC.
- El driver o puente JDBC-ODBC, que permite que se puedan utilizar los drivers ODBC como si fueran de tipo JDBC.

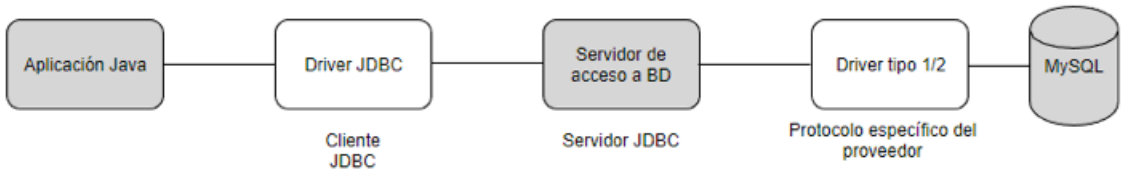
2.2. TIPOS DE CONECTORES

En función de los componentes anteriores, en un conector JDBC existen cuatro tipos de controladores JDBC. La denominación de estos controladores está asociada a un número de 1 a 4 y viene determinada por el grado de independencia respecto de la plataforma, prestaciones, etc. Seguidamente se muestran cada uno de estos tipos de conectores JDBC.

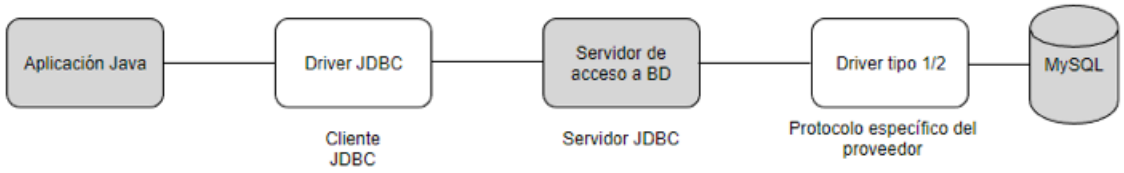
Driver tipo 1: utilizan una API nativa estándar, donde se traducen las llamadas de JDBC a invocaciones ODBC a través de librerías ODBC del sistema operativo.



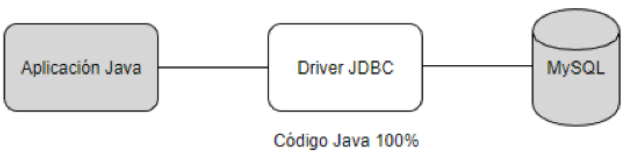
Driver tipo 2: utilizan una API nativa de la base de datos, es decir son drivers escritos parte en Java y parte en código nativo. El driver usa una librería cliente nativa, específica de la base de datos con la que se desea conectar No es un driver 100% Java. La aplicación Java hace una llamada a la base de datos a través del driver JDBC y este traduce la petición a invocaciones a la API del fabricante de la base de datos.



Driver tipo 3: utilizan un servidor remoto con una API genérica, es decir son drivers que usan un cliente Java puro que se comunica con un middleware server usando un protocolo independiente de la base de datos (por ejemplo, TCP/IP). Este tipo de drivers convierte las llamadas en un protocolo que puede utilizarse para interactuar con la base de datos.



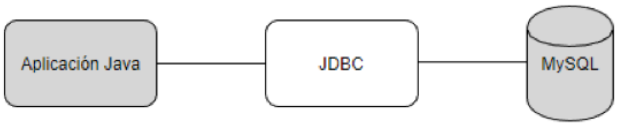
Driver tipo 4: es el método más eficiente de acceso a base de datos. Este tipo de drivers son suministrados por el fabricante de la base de datos y su finalidad es convertir llamadas JDBC en un protocolo de red comprendido por la base de datos. Este tipo de driver es el que se trabajará en los ejemplos incluidos en este capítulo



2.3. MODELOS DE ACCESO A BASE DE DATOS

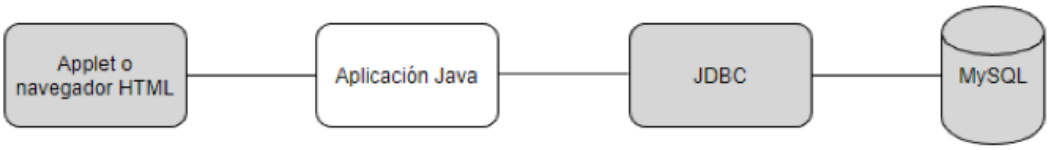
A la hora de establecer el canal de comunicación entre una aplicación Java y una base de datos se pueden identificar dos modelos distintos de acceso. Estos modelos dependen del número de capas que se contemple.

En el modelo de dos capas, la aplicación que accede a la base de datos reside en el mismo lugar que el driver de la base de datos. Sin embargo, la base de datos puede estar en otra máquina distinta, con lo que el cliente se comunica por red. Esta es la configuración llamada cliente-servidor, y en ella toda la comunicación a través de la red con la base de datos será manejada por el conector de forma transparente a la aplicación Java.



Alternativamente al modelo de dos capas, en el modelo de tres capas los comandos se envían a la capa intermedia de servicios, que envía las consultas a la base de datos. Esta las procesa y envía los resultados de vuelta a la capa intermedia, para que más tarde sean enviados al cliente. En este modelo una aplicación o applet de Java se está ejecutando en una máquina y accediendo a un driver de base de datos situado en otra máquina. Ejemplos de puesta en práctica de este modelo de acceso se dan en los siguientes casos:

- Cuando se tiene un applet accediendo al driver a través de un servidor web.
- Cuando una aplicación accede a un servidor remoto que comunica localmente con el driver.
- Cuando una aplicación, qué está en comunicación con un servidor de aplicaciones, accede a la base de datos por nosotros.



2.4. ACCESO A BASE DE DATOS MEDIANTE UN CONECTOR JDBC

Las dos ventajas que ofrece JDBC pasan por proveer una interfaz para acceder a distintos motores de base de datos y por definir una arquitectura estándar con la que los fabricantes puedan crear conectores que permitan a las aplicaciones Java acceder a los datos. Este apartado se centra en la primera de esas ventajas.

A continuación, se muestra cómo acceder a una base de datos utilizando JDBC. Lo primero que se debe hacer para poder realizar consultas en una base de datos es, obviamente, instalar la base de datos. Dada la cantidad de productos de este tipo que hay en el mercado, es imposible explicar la instalación de todas ellas, así que se optará por una en concreto y La elegida es una base de datos MySQL. Se ha elegido este gestor de bases de datos porque es gratuito y por funcionar en diferentes plataformas.

Antes de cargar el driver JDBC y obtener una conexión con la base de datos se deben hacer dos pasos:

Lo primero que se necesita para conectar a una base de datos es un objeto conector. Ese conector es el que sabe cómo interactuar con la base de datos. El lenguaje Java no viene con todos los conectores de todas las posibles bases de datos del mercado. Por tanto, se debe recurrir a Internet para obtener el conector que se necesite en cada caso.

En los ejemplos de este capítulo, se necesitará el conector de MySQL. Una vez descargado el fichero mysql. connectorjava-5.1.xx.zip, se descomprime y se localiza el fichero mysql-connectorjava-5.1.21-binjar (donde xx hace referencia a la versión más actual del mismo), que viene incluido en el fichero zip. En ese otro archivo un fichero con extensión jar ofrece la clase conector que nos interesa.

Para incluir el fichero mysql-connectorjava-5.1.21-binjar en cada proyecto se deberán seguir los siguientes pasos:

1. En la carpeta raíz del proyecto, crear la carpeta / lib.
2. Copiar el fichero mysql-connectorjava-5.1.21-binjar en la carpeta / lib que se acaba de crear.
3. Desde NetBeans IDE 7.1.2. pulsar con el botón derecho del ratón en el nombre del proyecto y seleccionar la opción de menú propiedades (Properties).
4. En el árbol lateral pulsar en Libraries.
5. En el botón de la izquierda pulsar en Add JARVFolder.
6. Seleccionar el fichero mysql-connector-java-5.1.21-binjar que se encuentra en la carpeta / lib del proyecto y pulsar Abrir.
7. Pulsar OK. La Figura 2.11 muestra cómo quedarían las propiedades del proyecto con la librería mysqlconnectorjava-5.1.21-binjar incorporada en tiempo de compilación.

La segunda acción para poder utilizar el conector sin problemas es localizarlo en el sistema operativo identificando su ruta en la variable de entorno CLASSPATH, siempre que nuestro IDE (Eclipse, Netbeans, etc.) utilice esa variable. Desde consola el comando para lograr este propósito sería el siguiente:

$ set CLASSPATH=<PATH_DEL TAR> \mysql-connector-iava-5.1.21-bin.Jar

Una vez localizado el driver (conector) en el sistema será posible cargarlo desde cualquier aplicación Java (Figura 2.10, pasos 1 y 2).

Un ejemplo es el siguiente código. Lo que hace es acceder a una base de datos llamada discográfica que se ha creado en MySQL. Esta base de datos tiene una tabla ALBUMES. Se han creado en ALBUMES tres campos: ID, clave primaria tipo numérico, TITULO, tipo VARCHAR (30), y AUTOR, tipo VARCHAR (30).

Ejemplo

```
public Gestor_conexion() {
//Constructor
// crea una conexión
Connection conn1 = null;
try {
String url1 = “jdbc: mysql://localhost:33.06/discografica”;
```

```
String user = "root"; String password = "";
//no tiene clave
conn1 = DriverManager. getConnection (url1, user, password);
if (conn1 != null) {
System.out.println ("Conectado a discográfica...") ;
} catch (SQLException ex) {
System.out.println ("ERROR: dirección no válida o usuario/clave");
ex.printstackTrace () :
```

El procedimiento de conexión con el controlador de la base de datos, independientemente de la arquitectura, es siempre muy similar. En primer lugar se carga el conector. Cualquier driver JDBC, independientemente de su tipo, debe implementar la interfaz java.sql Driver. La carga del driver se realizaba originalmente con Class.forName(driver). Sin embargo, al poner el jar del driver (MySQL en nuestro caso) en la carpeta lib del proyecto (o en el classpath del programa), cuando la clase DriverManager se inicializa, busca esta propiedad en el sistema y detecta que se necesita el driver elegido.

Una vez cargado el driver, el programador puede crear una conexión (paso 2 en la Figura 2.10). El objetivo es conseguir un objeto del tipo java.sql. Connection a través del método DriverManager getConnection (String url). En el código anterior se muestra el uso de este método.

La línea url 1 = "jdbc: mysql://localhost:3306/discografica"; indica que se desea acceder a una base de datos MySQL mediante JDBC, que la base de datos está localizable en el localhost (127.0.0.1) por el puerto 3306 y que su nombre es discográfico (sin tilde).

Si todo va bien, cuando se ejecute la sentencia donde el objeto DriverManager invoca al método get Connection () se crea una conexión a una base de datos MySQL. Si esa invocación fuera mal, se informará de una excepción gracias al uso de las sentencias try-catch utilizadas (captura de excepciones).

No hay que olvidar que, después de usar una conexión, ésta debe ser cerrada con el método close() de Connection. El siguiente código muestra un ejemplo de cómo hacerlo.

Ejemplo

```
public void cerrar Conexion (Connection conn1) {
 try {
Connll. Close ();
} catch (SQLException ex) {
System. out.println ("ERROR: al cerrar la conexión");
ex.print StackTrace ();
```

2.4.1. Pool de conexiones

La manera mostrada de obtener una conexión está bien para aplicaciones sencillas, donde únicamente se establece una conexión con la base de datos. Sin embargo, hay un pequeño problema con esta alternativa: varios hilos de ejecución no pueden usar una misma conexión física con la base de datos simultánea-

mente, ya que la información enviada o recibida por cada uno de los hilos de ejecución se entremezcla con la de los otros, haciendo imposible una escritura o lectura coherente en dicha conexión. Hay varias posibles soluciones para este problema:

Abrir y cerrar una conexión cada vez que la necesitemos. De esta forma, cada hilo de ejecución tendrá la suya propia. Esta solución en principio no es eficiente, puesto que establecer una conexión real con la base de datos es un proceso costoso. El hecho de andar abriendo y cerrando conexiones con frecuencia puede hacer que el programa vaya más lento de lo debido.

Usar una única conexión y sincronizar el acceso a ella desde los distintos hilos. Esta solución es más o menos eficiente, pero requiere cierta disciplina al programar, ya que es necesario poner siempre synchronized antes de hacer cualquier transacción con la base de datos. También tiene la pega de que los hilos deben esperar entre ellos.

Finalmente, también existe la posibilidad de tener varias conexiones abiertas (pool de conexiones), de forma que cuando un hilo necesite una, la pida, y cuando termine, la deje para que pueda ser usada por los demás hilos, todo ello sin abrir y cerrar la conexión cada vez. De esta forma, si hay conexiones disponibles, un hilo no tiene que esperar a que otro acabe. Esta solución es en principio la ideal y es la que se conoce como pool de conexiones.

Apostando por el tercero de los escenarios anteriores, en Java, un pool de conexiones es una clase que tiene abiertas varias conexiones a bases de datos. Cuando alguien necesita una conexión a base de datos, en vez de abrirla directamente con DriverManagerget Connection.0, se pide al pool usando su método pool.get Connection (). El pool coge una de las conexiones que ya tiene abierta, la marca para saber que está asignada y la devuelve. La siguiente llamada a este método pool.get Connection buscará una conexión libre para marcarla como ocupada y ofrecerla.

2.5. CLASES BÁSICAS DEL API JDBC

En la sección anterior se ha mostrado cómo hacer una conexión con una base de datos MySQL. Como se ha comentado anteriormente, la interfaz del conector JDBC reside en los paquetes java.sql y javax.sql. Lo que se ofrece en esos paquetes son en su mayoría interfaces, ya que la implementación específica de cada una de ellas es fijada por cada proveedor según su protocolo de bases de datos. En cualquier caso, en la interfaz hay distintos tipos de objetos que se deben tener presentes, por ejemplo, Connection, Statement y ResultSet. El resto de objetos necesarios se mostrarán en próximas secciones.

Los objetos de la clase Connection() ofrecen un enlace activo a una base de datos a través del cual un programa en Java puede leer y escribir datos, así como explorar la estructura de la base de datos y sus capacidades. Se crea con una llamada a DriverManager getConnection () o a DataSourceget Connection (en JDBC 2.0). En la sección anterior se han mostrado ejemplos asociados donde se utilizan ambas llamadas.

La interfaz DriverManager complementaria de la clase Connection. Con ella se registran los controladores JDBC y se proporcionan las conexiones que permiten manejar las URL específicas de JDBC. Se consigue con el método getConnection() de la propia clase.

La clase Statement proporciona los métodos para que las sentencias, utilizando el lenguaje de consulta estructurado (SQL), sean ejecutadas sobre la base de datos y se pueda recuperar el resultado de su ejecución. Hay tres tipos de sentencias Statement cada una especializa a la anterior. Estas sentencias se verán en las siguientes secciones.

Además de las clases anteriores, el API JDBC ofrece también la posibilidad de gestionar excepciones con la clase SQLException. Dicha clase es la base de las excepciones de JDBC. La mayor parte de las operaciones que proporciona el APIJDBC lanzarán la excepción java.sql.SQLException en caso de que se produzca algún error en la base de datos (por ejemplo, errores en la conexión, sentencias SQL incorrectas, falta de privilegios, etc.). Por este motivo es necesario dar un tratamiento adecuado a estas excepciones y encerrar todo el código JDBC entre bloques try/catch.

2.6. CLASES ADICIONALES DEL API JDBC

Además de las clases básicas anteriores, el API JDBC también ofrece la posibilidad de acceder a los metadatos de una base de datos. Con ellos se puede obtener información sobre la estructura de la base de datos y, gracias a ello, se pueden desarrollar aplicaciones independientemente del esquema que tenga la base de datos. Las principales clases asociadas a metadatos de una base de datos son las siguientes: Database MetaData y ResultSet MetaData.

Los objetos de la clase Database MetaData ofrecen la posibilidad de operar con la estructura y capacidades de la base de datos. Se instancian con connection.getMetaData(). Los metadatos son datos acerca de los datos, es decir, datos que explican la naturaleza de otros datos. La interfaz Database Meta Data contiene más de 150 métodos para recuperar información de una base de datos (catálogos, esquemas, tablas, tipos de tablas, columnas de las tablas, procedimientos almacenados, vistas etc.), así como información sobre algunas características del controlador JDBC que se esté utilizando. Estos métodos son útiles cuando se implementan aplicaciones genéricas que pueden acceder a diversas bases de datos.

Los objetos de la clase ResultSet MetaData son el ResultSet que se devuelve al hacer un execute Query0 de un objeto Database MetaData. Los métodos de Result Set Meta Data permiten determinar las características de un objeto ResultSet. Por ejemplo, con un objeto de la clase ResultSet MetaData se puede determinar el número de columnas; información sobre una columna, tal como el tipo de datos o la longitud; la precisión y la posibilidad de contener nulos, e información sobre si una columna es de solo lectura, etc.

3. EJECUCIÓN DE SENTENCIAS DE DEFINICIÓN DE DATOS

El lenguaje de definición de datos (Data Definition Language o Data Description Language (DDL según autores) es la parte de SQL dedicada a la definición de una base de datos. Dicho lenguaje consta de sentencias para definir

la estructura de la base de datos y permite definir gran parte del nivel interno de la misma. Por este motivo, estas sentencias serán utilizadas normalmente por el administrador de la base de datos.

Las principales sentencias asociadas con el lenguaje DDL son CREATE, ALTER y DROP Siempre se usan estas sentencias junto con el tipo de objeto y el nombre del objeto. Dichas sentencias permiten:

- CREATE sirve para crear una base de datos o un objeto.
- ALTER sirve para modificar la estructura de una base de datos o de un objeto.
- DROP permite eliminar una base de datos o un objeto.

Para enviar comandos SQL a la base de datos con JDBC se usa un objeto de la clase Statement. Este objeto se obtiene a partir de una conexión a base de datos, de esta forma:

Statement St = conexion.createStatement () ;

Statement tiene muchos métodos, pero hay dos especialmente interesantes: executeUpdate() y executequery0.

- execute Update 0: se usa para sentencias SQL que impliquen modificaciones en la base de datos (INSERT, UPDATE, DELETE, etc.).
- execute Query0: se usa para consultas (SELECT y similares).

El siguiente ejemplo muestra cómo se modifica una tabla desde una aplicación Java:

Ejemplo

```
// Añadir una nueva columna a una tabla ya existente
Statement sta = con.createStatement();
int count = sta.executeUpdate ("ALTER TABLE contacto ADD edad");
```

Por último, el siguiente código elimina la tabla llamada contacto de una base de datos previamente abierta:

```
st.executeUpdate(DROP TABLE contacto");
```

4. EJECUCIÓN DE SENTENCIAS DE MANIPULACIÓN DE DATOS

Las sentencias de manipulación de datos (Data Manipulation Language (DMIL)) son las utilizadas para insertar, borrar, modificar y consultar los datos que hay en una base de datos. Las sentencias DML son las siguientes:

La sentencia SELECT sirve para recuperar información de una base de datos y permite la selección de una o más filas y columnas de una o muchas tablas.

La sentencia INSERT se utiliza para agregar registros a una tabla.

La sentencia UPDATE permite modificar la información de las tablas.

La sentencia DELETE, permite eliminar una o más filas de una tabla.

El siguiente ejemplo muestra un método para insertar valores en una tabla álbum

Ejemplo

```
public void Insertar () ( try {
// Crea un statement
Statement sta = conn1 - create Statement();
// Ejecuta la inserción
sta.executeUpdate ('INSERT INTO album ' + 'VALUEs (3, 'Black Album", Metallica").");
// Cierra el statement
sta. Close ();
} catch (SQLException ex) {
System. out. println ("ERROR: al hacer un Insert");
ex. print StackTrace () :
```

En el ejemplo, partiendo de una conexión previa (conn. 1) se crea un objeto Statement llamado sta. Sobre ese objeto se ejecuta una consulta Insert into. SQLException, que se encarga de capturar los errores que se cometan en la sentencia. Por último, al terminar de usar el Statement, este debe cerrarse close() para evitar errores inesperados.

5. EJECUCIÓN DE CONSULTAS

La ejecución de consultas sobre bases de datos desde aplicaciones Java descansa en dos tipos de clases disponibles en el APIJDBC y en dos métodos. Las clases son Statement y ResultSet, y los métodos, executequery y execute Update.

5.1. CASLE STATEMENT

Como se ha comentado en varias ocasiones, las sentencias Statement son las encargadas de ejecutar las sentencias SQL estáticas con Connection.createStatement0.

El método execute Query0 de Statement está diseñado para sentencias que devuelven un único resultado (ResultSet), o como es el caso de las sentencias SELECT

Resultset res = sta. executeQuery () :

Los objetos de la clase ResultSet son los utilizados para representar la respuesta a las peticiones que se hacen a una base de datos. Esta clase no es más que un conjunto ordenado de filas de una tabla. Asociados a la clase ResultSet existen métodos como mext:0 y getXXX) para iterar por las filas y obtener los valores de los campos deseados. Después de invocar al método next0, el resultado recién traído está disponible en el ResulSet. La forma de recoger los campos es pedirlos con algún método getXXXO. Si se sabe de qué tipo es el dato, se puede pedir conget Int0, get String (), etc. Si no se sabe o da igual el tipo (como en el ejemplo), bastará con un get Object.0, que es capaz de traer cualquier tipo de dato.

El siguiente código muestra un ejemplo de acceso a la base de datos con una consulta SELECT que obtiene todos los álbumes cuyo título empieza por "B"

Ejemplo

```
public void Consulta Statement () {
 try {
Statement stmt = conn1. CreateStatement();
String query = "SELECT * FROM album WHERE titulo like `B%´";
ResultSet rs = stmt.executeQuery (query);
while (rs.next()) {
System.out.println("IID — "" + rs.getInt ( "id") + ", Título " + rs.getString ("titulo") + ", Autor "' +
rs.getString ("autor")) ;
}
rs.close();
 stmt.close () ;
} catch (SOLException ex) {
System.out.println("ERROR: al hacer un Insert");
ex.print.StackTrace();
```

El código ejecuta la consulta deseada con executeQuery() y el resultado lo devuelve en un ResultSet (llamado rs). El método next() de Result Set permite recorrer todas las filas para ir sacando cada uno de los valores devueltos. Como

el atributo id es de tipo Int se usa un método get Int0 para recuperarlo. Sin embargo, como título y autor es de tipo VARCHAR() se usa un getString().

Por último, al terminar de usar Result Set y Statement, estos deben cerrarse, close() para evitar errores inesperados.

5.2. CLASE PREPAREDSTATEMENT

Una primera variante de la sentencia Statement es la sentencia Prepared Statement. Se utiliza para ejecutar las sentencias SQL precompiladas. Permite que los parámetros de entrada sean establecidos de forma dinámica, ganando eficiencia.

El siguiente ejemplo muestra la ejecución de la consulta de la sección anterior, pero parametrizando el criterio de búsqueda. La diferencia principal con respecto a los Statementes que las consultas pueden tener valores indefinidos que se establecen con el símbolo interrogación (?). En las consultas se ponen tantas interrogaciones como parámetros se quieran usar. En el siguiente ejemplo solo hay un parámetro.

Ejemplo

```
public void Consulta_preparedStatement () {
try {
String query = "SELECT * FROM album WHERE titulo like ?";
Prepared Statement pst = conn1.prepareStatement (query);
pst.setString(1, "BÈ" ) ;
ResultSet rs = pst.executeQuery();
while (rs.next ()) {
System.out.println("IID — "" + rs.getInt ("id") + ", Título " + rs.getstring ("titulo") +
, Autor "" + rs.getString ("autor")) ;
rs.Close ();
pst. Close();
} catch (SQLException ex) {
System.out.println("ERROR: al consultar" );
ex.print.stackTrace();
```

Al crear el prepareStatement se precompila la consulta para dejarla preparada para recibir los valores de los parámetros. Si el parámetro que se quiere colocar es de tipo Int se usa set Int0. Sin embargo, en el ejemplo anterior se quiere dar un valor de texto por lo que se usa set String(I, "B%"). El primer valor (1) indica que la "B%" se asocia con la primera interrogación que se encuentra. Si hubiese más parámetros (?) entonces habría que indicar la posición que ocupa para que el prepareStatement sepa a cuál asociarle el valor (set String(2"), set String(3,"), etc.)

5.3. CLASE CALLABLESTATEMENT

Otro tipo de sentencias son las asociadas a los objetos de la clase CallableStatement, que son sentencias prepared Statement que llaman a un procedimiento almacenado, es decir, métodos incluidos en la propia base de datos. No todos los gestores de bases de datos admiten este tipo de procedimientos. La manera de proceder es la misma que la mostrada en el ejemplo prepareStatement aunque lo que se le da como parámetro es el nombre del procedimiento almacenado junto con los valores de los parámetros del procedimiento puestos como parámetros del Statement.

En el siguiente código se muestra un ejemplo de llamada para un supuesto procedimiento almacenado llamado Dame Albumes(titulo, autor), que devuelve todos los álbumes cuyo título y autor coincida con los valores dados.

Ejemplo

```
Callable Statement cs = conn1.prepareCall (“CALL. Dame Albumes (?, ?)”);
// Se proporcionan valores de entrada al procedimiento
cs.setString(1, “Blacks”);
cs.setString(2, Metallica”);
```

Las transacciones consisten en la ejecución de bloques de consultas manteniendo las propiedades ACID (AtomicyConsistency-Isolation-Durability), es decir, permiten garantizar integridad ante fallos y concurrencia de transacciones.

Después de que una transacción finaliza, la siguiente sentencia ejecutada automáticamente inicia la siguiente transacción. Una sentencia DDL o DCL es automáticamente completada y por consiguiente implícitamente finaliza una transacción.

Una transacción que termina con éxito se puede confirmar con una sentencia COMMIT, en caso contrario puede abortarse utilizando la sentencia ROLLBACK. En JDBC por omisión cada sentencia SQL se confirma tan pronto se ejecuta, es decir, una conexión funciona por defecto en modo auto-commit. Para ejecutar varias sentencias en una misma transacción es preciso deshabilitar el modo auto-commit, después se podrán ejecutar las instrucciones, y terminar con un COMMIT si todo va bien o un ROLLBACKen otro caso.

El siguiente código muestra el uso de transacciones con el ejemplo de insertar valores en la tabla álbum.

Ejemplo

```
public void Insertar con commit () {
try {
conn1.setAutoCommit(false);
Statement sta = connl. create Statement();
sta.executeUpdate(‘INSERT INTO album ‘ + ‘VALUES (5, “Black Album”, “Metallica’ ) ”);
sta.executeUpdate(“INSERT INTO album “ + “VALUES (6, “A kind of magic’, ‘Queen’)”);
conn1 . comumit () ;
```

```
}catch(SQLException ex)
{
System. out.printlin (“ERROR: al hacer un Insert”); try if (conn1 != null) conn1.rollback () ;
} Catch (SQLException se2){
sa2.printStackTrace ();
}
}
//end try ex. printstackTrace();
```

Como se puede ver en el ejemplo, si las dos operaciones Insert into se ejecutan correctamente, entonces se aplica un commit antes de salir. Sin embargo, si no es así, y salta una excepción, hay que hacer un rollback0. El rollback0 es obligado ponerlo dentro de un try/catch.

6. CONCEPTO DE COMPONENTE: CARACTERÍSTICAS

Un componente es, en general, una unidad de software que encapsula partes de código con una funcionalidad determinada. Los componentes pueden ser visuales del estilo a los proporcionados por los entornos de desarrollo para ser incluidos en interfaces de usuario, o no visuales, los cuales tienen funcionalidad como si fueran librerías remotas. Un componente software tiene las siguientes características principales:

- Un componente es una unidad ejecutable que puede ser instalada y utilizada independientemente.
- Puede interactuar y operar con otros componentes desarrollados por terceras personas, es decir, una compañía o un desarrollador puede usar un componente y agregarlo a lo que esté haciendo; o sea, los componentes se pueden componer.
- No tiene estado, al menos su estado no es externamente visible.
- Por último, un componente y la programación orientada a componentes puede, metafóricamente, asociarse a los componentes electrónicos y al uso que se hace de ellos para conformar un sistema mayor.

Ejemplos de modelos de componentes serían los ensamblados de Microsoft.NET, que son una agrupación lógica de uno o más módulos o ficheros de recursos (ficheros.GIF, HTML, etc.) que se engloban bajo un nombre común; los Enterprise JavaBeans de J2EE o el modelo de componentes CORBA (Common Object Request Broker Architecture). Este capítulo se centrará principalmente en los Entreprise JavaBeans (EJB) que son una solución muy extendida en el desarrollo Java para la Web. De hecho, los EJB forman parte del estándar de construcción de aplicaciones empresariales J2EE.

Un EJB encapsula una parte de la lógica de negocio de una aplicación, y puede accedera gestores de recursos como bases de datos y otros EJB. Desde otro punto de vista, un EJB puede ser accedido por otros EJB, servlets, servicios web, y aplicaciones cliente. Los EJB residen en contenedores que dan soporte a distintos ámbitos (seguridad, transacción, despliegue, concurrencia y gestión de su ciclo de vida).

En este capítulo se recurrirá a los EJBs para ejemplificar las principales propiedades y características de un componente software y su uso en el contexto del acceso a datos.

Un componente software (entre ellos un EJB) puede quedar caracterizado de la siguiente forma: Atributos, Operaciones + Eventos) + Comportamiento + (Protocolos + Escenarios) + Propiedades. Así, los Atributos, las Operaciones y los Eventos son parte de la interfaz de un componente, y representan su nivel sintáctico. El Comportamiento de estos operadores representa la parte semántica del componente. Los Protocolos determinan la interoperabilidad del componente con otros, es decir, la compatibilidad de las secuencias de los mensajes con otros componentes y el tipo de comportamiento que va a tener el componente en distintos Escenarios donde puede ejecutarse. Finalmente, el término Propiedades se refiere a las características extrafuncionales que puede tener un componente (dentro de ellas se incluye la seguridad, la fiabilidad o la eficiencia, entre otras).

7. PROPIEDADES

Las propiedades de un componente determinan su estado y lo diferencian del resto. Antes se ha comentado que los componentes carecen de estado, sin embargo, los componentes disponen de una serie de propiedades a las que se puede acceder y modificar de diferentes formas. Las propiedades de un componente se dividen en simples, indexadas, compartidas y restringidas.

Las propiedades de un componente se pueden examinar y modificar mediante métodos o funciones de acceso que acceden a ellas. Estas funciones son de dos tipos:

El método get, que sirve para consultar o leer el valor de una propiedad. Su sintaxis general es la siguiente: Tipodela Propiedadget.Nombre Propiedad0;.

El método set, que sirve para asignar o cambiar valor de una propiedad. Su sintaxis general en Java es setNombrePropiedad (Tipodela Propiedad valor);

También es posible establecer, modificar y consultar los valores de las propiedades de un componente mediante la sección propiedades que acompaña a muchos de los entornos integrados de desarrollo (IDE). Un ejemplo de este tipo de entornos es Netbeans 7 (cuyas versiones 7.1.2 y 7.2.1 se han usado en varios ejemplos de capítulos anteriores). Cuando este tipo de entornos carga un componente utiliza mecanismos de reflexión para cargar los valores de sus propiedades. En las próximas secciones se desarrolla el concepto de reflexión.

7.1. SIMPLES E INDEXADAS

Las propiedades simples son aquellas que representan un único valor. Por ejemplo, suponiendo un botón de una interfaz gráfica, las propiedades simples serían aquellas relacionadas con su tamaño, su color de fondo, su etiqueta, etc.

Además de las propiedades de valores únicos y simples, existe otro tipo de propiedades más complejas y muy similares a un conjunto de valores: indexadas. Los elementos de este tipo de propiedades comparten todos ellos el mismo tipo y a ellos se accede mediante un índice. Se puede acceder a ellas mediante los métodos de acceso mencionados antes (get/set), aunque la invocación a estos métodos cambiará un poco, ya que cada propiedad es solo accesible a través de su índice. Su sintaxis general en Java es set-Nombredela Propiedad (intíndice, Tipo de la Propiedad valor).

7.2. COMPARTIDAS Y RESTRINGIDAS

Además de las propiedades simples e indexadas, existen otros dos tipos de propiedades que tiene un componente: las propiedades compartidas y las restringidas.

Las propiedades compartidas son aquellas que cuando cambian notifican a todas las partes interesadas en esa propiedad, y solo a ellas, la naturaleza del cambio. El mecanismo de notificación está basado en eventos, es decir, existe un componente fuente que mediante un evento notifica a un componente receptor cuándose produce un cambio en la propiedad compartida. Debe quedar claro que estas propiedades no son bidireccionales y que, por tanto, los componentes receptores no pueden contestar.

Para que el componente dé soporte a propiedades compartidas, debe soportar dos métodos usados para registrar los componentes interesados en el cambio de propiedad (uno para adición y otro para eliminación). Su sintaxis general en Java es addPropertyChangeListener(PropertyChangeListenerx) y removePrope rtyChangeListener(PropertyChange Listener v).

Por otro lado, las propiedades restringidas buscan la aprobación de otros componentes antes de cambiar su valor. Como con las propiedades compartidas se deben proporcionar dos métodos de registro para los receptores. Su sintaxis general en Java es add Vetoable Change ListenerVetoable Changelistener x) y removeVetoable ChangelistenerVetoabl eChange Listener x).

8. ATRIBUTOS

Los atributos son uno de los aspectos relevantes de la interfaz de un componente ya que son los elementos que forman parte de la vista externa del componente (los representados como públicos). Estos elementos observables son particularmente importantes desde el punto de vista del control y del uso del componente, y son la base para el resto de los aspectos que caracterizan a un componente.

9. EVENTOS: ASOSCIACIÓN DE ACCIONES A EVENTOS

Una interfaz contiene solo información sintáctica de las signaturas de las operaciones entrantes y salientes de un componente, con las cuales otros componentes interactúan con él. A este tipo de interacción se le conoce con el nombre de control proactivo, es decir las operaciones de un componente son activadas mediante las llamadas de otro.

Además del control proactivo (la forma más común con la que se llama a una operación) existe otra forma que se denomina control reactivo y que se refiere a los eventos de un componente, como el que permite por ejemplo el modelo de componentes EJB. En el control reactivo, un componente puede generar eventos que se corresponden con una petición de llamada a operaciones; más tarde otros componentes del sistema recogen estas peticiones y se activa la llamada a una operación destinada a su tratamiento. Un símil de este funcionamiento sería por ejemplo cuando un icono de escritorio cambia de forma al pasar por encima el cursor del ratón. En este caso, el componente icono está emitiendo un evento asociado a una operación que cambia la forma del icono.

10. INTROSPECCIÓN. REFLEXIÓN

Las herramientas de desarrollo descubren las características de un componente (esto es, sus propiedades, sus métodos y sus eventos) mediante un proceso conocido como introspección. Introspección se puede definir como un mecanismo mediante el cual se pueden descubrir las propiedades, métodos y eventos que un componente contiene. Los componentes soportan la introspección de dos formas:

Usando las convenciones específicas de nombres conocidas cuando se nombran las características del componente. Para el caso de EJB, la clase Introspector examina el EJB buscando esos patrones de diseño para descubrir sus características.

Proporcionando explícitamente información sobre la propiedad, el método o el evento con una clase relacionada.

En particular, los EJB admiten la introspección a múltiples niveles. En el nivel bajo, esta introspección se puede conseguir por medio de las posibilidades de reflexión. Estas posibilidades permiten que los objetos Java descubran información acerca de los métodos públicos, campos y constructores de clases que se han cargado durante la ejecución del programa. La reflexión permite que la introspección se cumpla en todos los componentes de software, y todo lo que tiene que hacer es declarar un método o variable como public para que se pueda descubrir por medio de la reflexión.

11. PERSISTENCIA DEL COMPONENTE

En capítulos anteriores se ha dejado claro que el concepto de persistencia es hacer que los objetos de una aplicación existan más allá de la ejecución de la misma, guardándolos en bases de datos.

En el caso de EJB la persistencia está facilitada con la librería Java Persistence API (JPA). JPA es una especificación de Sun Microsystems para la persistencia de objetos Java a cualquier base de datos relacional. Esta API fue desarrollada para la plataforma J2EE e incluida en el estándar desde la versión EJB3.0. Para relacionarlo con Hibernate visto en el Capítulo 4, JPA es una API, es solo una interfaz, que puede ser implementada con Hibernate o con otro ORM.

Para su utilización, JPArequiere de J2SE 1.5(también conocida como Java 5) o superior, ya que hace uso intensivo de las nuevas características del lenguaje Java, como las anotaciones y los genéricos. Para utilizar la librería JPA se deben tener conocimientos de programación orientada a objetos, de Java y del lenguaje de consulta estructurado (SQL).

- Persistence. La clase javax, persistence. Persistence contiene métodos estáticos de ayuda para obtener una instancia de Entity Manager Factory de una forma independiente al vendedor de la implementación de JPA.
- Entity Manager Factory. La clase javax, persistence. Entity Manager Factory ayuda a crear objetos de Entity Manager utilizando el patrón de diseño del Factory.
- Entity Manager. La clase javax, persistence. Entity Manager es la interfaz principal de JPA utilizada para la persistencia de las aplicaciones. Cada Entity Manager puede realizar operaciones de creación, lectura, modificación y borrado sobre un conjunto de objetos persistentes.
- Entity. La clase.javax, persistence. Entity es una anotación Java que se coloca a nivel de clases Java serializables y en la que cada objeto de una de estas clases anotadas representa un registro de una base de datos.
- Entity Transaction. Cada instancia de Entity Manager tiene una relación de uno a uno con una instancia de javax, persistence. Entity Transaction. Permite operaciones sobre datos persistentes de manera que agrupados formen una unidad de transacción, en el que todo el grupo sincroniza su estado de persistencia en la base de datos o todos fallan en el intento. En caso de fallo, la base de datos quedará con su estado original.
- Query. La interfaz.javax, persistence. Query está implementada por cada vendedor de JPA para encontrar objetos persistentes manejando cierto criterio de búsqueda.JPA estandariza el soporte para consultas utilizando Java Persistence Query Language (JPQL) y Structured Query Language (SQL). Se puede obtener una instancia de Query desde una instancia de un Entity Manager.

Por último, las anotaciones JPA, conocidas también como anotaciones EJB3.0, se encuentran en el paquete javax.persistence.* Muchos IDE que soportan a JDK5 como Eclipse, Netbeans, IntelliJ IDEA, poseen herramientas y plugins para generar clases de entidad con anotaciones de JPA a partir de un esquema de base de datos.

12. HERRAMIENTAS PARA DESARROLLO DE COMPONENTES NO VISUALES

Muchos de los lenguajes que dicen ser orientados a componentes, son realmente lenguajes de configuración o entornos de desarrollo visuales, como era el caso de Visual Basic, Delphi, C++ Builder o JBuilder. Estos entornos permitían crear y ensamblar componentes, pero separaban totalmente los entornos de desarrollo y de utilización de componentes.

Actualmente, los entornos de desarrollo de componentes más utilizados son NetBeans; Eclipse para el desarrollo de componentes en Java (por ejemplo, EJB); y Microsoft.NET para el desarrollo de ensamblados (COM+, DCOM). Estos entornos ofrecen alternativas de generación automática de código que facilitan enormemente el desarrollo de aplicaciones que usen componentes.

13. EMPAQUETADO DE COMPONENTES

Al hablar en general existen muchas alternativas para empaquetar componentes. Sin embargo, centrados en EJB, una aplicación J2EE se distribuye en un archivo empresarial (EAR) que es un archivo Java estándar (JAR) con una extensión ear. El uso de archivos EAR y módulos hace posible ensamblar una gran cantidad de aplicaciones Java EE utilizando alguno de los mismos componentes. No se necesita codificación extra; es solo un tema de empaquetado de varios módulos J2EE en un fichero.ear de J2EE.

Un módulo J2EE consta de uno o más componentes J2EE para el mismo tipo de contenedor y un descriptor de despliegue de componente para ese tipo. Un descriptor de despliegue de módulo EJB especifica los atributos de una transacción y las autorizaciones de seguridad para un EJB. Un módulo JavaEE sin un descriptor de despliegue de aplicación puede ser desplegado como un módulo independiente.

Antes de EJB3.1 todos los EJB tenían que estar empaquetados en estos archivos. Como una buena parte de todas las aplicaciones J2EE contienen un front-end web y un back-end con EJB, esto significa que debe crearse un lear que contenga a la aplicación con dos módulos: un luar y un ejb-jar. Esto es una buena práctica en el sentido de que se crea una separación estructural clara entre el front-end y el back-end. Pero para las aplicaciones simples resulta demasiado complicada.

Los cuatro tipos de módulos de J2EE para aplicaciones web con EJB son los siguientes:

- Módulos EJB, que contienen los ficheros con clases para EJB y un descriptor de despliegue EJB. Los módulos EJB son empaquetados como ficheros JAR con una extensión jar.
- Los módulos web, que contienen ficheros conservlets, ficheros JSP, fichero de soporte de clases, ficheros GIF y HTML y un descriptor de despliegue de aplicación web. Los módulos web son empaquetados como ficheros JAR con una extensión.uvar (ueb archive).
- Los módulos de aplicaciones de cliente que contienen los ficheros con las clases y un descriptor de aplicación de cliente. Los módulos de aplicación clientes son empaquetados como ficheros JAR con una extensión jar.

14. TIPOS DE EJB

Concretando en EJB, existen tres tipos de EJB:

- EJB de entidad (Entity EJBs): su objetivo es encapsular los objetos de lado de servidor que almacenan los datos. Los EJB de entidad presentan la característica fundamental de la persistencia:

Persistencia gestionada por el contenedor (CMP): el contenedor se encarga de almacenar y recuperar los datos del objeto de entidad mediante un mapeado en una tabla de una base de datos.

Persistencia gestionada por el EJB (BMP): 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.

- EJB de sesión (Session. EJBs): gestionan el flujo de la información en el servidor Generalmente sirven a los clientes como una fachada (fagade) de los servicios proporcionados por otros componentes disponibles en el servidor. Puede haber dos tipos:

Con estado (Stateful). Los EJB de sesión con estado son objetos distribuidos que poseen un estado. El estado no es persistente, pero el acceso al EJB se limita a un solo cliente.

Sin estado (Stateless). Los EJB de sesión sin estado son objetos distribuidos que carecen de estado asociado permitiendo por tanto que se acceda a ellos concurrentemente. No se garantiza que los contenidos de las variables de instancia se conserven entre llamadas al método.

- EJB dirigidos por mensajes (Message-driven EJBs): los únicos EJB con funcionamiento asíncrono. Usando el Java Messaging System (JMS), se suscriben a un tópico (topic) o a una cola (queue) y se activan al recibir un mensaje dirigido a dicho tópico o cola. No requieren de su instanciación por parte del cliente.

Los EJB se disponen en un contenedor EJB dentro del servidor de aplicaciones. La especificación describe cómo er"JB interactúa con su contenedor y cómo el código cliente interactúa con la combinación del EJB y el contenedor Cada EJB debe facilitar una clase de implementación Java y dos interfaces Java. El contenedor EJB creará instancias de la clase de implementación Java para facilitar la implementación EJB. Las interfaces Java son utilizados por el código cliente del EJB.

Las dos interfaces, conocidas como interfaz local e interfaz remota, especifican las signaturas de los métodos remotos del EJB. Los métodos remotos se dividen en dos grupos:

- Métodos que no están ligados a una instancia específica, por ejemplo aquellos utilizados para crear una instancia EJB o para encontrar una entidad EJB existente. Estos métodos se declaran en la interfaz local.
- Métodos ligados a una instancia específica. Se ubican en la interfaz remota. Dado que se trata simplemente de interfaces Java y no de clases concretas, el contenedor EJB es necesario para generar clases para esas interfaces que actuarán como un proxy en el cliente. El cliente invoca un método en los proxies generados que a su vez sitúa los argumentos en un mensaje y envía dicho mensaje al servidor EJB.

15. EJEMPLO DE EJB CON NETBEANS

Vista toda la teoría anterior, en esta sección se muestra un ejemplo de creación de un componente que permite el acceso a datos almacenados en una base de datos. En concreto el componente se explica con los siguientes puntos:

- Los datos están almacenados en una base de datos JavalDB dentro del entorno NetBeans 7.2.1.
- Se desarrolla un EJB usando las posibilidades de NetBeans 7.2.1 con J2EE. Este componente hará una entidad. De hecho, la solución que se utilizará es muy parecida al mapeo con Hibernate.
- Se creará un servlet que usará las funciones del EJB creado.
- Se creará una página JSP que invocará al servlet.
- Para que la aplicación funcione como aplicación web que es, será necesario un servidor de aplicaciones. En este caso se usará Glass Fish dentro del entorno de NetBeans 7.2.1.

Antes de explicar los pasos a seguir es necesario crear el proyecto Java y una conexión a una base de datos.

Crear una base de datos en el propio entorno de IDE NetBeans (Java DB). Para el ejemplo que se detalla se ha creado una base de datos llamada "discografía" que tiene una tabla ALBUMES para el esquema ROOT Se han creado en ALBUMES tres campos: ID, clave primaria tipo numérico;TITULO, tipo VARCHAR(30); y AUTOR, tipo VARCHAR(30). La Figura 4.2 muestra en NetBeans la estructura creada.

15.1. Creación del proyecto

El proyecto que usará un EJB debe ser un proyecto web. Para crearlo con NetBeans 7.2.1 con J2EEse deben seguir los siguientes pasos: (1) ir al menú Fichero (File)->Nuevo proyecto. Dentro de la carpeta veb se encuentra un tipo de proyecto llamado Web Application. (2) Se pulsa en Siguiente. (3) El nombre del proyecto es EJBAcceso. (4). Se marca la opción Use Dedicated Folderfor Storing Libraries y se deja por defecto la ruta.\lib. (5) Se pulsa en Siguiente. (6) Se selecciona GlassFish como servidor (server). (7). Se marca la opción Enable Contexts and Dependency Injection (8) Una vez seleccionada se le da a Terminar.

Creado el proyecto, NetBeans creará una estructura de carpetas y todo lo necesario para ejecutar aplicaciones web depurables sobre GlassFish, incluido el despliegue de la aplicación. La Figura 6.3 muestra la estructura de carpetas creada.

15.2. Añadiendo soporte para JPA

Como se ha comentado en las secciones anteriores, JPA permite la persistencia de objetos Java en sistemas relacionales. JPA es una API, es solo una interfaz, que puede ser implementada con Hibernate o con otro ORM.

Con JPA se añadirá una nueva entidad de persistencia al proyecto, la cual se creará desde la tabla ALBUMES de la base de datos Discografía creada antes.

Para añadir una clase de persistencia en NetBeans 7.2.1 con J2EE se deben seguir los siguientes pasos:

1. seleccionando la raíz del proyecto (EJBAcceso) se pulsa en el botón derecho del ratón y, del menú contextual que aparece, se selecciona Nuevo (New) y, se selecciona dentro de las posibilidades (otros...), un tipo archivo Entity Classes from Database en la carpeta Persistencia.

2. Se pulsa en Siguiente.
3. De la lista desplegable se selecciona la conexión a la base de datos creada antes y aparecerá en la lista de la izquierda la tabla que contiene.
4. Se selecciona la tabla que se quiere mapear (llevándola a la otra lista con añadir Add). Para este ejemplo es ALBUMES, que fue creada con anterioridad en la base de datos.
5. Sepulsa en Siguiente.
6. En la ventana (Entity Classes) solo se añade entidades como nombre del paquete (package) donde se guardará la entidad creada.
7. Una vez añadido se le da a Terminar.

15.3. Creando un EJB de sesión sin estado (stateless)

En este ejemplo se creará un EJB de sesión stateless, es decir, cuando un cliente invoca un método del EJB, el EJB está listo para ser reusado por otro cliente, y la información almacenada en el EJB es desechada cuando el cliente deja de acceder al EJB. En la Sección 6.9 se han mostrado los tipos de EJB.

Los pasos para crear un EJB stateless son:

1. seleccionando la raíz del proyecto (EJBAcceso) se pulsa en el botón derecho del ratón y, del menú contextual que aparece, se selecciona Nuevo (New) y, se selecciona dentro de las posibilidades (otros...), un tipo archivo Session Bean en la carpeta Enterprise JavaBean.
2. Sepulsa en Siguiente.
3. Se le pone por nombre a la sesión Albumes EJB.
4. Se le llama beans al paquete que contendrá el fichero.
5. Se marca como tipo de sesión (Session Type), Stateless. Del resto (Create Interface) no se selecciona nada.
6. Se pulsa en Terminar.

Hechos estos pasos, se habrá creado en la estructura de carpetas del proyecto un nuevo paquete llamado beans. Dentro de ese paquete habrá un archivo:Albumes EJB.java. Hay que abrir ese archivo, y en el cuerpo de la clase, añadir como lógica de negocio el siguiente código:

Ejemplo

```
@PersistenceUnit
EntityManagerFactory emf;
public List findAll() {
return emf.createEntityManager().createNamed
Query (“Albumes. find All”).getResultList () ;
}
```

Además, se deben añadir los siguientes import:

Ejemplo

```
import java. util. List;
import javaix.persistence.Entity ManagerFactory;
import javaix.persistence.Persistence Unit;
import javax.eib.Stateless;
```

Este es el código que le da significado al método Find All O del EJB, el cual devolverá una lista java. util. List) con los objetos Albumes que devuelva una consulta SQL sobre la base de datos ALBUMES. Este método será llamado desde un servlet que se añadirá en el siguiente paso.

15.4. Creando un servlet

Los pasos para crear un servlet son los siguientes:

1. seleccionando el raíz del proyecto (EJBAcceso) se pulsa en el botón derecho del ratón y, del menú contextual que aparece, se selecciona Nuevo (New) y se selecciona dentro de las posibilidades (otros...), un tipo archivo Servlet en la carpeta Web.
2. Se pulsa en Siguiente.
3. Se le pone por nombre al servlet ServletEJB.
4. Se le llama servlets al paquete que contendrá el fichero.
5. Se pulsa en Siguiente.
6. Sepulsa en Terminar.

Hecho esto, un nuevo paquete servlets y un fichero ServletEJB.java es añadido al proyecto. Hay que abrir el fichero Servlet EJB y añadir el siguiente código:

Antes del método process Request hay que añadir:

Ejemplo

```
@EJB
Albumes EJB aEJB;
Después, tenemos que sustituir todo el contenido del try por el siguiente código:
List<Albumes> i = aEJB.findAll();
out.println("<html>");
out.println("<head>");
out.println("<title>Servlet Albumservlet- title>");
out.println("</head>");
```

```
out.println("<body>");
for (int i = 0; i < 10; i++)
out. printlin ("«b>Titulo: </b>"+ 1.get(i).getTitulo() +", <b>Autor </b>"+1.get(i).getAutor ()
+"<br>" ) ;
out.println("</body >");
out.println("</html>");
```

Por último, se deben añadir las importaciones necesarias:

```
import java.util.List;
import beans.Albumes EJB;
import entidades.Albumes;
```

Al ejecutar el proyecto aparecerá una página web en el navegador por defecto con un botón enviar que al presionarlo llamará a ServletEJB que a su vez invocará al método findAll0 de Albumes EJB, que es el EJB creado.

Actividad

4.1. Utiliza r el contenido de la sección para configurar el entorno de desarrollo (IDE NEtBeans si es posible) para incluir las librerías JDBC que dan acceso a MySQL

4.2. Crear una base de datos de ejemplo para una discográfica: una tabla con canción (título (Varchar()), duración (Varchar()), letra (Varchar()) y albúm (id (Int), título (Varchar()), año en el que se publicó (Varchar()).

La relación entre las tablas álbum y canción es uno a muchos: un álbum está compuesto por muchas canciones.

4.3. Utilizar las tablas creadas en la actividad 4.2 para crear una aplicación Java que conecte con la base de datos.

4.4. Utilizar las tablas creadas en la Actividad 4.2 para crear una aplicación Java que permita modificar la tabla álbum para incluir un nuevo campo que contenga las imágenes de las carátulas de cada álbum.

4.5. Utilizar las tablas creadas en la Actividad 4.2 para crear una aplicación Java que permita consultas SELECT las tablas álbum y canciones. Las consultas deben ser parametrizadas (Statement) y no parametrizadas (preparedStatement). El resultado debe mostrarse en forma de lista de resultados.

4.6. Utilizar las tablas creadas en la Actividad 4.2 para crear una aplicación Java que permita realizar varias inserciones de datos en las tablas canciones y álbum de manera atómica. Si falla la inserción en una de las tablas entonces todo el proceso se debe anular.

4.7. Repetir los pasos de la sección 4.10 para crear una aplicación Java que acceda a una base de datos creada con JavaDB dentro del propio entorno de NetBeans. La base de datos debe tener dos tablas: Álbum y Canción. La tabla Álbum puede ser como la mostrada anteriormente. La tabla Canciones representa a las canciones de un álbum y rendrá los siguientes atributos (id del álbum, id de la sección [clave], t´tiulo de la canción y duración)

Índice