Patrón DAO / VO

Implementación de la capa de Modelo de nuestra aplicación

En un patrón de desarrollo de aplicaciones web MVC (modelo, vista, controlador), la capa de modelo (datos y persistencia) puede implementarse de distintas maneras. Aquí os proponemos un patrón de tipo DAO / VO simplificado, en el que además separamos la conectividad a la base de datos de los objetos DAO (en los modelos DAO originales, la conectividad a base de datos la gestionan también los objetos DAO).

Value Objects. VO

Los Value Objets (VO) son implementaciones en memoria (clases y objetos) de las entidades de nuestro modelo de aplicación. Normalmente se corresponderán total o casi totalmente con las tablas de entidades principales que tengamos en nuestra base de datos. La idea es que nuestra aplicación se comunique con la base de datos utilizando los VO como objetos de transferencia. Los objetos DAO serán los mensajeros que trasladarán esos VO entre la capa de lógica de la aplicación y el motor de persistencia (la base de datos). Los VO suelen ser, por tanto clases básicas de java (POJO o "plain old java object"), con algún pequeño añadido.

¿Qué debe tener un VO?
Los elementos fundamentales de un VO son:
- **Propiedades** que representen los campos o atributos que tenemos en la tabla de la base de datos que estamos modelando, con los tipos de datos correspondientes. Un VARCHAR de base de datos, será un String en nuestro VO, un INT será un int o un Integer, etc.
- **Métodos set / get** para acceder a esas propiedades
- (Opcional, pero útil) **Constructor** capaz de generarnos un VO a partir de todas sus propiedades.

Ejemplo:
Imaginemos que nuestra aplicación necesita una entidad llamada "demo", cuya representación en base de datos es la siguiente tabla:

```
CREATE TABLE `demo` (
  `id` int(11) NOT NULL,
  `name` varchar(100) DEFAULT NULL
)
```

Nuestro VO será así:

```
package es.unizar.sisinf.data.vo;


/**
 * Clase VO que implementa el objeto DEMO
 * @author
 *
 */
public class DemoVO {

	/**
	 * propiedad id: identificador único de la entidad
	 */
	private int id;

	/**
	 * propiedad name: Nombre
	 */
	private String name;
```

```
        /**
         * constructor que nos permite crear objetos DEMO a partir de sus componentes
         * @param id
         * @param name
         */
        public DemoVO(int id, String name) {
                this.id = id;
                this.name = name;
        }


        /*
         * Métodos set / get de las propiedades
         */

        public int getId() {
                return id;
        }
        public void setId(int id) {
                this.id = id;
        }
        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
}
```

Data Access Objects. DAO

Como ya hemos indicado, los DAO son los mensajeros, que transfieren objetos VO entre la capa de lógica de la aplicación, y la capa de persistencia. Para ello, estas clases implementarán todas las funciones (API) que sean necesarias para “conceptualizar” las necesidades de nuestra aplicación, tanto en lectura como en escritura de datos. Los objetos DAO abstraen y encapsulan la comunicación con la base de datos, de manera que la capa de lógica de la aplicación no necesite saber a qué base de datos se está conectando, ni si es una base de datos o cualquier otro sistema (por ejemplo, un API basada en servicios web). Simplemente, invoca unas funciones en la capa DAO, y recibe un resultado.

Un ejemplo simple de cómo implementaríamos una clase DAO para gestionar nuestros objetos DemoVO, con un par de métodos básicos, sería la siguiente:

```java
package es.unizar.sisinf.data.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import es.unizar.sisinf.data.db.ConnectionManager;
import es.unizar.sisinf.data.vo.DemoVO;

/**
 * Clase DAO de gestión de objetos DEMO
 * @author
 *
 */
public class DemoDAO {

	private static String findByIdQuery = "SELECT * FROM demo WHERE id = ?";
	private static String findAll = "SELECT * FROM demo";



	/**
	 * Busca un registro en la tabla DEMO por ID
	 * @param id Identificador del registro buscado
	 * @return Objeto DemoVO con el identificador buscado, o null si no se encuentra
	 */
	public DemoVO findById(int id) {

		DemoVO result = null;

		try {
			// Abrimos la conexión e inicializamos los parámetros
			Connection conn = ConnectionManager.getConnection();
			PreparedStatement ps = conn.prepareStatement(findByIdQuery);
			ps.setInt(1, id);

			// Ejecutamos la consulta
			ResultSet rs = ps.executeQuery();

			// Leemos resultados
			if(rs.first()) {
				result = new DemoVO(rs.getInt("id"),rs.getString("name"));
			} else {
				result = null;
			}
			ConnectionManager.releaseConnection(conn);

		} catch(SQLException se) {

			se.printStackTrace();

		} catch(Exception e) {

			e.printStackTrace(System.err);
		}

		return result;
	}



	/**
	 * Devuelve todos los registros de la tabla DEMO
	 * @return Lista de todos los registros de la tabla DEMO
	 */
	public List<DemoVO> findAll(){

		List<DemoVO> result = new ArrayList<DemoVO>();

		try {
			// Abrimos la conexión e inicializamos los parámetros
			Connection conn = ConnectionManager.getConnection();
			PreparedStatement ps = conn.prepareStatement(findAll);
```

```
                        // Ejecutamos la consulta
                        ResultSet rs = ps.executeQuery();

                        // Leemos resultados
                        while(rs.next()) {
                                DemoVO tmp = new DemoVO(rs.getInt("id"),rs.getString("name"));
                                result.add(tmp);
                        }
                        ConnectionManager.releaseConnection(conn);

                } catch(SQLException se) {

                        se.printStackTrace();

                } catch(Exception e) {

                        e.printStackTrace(System.err);
                }

               return result;
        }


}
```

En este ejemplo de clase DAO, la conexión a la base de datos la delegamos en otra clase, ConnectionManager, que es la que abstrae la conexión concreta al gestor de base de datos que utilicemos. Lo importante en este caso es que obtenemos una conexión antes de realizar la consulta, y la liberamos una vez hemos terminado la operación. Si no liberamos la conexión, una nueva llamada a la base de datos podría dejarnos la aplicación atascada (deadlock), en espera de una conexión libre.

Conexión a la base de datos

Para conectarnos a una base de datos, hemos implementado una clase auxiliar, ConnectionManager, que será la encargada de establecer la conexión con la base de datos, y "prestársela" a los objetos DAO que se lo soliciten.

La forma más sencilla y directa de establecer esta conexión, es configurar por código los parámetros de la conexión (IP o nombre del servidor de base de datos, puerto, nombre de la base de datos, usuario y contraseña …), y crear una conexión nueva con esos parámetros.

```
package es.unizar.sisinf.data.db;
import java.sql.*;


/**
 * Clase que abstrae la conexion con la base de datos.
 * @author
 *
 */
public class ConnectionManager {


        // JDBC nombre del driver y URL de BD
         private static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
         private static final String DB_URL = "jdbc:mysql://localhost:3306/sisinf";


         // Credenciales de la Base de Datos
         private static final String USER = "user";
         private static final String PASS = "password";


        // Devuelve una nueva conexion.
         public final static Connection getConnection() throws SQLException {

                Connection conn = null;
                Statement stmt = null;

                try{
                        //STEP 1: Register JDBC driver
                        Class.forName(JDBC_DRIVER);

                        //STEP 2: Open a connection
                        System.out.println("Connecting to database...");
                        conn = DriverManager.getConnection(DB_URL,USER,PASS);

                         return conn;
                 } catch (Exception e) {
                         e.printStackTrace();
                         return null;
                 }

         }


         // Libera la conexion, devolviendola al pool
         public final static void releaseConnection(Connection conn) throws SQLException {
                 conn.close();
         }


}
```

Conexión a la base de datos a través de TOMCAT

Una de las posibilidades que nos ofrecen los servidores de aplicaciones, como Tomcat, es la de gestionar pooles de conexiones a base de datos. La gestión de pooles de conexiones a través de un servidor de aplicaciones tiene varias ventajas:

- Nuestra aplicación accede a la base de datos a través de un alias, no mediante una serie de parametros “hardcodeados” en nuestra aplicación. De esta manera, si cambiara la ubicación física de la base de datos, no sería necesario rehacer y redesplegar la aplicación, sino solo cambiar la configuración del servidore de aplicaciones. Además, podríamos tener varios entornos (desarrollo y producción), y en cada entorno, la misma aplicación accedería a la base de datos correspondiente a cada caso, sin tener que tocar nada. Solo configuraríamos de distinta manera los servidores de aplicaciones.

- La operación más costosa en una conexión a base de datos es, precisamente, abrir la conexión. Una vez establecida, las consultas suelen ser bastante rápidas. Por eso no tiene mucho sentido estar abriendo y cerrando la conexión a la base de datos cada vez que queremos hacer una pequeña consulta, como puede ser la de validar una contraseña al hacer login. Si el servidor es capaz de mantener un pool de conexiones (un paquete más o menos grande de conexiones preabiertas), cuando la aplicación necesita una conexión, el servidor se la “presta”, y cuando la aplicación ya no la necesita, se la devuelve para que sea reutilizada por otra petición. En entornos multiaplicación y multiusuarios, esta opción nos da mucho mejor rendimiento.

Configuración en TOMCAT

¿Cómo configuramos una conexión en TOMCAT, para que pueda ser utilizada por una o varias aplicaciones?

Necesitaremos tocar dos ficheros de configuración de Tomcat: server.xml y context.xml.

Server.xml. Añadiremos la configuración de la conexión a la base de datos, con un alias interno de Tomcat. En la sección “GlobalNamingResources” añadiremos lo siguiente:

```
 <GlobalNamingResources>
...
    <Resource name="jdbc/SIDB"
          global="jdbc/SIDB"
          auth="Container"
          type="javax.sql.DataSource"
          driverClassName="com.mysql.jdbc.Driver"
          url="jdbc:mysql://sisinf-mariadb:3306/sisinf"
          username="root"
          password="sisinf"

          maxActive="100"
          maxIdle="20"
          minIdle="5"
          maxWait="10000"/>

  </GlobalNamingResources>
```

Los parámetros que hay que configurar en cada caso serán:

- name / global: Es el alias interno que le damos a nuestra conexión. Cada conexión que configuremos tendrá un alias distinto.
- driverClassName: La clase que implementa el driver de conexión a nuestra base de datos. En nuestro caso, es el driver de MySql
- url: URL de conexión a nuestra base de datos. Servidor, puerto, nombre de la base de datos…
- username
- password

En el fichero “context.xml” crearemos un alias público, es decir, utilizable por las aplicaciones, que apunte a la conexión antes creada. Dentro del elemento “Context” añadiremos lo siguiente:

```
<ResourceLink name="jdbc/sisinfDB"
                  global="jdbc/SIDB"
                  auth="Container"
                  type="javax.sql.DataSource" />
```

En este caso, *name* es el alias público que le damos al recurso, y *global* apunta a la conexión creada en *server.xml*.

Ahora, nuestra aplicación ya puede conectarse a esta base de datos, utilizando el alias “jdbc/sisinfDB”, sin preocuparse de qué tipo de base de datos es, o en qué servidor está funcionando. Simplemente es una base de datos compatible con JDBC, y podrá utilizarla según dicho estándar.

¿Cómo será ahora nuestro gestor de conexiones?

ConnectionManager para utilizar recursos JNDI

Cuando queramos utilizar *DataSources* definidos en el servidor, en vez de crear las conexiones desde cero, simplemente buscaremos el alias de nuestra conexión en el contexto del servidor, y obtendremos una conexión del mismo:

```
package es.unizar.sisinf.data.db;
import java.sql.*;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.sql.DataSource;

/**
 * Clase que abstrae la conexion con la base de datos.
 * @author
 *
 */
public class ConnectionManager {


	// Devuelve la conexion, para no tener que abrirla y cerrarla siempre.
	 public final static Connection getConnection() throws SQLException {

		try {
			Context initCtx = new InitialContext();
			Context envCtx = (Context) initCtx.lookup("java:comp/env");
			System.out.println(envCtx.toString());
			DataSource ds = (DataSource)envCtx.lookup("jdbc/sisinfDB");
			System.out.println(ds.toString());

			Connection conn = ds.getConnection();
			return conn;
		} catch (Exception e) {
			e.printStackTrace();
			return null;
		}

	}


	 // Libera la conexion, devolviendola al pool
	 public final static void releaseConnection(Connection conn) throws SQLException {
		conn.close();
	 }


}
```

En este código, la única sentencia que necesitaríamos personalizar es:

```
DataSource ds = (DataSource)envCtx.lookup("jdbc/sisinfDB");
```

… en la que debemos indicar el alias que hemos configurado en *context.xml*.

Con esta configuración, ya deberíamos ser capaces de conectarnos a cualquier gestor de base de datos SQL que tengamos levantado y accesible