Tema 8. HIBERNATE

Hibernate es una herramienta de Mapeo objeto-relacional (ORM, Object-Relational Mapping) para la plataforma Java (y disponible también para .Net con el nombre de NHibernate) que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) que permiten establecer estas relaciones.

Hibernate Framework proporciona una facilidad asombrosa para guardar y consultar tus objetos Java sobre una base de datos cualquiera, además de proveerte un camino seguro y lleno de facilidades para que desarrolles tu capa de datos rápida, tranquila y confiadamente.

Hibernate es software libre, distribuido bajo los términos de la licencia GNU LGPL.

Características

Como todas las herramientas de su tipo, Hibernate busca solucionar el problema de la diferencia entre los dos modelos de datos coexistentes en una aplicación: el usado en la memoria de la computadora (orientación a objetos) y el usado en las bases de datos (modelo relacional). Para lograr esto permite al desarrollador detallar cómo es su modelo de datos, qué relaciones existen y qué forma tienen. Con esta información Hibernate le permite a la aplicación manipular los datos de la base operando sobre objetos, con todas las características de la POO. Hibernate convertirá los datos entre los tipos utilizados por Java y los definidos por SQL. Hibernate genera las sentencias SQL y libera al desarrollador del manejo manual de los datos que resultan de la ejecución de dichas sentencias, manteniendo la portabilidad entre todos los motores de bases de datos con un ligero incremento en el tiempo de ejecución.

Hibernate está diseñado para ser flexible en cuanto al esquema de tablas utilizado, para poder adaptarse a su uso sobre una base de datos ya existente. También tiene la funcionalidad de crear la base de datos a partir de la información disponible.

Hibernate ofrece también un lenguaje de consulta de datos llamado HQL (Hibernate Query Language), al mismo tiempo que una API para construir las consultas programáticamente (conocida como "criteria").

Hibernate para Java puede ser utilizado en aplicaciones Java independientes o en aplicaciones Java EE, mediante el componente Hibernate Annotations que implementa el estándar JPA, que es parte de esta plataforma.

Acceso a los datos

Para acceder a los datos de la base de datos hay que indicar a Hibernate donde encontrar este objeto y donde encontrar la base de datos, le dices como se lama tu objeto y como se lama la tabla de tu base de datos que lo representa, le dices cuales son los atributos de tu objeto y exactamente a que columna de la tabla representa cada uno y también su tipo de dato Java.

IMPORTANTE: Utilizando el patrón DAO abstraemos toda la capa de datos de nuestra aplicación y dejamos limpia toda la lógica del negocio. Esto es fundamental a la hora de diseñar una arquitectura prolija y elegante porque es importante conseguir una aplicación que sea completamente independiente de la base de datos que se quiera usar.

Data Access Object, de ahora en adelante DAO, es un **patrón de diseño utilizado para encapsular la interacción de una aplicación con la base de datos**.

Que es Data Access Object (DAO)

La mayoría de las aplicaciones, tienen que persistir datos en algún momento, ya sea serializándolos, guardándolos en una base de datos relacional, o una base de datos orientada a objetos, etc. Para hacer esto, la aplicación interactúa con la base de datos. El "como interactúa" NO debe ser asunto de la capa de lógica de negocio de la aplicación, ya que para eso está la capa de persistencia, que es la encargada de interactuar con la base de datos. Sabiendo esto, podemos decir que DAO es un **patrón de diseño utilizado para crear esta capa de persistencia**.

Pero… ¿de qué sirve tener una capa de persistencia?

Bueno, imaginemos que hicimos una aplicación de gestión para nuestra empresa. Utilizando como motor de base de datos Oracle. Pero no tenemos dividida la capa de lógica de negocio de la de persistencia. Por lo que la interacción con la base de datos se hace directamente desde la capa de lógica de negocio. Nuestra aplicación consiste en muchísimas clases, y gran parte de ellas interactúan con la base de datos (conectándose a la base de datos, guardando y recuperando datos, etc.).

Nuestra aplicación va de maravilla, pero de pronto surge la necesidad de cambiar el motor de la base de datos a PostgreSQL. ¿Cómo solucionarlo?.

Si hubiéramos tenido por separado la capa de lógica de negocio de la de persistencia, habría sido suficiente con modificar la capa de persistencia para que la aplicación pudiera utilizar el nuevo motor de base de datos, sin tener que modificar nada de la capa de lógica de negocio. Pero como en el ejemplo anterior NO usamos una capa de persistencia, sino que interactuamos con la base de datos directamente desde la capa de lógica de negocio, entonces vamos a tener que modificar todas las clases, cambiando todas las consultas SQL, la manera de acceder a la base de datos, etc. para adecuarse al nuevo motor de la base de datos.

Bien, ahora que sabemos porque es importante tener separadas las capas de lógica de negocio y de persistencia, vamos a ver como utilizar el patrón de diseño DAO para crear nuestra propia capa de persistencia.

Como funciona DAO

Como dijimos antes, DAO encapsula el acceso a la base de datos. Por lo que cuando la capa de lógica de negocio necesite interactuar con la base de datos, va a hacerlo a través de la API que le ofrece DAO. Generalmente esta API consiste en métodos CRUD (Create, Read, Update y Delete). Entonces por ejemplo cuando la capa de lógica de negocio necesite guardar un dato en la base de datos, va a llamar a un método create(). Lo que haga este método, es problema de DAO y depende de como DAO implemente el método create(), puede que lo implemente de manera que los datos se almacenen en una base de datos relacional como puede que lo implemente de manera que los datos se almacenen en ficheros de texto. Lo importante es que la capa de lógica de negocio no tiene porque saberlo, lo único que sabe es que el método create() va a guardar los datos, así como el método delete() va a eliminarlos, el método update() actualizarlos, etc. Pero no tiene idea de como interactúa DAO con la base de datos.

En una aplicación, hay tantos DAOs como modelos. Es decir, en una base de datos relacional, por cada tabla, habría un DAO.

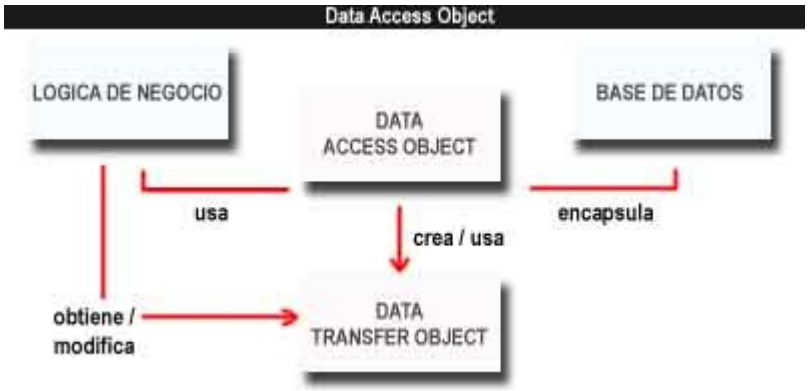
DAO consiste básicamente en una clase que es la que interactúa con la base de datos. Los métodos de esta clase dependen de la aplicación y de lo que queramos hacer. Pero generalmente se implementan los métodos CRUD para realizar las "4 operaciones básicas" de una base de datos.

Bien, nos falta comprender algo más para poder empezar a ver código. Los DTO (Data Transfer Object) o también denominados VO (Value Object). Son utilizados por DAO para transportar los datos desde la base de datos hacia la capa de lógica de negocio y viceversa. Por ejemplo, cuando la capa de lógica de negocio llama al método create(), ¿qué es lo que hace DAO? inserta un nuevo dato… ¿pero qué dato? el que la capa de lógica de negocio le pase como parámetro… ¿y cómo se lo pasa este dato? bueno, a través de un DTO.

Podría decirse que un DTO es un objeto común y corriente, que tiene como atributos los datos del modelo, con sus correspondientes accessors (getters y setters).

Por ejemplo, si tuviéramos una base de datos relacional con una tabla employers, con los campos id, name y salary. Entonces tendríamos que crear una clase EmployerDTO, con los atributos id, name y salary, que van a utilizar la capa de negocio y de persistencia para transportar los datos entre las dos capas.

Entonces cuando la capa de lógica de negocio quiera guardar un dato en la base de datos, va a crear un objeto EmployerDTO, a través de los accessors va a modificar los atributos, y después se lo va a pasar al método create() de DAO. Entonces DAO va a leer los datos del DTO, y los va a guardar en la base de datos. Lo mismo pasaría para eliminar datos. Y para actualizarlos además se le pasaría el ID, para saber que dato actualizar. Para buscar datos, sería parecido, ya que se le pasa al método read() el DTO para usarlo como patrón de búsqueda, pero con la diferencia de que este método tiene valor de retorno, ya que devuelve otro DTO con los datos del resultado de la búsqueda.



Historia

Hibernate fue una iniciativa de un grupo de desarrolladores dispersos alrededor del mundo conducidos por Gavin King.

Tiempo después, JBoss Inc. (empresa comprada por Red Hat) contrató a los principales desarrolladores de Hibernate y trabajó con ellos en brindar soporte al proyecto.

La rama actual de desarrollo de Hibernate es la 3.x, la cual incorpora nuevas características, como una nueva arquitectura Interceptor/Callback, filtros definidos por el usuario, y —opcionalmente— el uso de anotaciones para definir la correspondencia en lugar (o conjuntamente con) los archivos XML. Hibernate 3 también guarda cercanía con la especificación EJB 3.0 (aunque apareciera antes de la publicación de dicha especificación por Java Community Process) y actúa como la espina dorsal de la implementación de EJB 3.0 en Jboss.

Ejemplo:

Hibernate funciona asociando a cada tabla de la base de datos un Plain Old Java Object (POJO, a veces llamado Plain Ordinary Java Object). Un POJO es similar a una Java Bean, con propiedades accesibles mediante métodos setter y getter, como por ejemplo:

```
package net.sf.hibernate.examples.quickstart;

public class Cat {

    private String id;
    private String name;
    private char sex;
    private float weight;

    public Cat() {
    }

    public String getId() {
        return id;
    }

    private void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public char getSex() {
        return sex;
    }

    public void setSex(char sex) {
        this.sex = sex;
    }

    public float getWeight() {
        return weight;
    }

    public void setWeight(float weight) {
        this.weight = weight;
    }

}
```

Para poder asociar el POJO a su tabla correspondiente en la base de datos, Hibernate usa los ficheros hbm.xml.

Para la clase Cat se usa el fichero Cat.hbm.xml para mapearlo con la base de datos. En este fichero se declaran las propiedades del POJO y sus correspondientes nombres de columna en la base de datos, asociación de tipos de datos, referencias, relaciones x a x con otras tablas etc:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>

    <class name="net.sf.hibernate.examples.quickstart.Cat" table="CAT">

        <!-- A 32 hex character is our surrogate key. It's automatically
            generated by Hibernate with the UUID pattern. -->
        <id name="id" type="string" unsaved-value="null" >
            <column name="CAT_ID" sql-type="char(32)" not-null="true"/>
            <generator class="uuid.hex"/>
        </id>

        <!-- A cat has to have a name, but it shouldn' be too long. -->
        <property name="name">
            <column name="NAME" length="16" not-null="true"/>
        </property>

        <property name="sex"/>

        <property name="weight"/>

    </class>

</hibernate-mapping>
```

De esta forma en nuestra aplicación podemos usar el siguiente código para comunicarnos con nuestra base de datos:

```
sessionFactory = new Configuration().configure().buildSessionFactory();
Session session = sessionFactory.openSession();

Transaction tx= session.beginTransaction();

Cat princess = new Cat();
princess.setName("Princess");
princess.setSex('F');
princess.setWeight(7.4f);

session.save(princess);
tx.commit();

session.close();
```

Además tiene la ventaja de que nos es totalmente transparente el uso de la base de datos pudiendo cambiar de base de datos sin necesidad de cambiar una línea de código de nuestra aplicación, simplemente cambiando los ficheros de configuración de Hibernate.

Configuración de Hibernate

Hasta ahora hemos obviado un dato importante en la configuración de Hibernate. Los datos de configuración de la base de datos

Podemos usar un fichero hibernate.properties o hibernate.cfg.xml que debe estar en el path de la aplicación:

hibernate.properties:

```
## MySQL

hibernate.dialect net.sf.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class org.gjt.mm.mysql.Driver
hibernate.connection.driver_class com.mysql.jdbc.Driver
hibernate.connection.url jdbc:mysql:///test
hibernate.connection.username cesar
hibernate.connection.password
```

hibernate.cfg.xml - Conexión mediante Datasource:

```
 v:shape="_x0000_s1026" class="O" style=""
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">

<hibernate-configuration>
 <session-factory>
  <property name="connection.datasource">
    java:comp/env/jdbc/shop</property>

  <property name="dialect">net.sf.hibernate.dialect.MySQLDialect
    </property>
  <property name="use_outer_join">true</property>
  <property name="transaction.factory_class">
    net.sf.hibernate.transaction.JDBCTransactionFactory</property>

  <property name="show_sql">true</property>

  <!-- Mapping files -->
  <mapping resource="com/shop/Category.hbm.xml" />
  <mapping resource="com/shop/Product.hbm.xml" />
 </session-factory>
</hibernate-configuration>
```

hibernate.cfg.xml - Conexión Directa:

```
 v:shape="_x0000_s1026" class="O" style=""
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">

<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">
com.mysql.jdbc.Driver</property>
<property name="connection.url">
```

```
jdbc:mysql://localhost/test</property>
<property name="connection.username">cesar</property>
<property name="connection.password"></property>

<property name="dialect">net.sf.hibernate.dialect.MySQLDialect</property>
<property name="use_outer_join">true</property>
<property
name="transaction.factory_class">net.sf.hibernate.transaction.JDBCTransactionFa
ctory</property>
<property name="show_sql">true</property>

<!-- Mapping files -->
  <mapping resource="com/shop/Category.hbm.xml" />
  <mapping resource="com/shop/Product.hbm.xml" />
 </session-factory>
</hibernate-configuration>
```

En caso de encontrarse ambos ficheros el .properties y el .hbm.xml se usara el .hbm.xml. Desde la aplicación también podemos especificar el fichero a usar:

```
SessionFactory sf = new Configuration()
    .configure("catdb.cfg.xml")
    .buildSessionFactory();
```

En estos ficheros se indican los parámetros de conexión de la base de datos como la base de datos a la que conectar, usuario y password etc.

Un parámetro interesante es el Dialecto de Hibernate. En este parámetro se indica el nombre de la clase que se encargará de comunicarse con la base de datos en el SQL que entienda la base de datos. Este parámetro ha de ser siempre especificado. El valor ha de ser una subclase que herede de `net.sf.hibernate.dialect.Dialect`

Hibernate nos proporciona los siguientes dialectos:

RDBMS	Dialect
DB2	`net.sf.hibernate.dialect.DB2Dialect`
MySQL	`net.sf.hibernate.dialect.MySQLDialect`
SAP DB	`net.sf.hibernate.dialect.SAPDBDialect`
Oracle (any version)	`net.sf.hibernate.dialect.OracleDialect`
Oracle 9	`net.sf.hibernate.dialect.Oracle9Dialect`
Sybase	`net.sf.hibernate.dialect.SybaseDialect`
Sybase Anywhere	`net.sf.hibernate.dialect.SybaseAnywhereDialect`
Progress	`net.sf.hibernate.dialect.ProgressDialect`
Mckoi SQL	`net.sf.hibernate.dialect.MckoiDialect`
Interbase	`net.sf.hibernate.dialect.InterbaseDialect`
Pointbase	`net.sf.hibernate.dialect.PointbaseDialect`
PostgreSQL	`net.sf.hibernate.dialect.PostgreSQLDialect`

RDBMS	Dialect
HypersonicSQL	net.sf.hibernate.dialect.HSQLDialect
Microsoft SQL Server	net.sf.hibernate.dialect.SQLServerDialect
Ingres	net.sf.hibernate.dialect.IngresDialect
Informix	net.sf.hibernate.dialect.InformixDialect
FrontBase	net.sf.hibernate.dialect.FrontbaseDialect

Aquí observamos la gran importancia del fichero de configuración, pues es aquí donde se especifica que base de dato usamos, por lo que si cambiáramos de base de datos bastaría con cambiar este fichero de configuración, manteniendo nuestra aplicación intacta.

Hibernate Query Languaje HQL

Hibernate nos proporciona además un lenguaje con el que realizar consultas a la base de datos.

Este lenguaje es similar a SQL y es utilizado para obtener objetos de la base de datos según las condiciones especificadas en el HQL.

El uso de HQL nos permite usar un lenguaje intermedio que según la base de datos que usemos y el dialecto que especifiquemos será traducido al SQL dependiente de cada base de datos de forma automática y transparente.

Así una forma de recuperar datos de la base de datos con Hibernate sería:

Hibernate

```
Session session =
sessionFactory.openSession();
List cats = null;
try {
categories = session.find("from Cat");

Iterator i = categories.iterator();
while (i.hasNext() == true) {
Cat cat = (Cat)i.next();
...
}
} finally {
session.close();
}
```

JDBC

```
Driver d =
(Driver)
Class.forName("com.mysql.jdbc.Driver").
newInstance();
DriverManager.registerDriver(d);


try {
Connection con =
DriverManager.getConnection(
"jdbc:mysql://yamcha/test",
"cesar",
"");

Statement stmt = con.createStatement();

String select =
"SELECT * from cat";

ResultSet res =
stmt.executeQuery(select);


while (res.next() == true) {
String catID = res.getString("id");
String catName = res.getString("name");

Cat cat = new Cat(catID,catName);
(.......)

list.add(cat);
```

	}
	stmt.close();
	con.commit();
	con.close();
	
	} catch (Throwable ex) {
	System.out.println(" Error visualizando datos ");
	}

De esta forma:

HQL	**SQL**
`from Cat`	select * from cat

Como podemos observar se simplifica considerablemente el código, así como se desacopla el uso de la base de datos de nuestra lógica de aplicación.