

ORM

¿QUÉ APRENDEREMOS?

- ✓ Concepto de mapeo objeto-relacional.
- ✓ Características de las herramientas ORM.
- ✓ Herramientas ORM más empleadas.
- ✓ Instalación de una herramienta ORM.
- ✓ Estructura de un fichero de mapeo: elementos y propiedades.
- ✓ Mapeo de colecciones, relaciones y herencia.
- ✓ Clases persistentes.
- ✓ Sesiones. Estados de un objeto.
- ✓ Carga, almacenamiento y modificación de objetos.
- ✓ Consultas SQL.
- ✓ Lenguajes propios de la herramienta ORM.
- ✓ Gestión de transacciones.

Concepto de mapeo objeto-relacional

El mapeo objeto-relacional (ORM) es una técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional, utilizando un motor de persistencia.

En la práctica, esto crea una base de datos orientada a objetos virtual, sobre la base de datos relacional. Esto posibilita el uso de las características propias de la orientación a objetos.

Las herramientas ORM nos permiten crear una capa de acceso a datos; una forma sencilla y válida de hacerlo, es crear una clase por cada tabla de la base de datos y mapearlas una a una.

Una gran ventaja, es que estas herramientas aportan un lenguaje de consultas orientado a objetos propio y totalmente independiente de la base de datos que usemos, lo que nos permitirá migrar de una base de datos a otra sin tocar nuestro código, simplemente debemos modificar el fichero de configuración.

Características de las herramientas ORM

Ventajas:

- ❖ Ayudan a reducir el tiempo de desarrollo software
- ❖ Abstracción de la base de datos
- ❖ Reutilización
- ❖ Permiten generar código más limpio
- ❖ Independientes de la base de datos
- ❖ Incentivan la portabilidad y escalabilidad de los programas software

Inconvenientes:

- ❖ Mayor lentitud en el acceso a datos

Herramientas ORM más empleadas

Java

- ❖ Hibernate
- ❖ Jooq
- ❖ ActiveJDBC

.NET

- ❖ Entity Framework (también versión .NET Core)
- ❖ NHibernate
- ❖ Dapper

PHP

- ❖ Doctrine

.Python

- ❖ Django

Hibernate

Hibernate es una herramienta de mapeo objeto-relacional para la plataforma Java que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante ficheros declarativos (XML que permiten establecer las relaciones, y también mediante anotaciones).

Se ha convertido en el estándar de facto para almacenamiento persistente cuando queremos independizar la capa de negocio del almacenamiento de la información. Esta capa de persistencia permite abstraer al programador Java de las particularidades de una determinada base de datos proporcionando clases que envolverán los datos recuperados de las filas de las tablas. Hibernate busca solucionar la diferencia entre los dos modelos de datos usados para organizar y manipular datos: el modelo de objetos proporcionado por el lenguaje de programación y el modelo relacional usado en las bases de datos.

Hibernate - Arquitectura

Para almacenar y recuperar estos objetos de la base de datos, el desarrollador debe hacer uso del motor de Hibernate mediante un objeto especial que es la sesión (clase Session), equiparable al concepto de conexión JDBC.

La clase Session ofrece métodos para interactuar con la BD tal como se hace con una conexión JDBC, con la diferencia que resulta más sencillo su uso. Por ejemplo, guardar un objeto lo hacemos con el método save pasando de parámetro el objeto completo, sin necesidad de especificar sentencias en el código.

Una sentencia Session no consume mucha memoria y su creación y destrucción es muy barata. Esto es importante, ya que nuestra aplicación necesitará crear y destruir sesiones todo el tiempo.

Hibernate - Interfaces

- ❖ **SessionFactory** : permite obtener instancias Session. Esta interfaz debe compartirse entre muchos hilos de ejecución. Si la aplicación accede a varias bases de datos se necesitará una SessionFactory por cada base de datos.
- ❖ **Configuration**: se utiliza para configurar Hibernate. La aplicación utiliza una instancia de Configuration para especificar la ubicación de los documentos que indican el mapeado de los objetos y propiedades específicas de Hibernate.
- ❖ **Query**: permite realizar consultas a la base de datos y controla cómo se ejecutan dichas consultas. Las consultas se escriben en HQL o en el dialecto SQL nativo de la base de datos que estemos utilizando. Una instancia Query se utiliza para enlazar los parámetros de la consulta, limitar el número de resultados devueltos y para ejecutar dicha consulta.
- ❖ **Transaction**: nos permite asegurar que cualquier error que ocurra entre el inicio y el final de la transacción produzca el fallo de la misma.

Hibernate - Instalación y configuración

- ❖ Incluir librerías: debemos incluir tanto las librerías de Hibernate como de los drivers de conexión a base de datos que vayamos a usar.
- ❖ Creación de las entidades: se deberán implementar las clases que representan a los objetos asociados a los datos persistentes en BD.
- ❖ Configuración de archivos de propiedades y mapeo: el motor de persistencia de Hibernate utiliza una serie de archivos de configuración XML a través de los cuales se especifica, por un lado, de conexión con la base de datos, y por otro, la relación entre las entidades y tablas y campos de la base de datos.
- ❖ Utilización del API Hibernate: una vez completados los pasos anteriores, las aplicaciones ya pueden hacer uso del API de Hibernate para recuperar y persistir entidades en la base de datos.

Hibernate - Librerías

El primer paso para poder hacer uso de Hibernate en nuestro proyecto es incluir sus librerías. Como Hibernate se conectará a una base de datos, debemos incluir en nuestras dependencias el driver de conexión a la base de datos. Si utilizamos Maven podemos hacerlo fácilmente incluyendo ambas dependencias en nuestro pom.xml.

```
<!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-core -->
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.6.14.Final</version>
</dependency>

<!-- https://mvnrepository.com/artifact/com.microsoft.sqlserver/mssql-jdbc -->
<dependency>
  <groupId>com.microsoft.sqlserver</groupId>
  <artifactId>mssql-jdbc</artifactId>
  <version>11.2.2.jre8</version>
</dependency>
```

Hibernate - Clases persistentes

Las clases que se usarán en el mapeo de Hibernate deben cumplir los siguientes requisitos:

- ❖ Deberá contar con un constructor sin parámetros
- ❖ Tendrá métodos set/get para acceder a sus datos
- ❖ Debe implementar la interfaz Serializable

Habitualmente, se creará una clase de entidad por cada tabla de la base de datos con la que se vaya a trabajar, de modo que un objeto de entidad representará a un registro de dicha tabla, lo que implica que cada atributo o campo de la entidad estará asociado a una columna de la tabla correspondiente.

Hibernate - Clase Tarea

```
package org.example.model;

import java.io.Serializable;
import java.sql.Timestamp;
import java.util.Set;

public class Tarea implements Serializable {

    private int id;
    private String nombre;
    private String descripcion;
    private Timestamp fecha;
    private String estado;
    private Set<Subtarea> subtareas;

    public Tarea() {
    }

    public int getId() { return id; }

    public void setId(int id) { this.id = id; }

    public String getNombre() { return nombre; }

    public void setNombre(String nombre) { this.nombre = nombre; }

    public String getDescripcion() { return descripcion; }

    public void setDescripcion(String descripcion) { this.descripcion = descripcion; }
```

```
    public Timestamp getFecha() { return fecha; }

    public void setFecha(Timestamp fecha) { this.fecha = fecha; }

    public String getEstado() { return estado; }

    public void setEstado(String estado) { this.estado = estado; }

    public Set<Subtarea> getSubtareas() { return subtareas; }

    public void setSubtareas(Set<Subtarea> subtareas) { this.subtareas = subtareas; }

    @Override
    public String toString() {
        return "Tarea{" +
            "id=" + id +
            ", nombre='" + nombre + '\'' +
            ", descripcion='" + descripcion + '\'' +
            ", fecha=" + fecha +
            ", estado='" + estado + '\'' +
            ", subtareas=" + subtareas +
            '}';
    }
}
```

Hibernate - Clase Subtarea

```
package org.example.model;

import java.io.Serializable;

public class Subtarea implements Serializable {

    private int id;
    private int idTarea;
    private String nombre;
    private Tarea tarea;

    public Subtarea() {
    }

    public int getId() { return id; }

    public void setId(int id) { this.id = id; }

    public String getNombre() { return nombre; }

    public void setNombre(String nombre) { this.nombre = nombre; }


    public int getIdTarea() { return idTarea; }

    public void setIdTarea(int idTarea) { this.idTarea = idTarea; }
```

```
public Tarea getTarea() { return tarea; }

public void setTarea(Tarea tarea) { this.tarea = tarea; }

@Override
public String toString() {
    return "Subtarea{" +
        "id=" + id +
        ", idTarea=" + idTarea +
        ", nombre='" + nombre + '\'' +
        // ", tarea=" + tarea +
    '}';
}
```

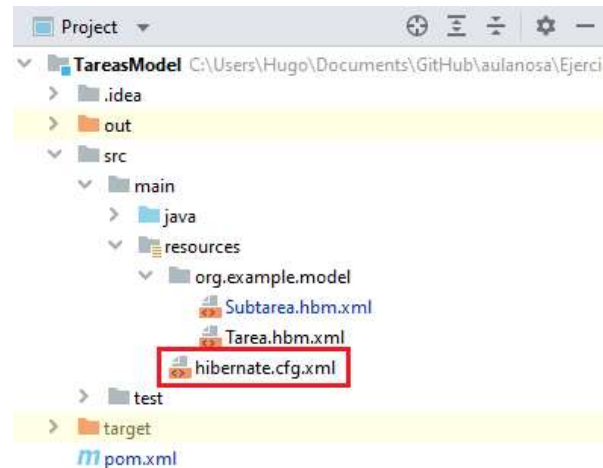


Si quitamos comentario a esta línea se embuclará, ya que en el toString de tarea le hemos indicado que muestre las subtareas asociadas, y si quitamos el comentario le estaríamos diciendo que al mostrar la subtarea nos vuelva a mostrar la tarea, etc...

Hibernate - Archivo HIBERNATE.CFG.XML

Se trata del archivo de configuración principal de Hibernate, en él se deben indicar las propiedades de conexión a la base de datos contra la que se quiere mapear las entidades.

Este archivo debe estar en el raíz de nuestra aplicación. Al estar trabajando con Maven, debemos crear dicho archivo en nuestra carpeta de recursos (src/main/resources).



Hibernate - Ejemplo de archivo HIBERNATE.CFG.XML

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

1  <session-factory>

    <!-- Propiedades de conexión a base de datos -->
    <property name="hibernate.dialect">org.hibernate.dialect.SQLServerDialect</property>
    <property name="hibernate.connection.driver_class">com.microsoft.sqlserver.jdbc.SQLServerDriver</property>
    <property name="hibernate.connection.url">jdbc:sqlserver://localhost:1433;databaseName=Tareas</property>
    <property name="hibernate.connection.username">sa</property>
    <property name="hibernate.connection.password">ADMINO</property>
    <property name="hibernate.default_schema">dbo</property>

    <!-- Escriba todas las sentencias SQL en la consola. -->
    <property name="hibernate.show_sql">true</property>
    <!-- Validar el esquema de la base de datos -->
    <property name="hibernate.hbm2ddl.auto">validate</property>
    <!-- Proporciona una estrategia personalizada para el alcance de la sesión actual.-->
    <property name="hibernate.current_session_context_class">thread</property>

    <!-- MAPEOS -->
    <mapping resource="org/example/model/Tarea.hbm.xml"/>
    <mapping resource="org/example/model/Subtarea.hbm.xml"/>

1  </session-factory>

</hibernate-configuration>
```

Hibernate - Elementos HIBERNATE.CFG.XML

De los elementos configurables en este archivo, destacamos los siguientes:

- ❖ `<session-factory>`: en este elemento se definen las propiedades necesarias para poder establecer una sesión Hibernate sobre una determinada base de datos.
- ❖ `<property>`: cada elemento define una propiedad necesaria para que Hibernate pueda establecer conexiones con la base de datos. Entre las propiedades más importantes destacamos las siguientes:
 - ❖ `hibernate.dialect`: se trata de la clase que le indica con que tipo de base de datos vamos a trabajar.
 - ❖ `hibernate.connection.driver_class`: clase correspondiente al driver que deberá utilizar Hibernate para conectar con la base de datos.
 - ❖ `hibernate.connection.url`: cadena de conexión a la base de datos.
 - ❖ `hibernate.connection.username`: usuario de la base de datos
 - ❖ `hibernate.connection.password`: password de la base de datos
- ❖ `<mapping>`: mediante el elemento de mapping le indicamos a Hibernate la localización de los archivos de mapeo ORM de cada una de las clases entidad.

Hibernate - Sesiones

Para poder utilizar la persistencia en Hibernate es necesario definir un objeto Session utilizando la clase SessionFactory.

La sesión corresponde con un objeto que representa una unidad de trabajo con la base de datos.

La sesión nos permite representar el gestor de persistencia, ya que dispone de una API básica que nos permite cargar y guardar objetos.

Para poder crear la sesión lo haremos de la siguiente manera:

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

Hibernate - Sesiones - Métodos

La clase Session tiene como métodos más destacables los siguientes:

- ❖ beginTransaction(): método para iniciar transacciones.
- ❖ save(): método para hacer un objeto persistente en la base de datos.
- ❖ delete(): método para eliminar los datos de un objeto en la base de datos.
- ❖ update(): método para modificar un objeto.
- ❖ get(): método para recuperar un objeto.
- ❖ createQuery(): método para crear una consulta HQL y así poder ejecutarla sobre la base de datos.
- ❖ createNativeQuery(): método para crear una consulta SQL y así poder ejecutarla sobre la base de datos.

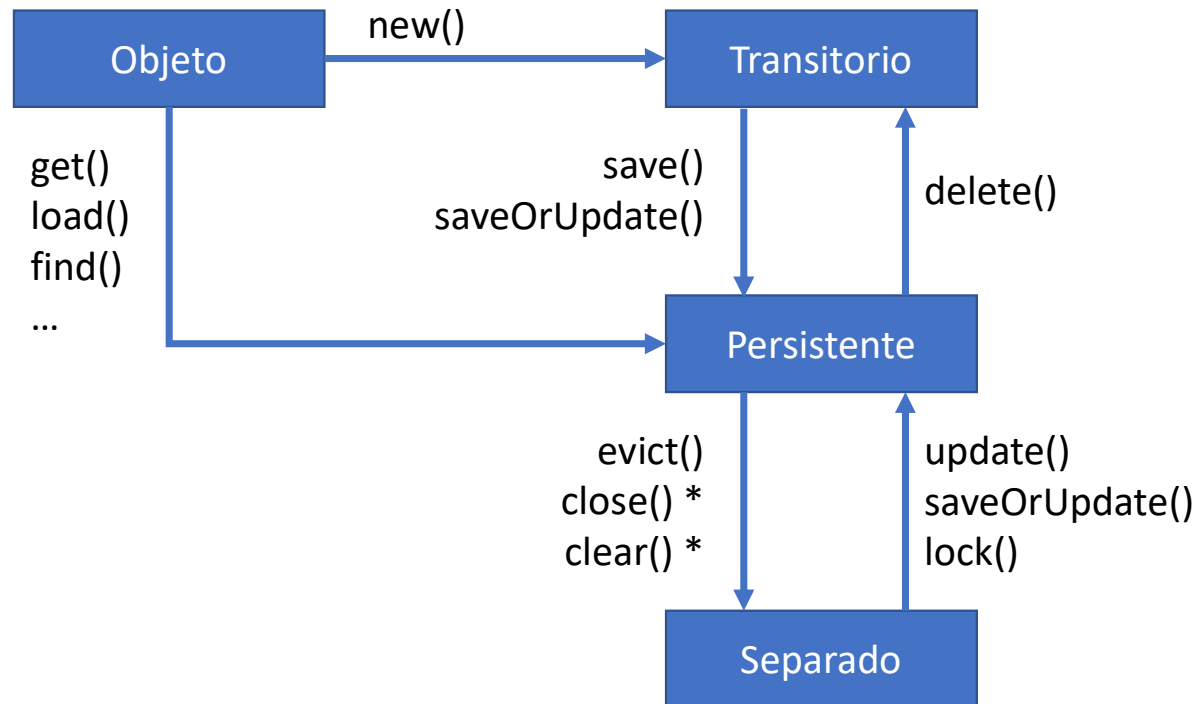
Siempre debemos cerrar la sesión después de usarla con el método close().

Hibernate - Estados de un objeto

Los estados en los que se puede encontrar un objetos son:

- ❖ **Transitorio (Transient)** : un objeto es transitorio si se acaba de crear una instancia con el operador new y no está asociado con una sesión de Hibernate. No tiene representación persistente en la base de datos y no se le ha asignado ningún valor de identificador.
- ❖ **Persistente (Persistent)** : una instancia persistente tiene una representación en la base de datos y un valor de identificador. Es posible que solo se haya guardado o cargado; sin embargo, está por definición en el ámbito de una sesión. Hibernate detectará cualquier cambio realizado en un objeto en estado persistente y sincronizará el estado con la base de datos cuando se complete la unidad de trabajo.
- ❖ **Separado (Detached)** : una instancia separada es un objeto que ha sido persistente, pero su sesión se ha cerrado. La referencia al objeto sigue siendo válida, por supuesto, y la instancia separada podría incluso modificarse en este estado. Una instancia desconectada se puede volver a adjuntar a una nueva sesión en un momento posterior, haciendo que (y todas las modificaciones) sean persistentes nuevamente.

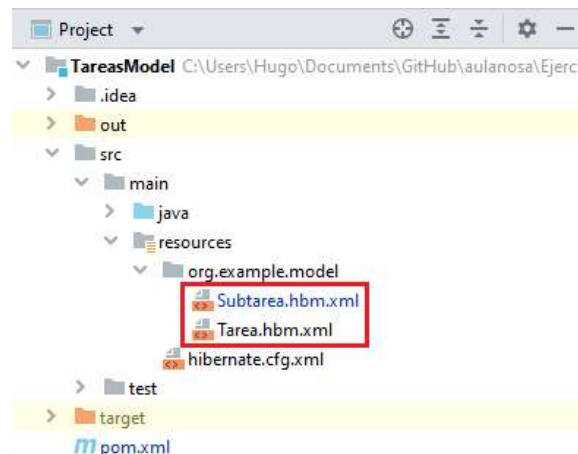
Hibernate - Estados de un objeto



Hibernate - Ficheros de mapeo

Los archivos de configuración ORM (hbm) definen las reglas de mapeo entre las clases de entidad y las bases de datos. Por cada clase de entidad que vaya a ser gestionada por hibernate deberá definirse un archivo de configuración de nombre XXXXXXXXXX.hbm.xml.

Estos archivos habitualmente estarán situados en el mismo paquete en el que se encuentre la clase de entidad. Al estar trabajando con Maven, debemos crear dicho paquete en nuestra carpeta de recursos (src/main/resources).



Hibernate - Ejemplo ficheros de mapeo

Tarea.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="org.example.model.Tarea" table="tarea">
        <id name="id" column="id">
            <generator class="native"></generator>
        </id>
        <property name="nombre" type="string" column="nombre"/>
        <property name="descripcion" type="string" column="descripcion"/>
        <property name="fecha" type="timestamp" column="fecha"/>
        <property name="estado" type="string" column="estado"/>
        <set name="subareas" table="subareas" cascade="all">
            <key column="idTarea"/>
            <one-to-many class="org.example.model.Subtarea"/>
        </set>
    </class>
</hibernate-mapping>
```

Subtarea.hbm.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="org.example.model.Subtarea" table="subtarea">
        <id name="id" column="id">
            <generator class="native"></generator>
        </id>
        <property name="nombre" type="string" column="nombre"/>
        <property name="idTarea" type="int" column="idTarea"/>
        <many-to-one name="tarea" column="idTarea" not-null="true"
            insert="false" update="false"/>
    </class>
</hibernate-mapping>
```

Hibernate - Ficheros de mapeo

Las reglas de mapeo entre la entidad y la base de datos se definen en un elemento `<class>` que a su vez forma parte del elemento raíz `<hibernate-mapping>`.

Los atributos de `<class>` son:

- ❖ `name` : nombre cualificado de la clase de entidad.
- ❖ `table`: nombre de la tabla de la base de datos con la que será mapeada la entidad.
- ❖ `catalog`: se trata de un elemento opcional, que indica el nombre de la base de datos a la que pertenece la tabla.

Hibernate - Ficheros de mapeo

Dentro del elemento `<class>` se deberán indicar los siguientes elementos para definir las reglas de mapeo:

- ❖ `<id>`: identifica la clave primaria de la entidad.
 - ❖ `column`: nombre de la columna de la base de datos con la que será mapeado el campo clave.
 - ❖ `<generator>`: estrategia de generación de la clave primaria.
- ❖ `<property>`: la manera en que cada uno de los campos de la entidad será persistidos.
 - ❖ `name`: nombre del método get/set correspondiente (omitiendo las palabras set/get).
 - ❖ `type`: tipo del método get/set.
 - ❖ `not-null`: en caso de que la columna no admita valores nulos se debe indicar a true.
 - ❖ `length`: para columnas de tipo texto, se indicará la longitud del campo en base de datos.
 - ❖ `column`: es opcional, si no se indica Hibernate supondrá que el campo debe ser mapeado a una columna cuyo nombre coincida con el de la propiedad.

Hibernate - Ficheros de mapeo - Relaciones

En Hibernate se puede establecer relaciones entre entidades, facilitando así la obtención de objetos de una entidad que se encuentran relacionados con otras.

En el modelo relacional las tablas de una base de datos se relacionan a través de columnas comunes que existen en ambas tablas, a través de las cuales se puede identificar a los registros de una tabla que se corresponden con los registros de la otra. La relación entre tablas permite que se puedan establecer relaciones entre entidades dentro de la capa de persistencia.

Las entidades se relacionan a través de un atributo de la clase entidad, en el que se almacenará el objeto u objetos de la entidad relacionada que se corresponden con el objeto de la entidad principal.

Hibernate - Ficheros de mapeo - Relaciones - Uno a muchos

Se deberá emplear el elemento `<set>` o `<bag>` en lugar de `<property>`. Si usamos `<set>` debemos asegurarnos que nuestra clase de mapeo use la clase `Set` para la entidad relacionada, si usamos `<bag>` debemos usar la clase `List`.

```
public class Tarea implements Serializable {  
  
    private int id;  
    private String nombre;  
    private String descripcion;  
    private Timestamp fecha;  
    private String estado;  
    private Set<Subtarea> subtareas;  
}
```

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE hibernate-mapping PUBLIC  
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  
<hibernate-mapping>  
    <class name="org.example.model.Tarea" table="tarea">  
        <id name="id" column="id">  
            <generator class="native"></generator>  
        </id>  
        <property name="nombre" type="string" column="nombre"/>  
        <property name="descripcion" type="string" column="descripcion"/>  
        <property name="fecha" type="timestamp" column="fecha"/>  
        <property name="estado" type="string" column="estado"/>  
        <set name="subtareas" table="subtareas" cascade="all" lazy="true">  
            <key column="idTarea"/>  
            <one-to-many class="org.example.model.Subtarea"/>  
        </set>  
    </class>  
</hibernate-mapping>
```

Hibernate - Ficheros de mapeo - Relaciones - Uno a muchos

El elemento `<set>/<bag>` deberá definir los siguientes atributos y subelementos:

- ❖ `name`: nombre de la propiedad que da acceso.
- ❖ `table`: nombre de la tabla asociada a la entidad a la que pertenecen los objetos almacenados.
- ❖ `cascade`: si queremos que una operación de creación, eliminación o actualización de una entidad sea propagada a las entidades relacionadas, habrá que indicarlo en este atributo. Es opcional, si no se especifica no se propagará ninguna operación.
- ❖ `lazy`: determina la forma en la que serán cargados los objetos relacionados en la colección.
- ❖ `fetch`: indica que tipo de operación se va a lanzar sobre la base de datos para recuperar la colección.
- ❖ `<key>`: a través de su atributo `column`, se indica el nombre del campo que es clave ajena, es decir, la columna de la tabla asociada a la otra entidad que lo relaciona con esta.
- ❖ `<one-to-many>`: en su atributo `class` se indica la clase de entidad relacionada.

Hibernate - Ficheros de mapeo - Relaciones - Muchos a uno

Se usará el elemento `<many-to-one>`. Debemos asegurarnos que nuestra clase de mapeo tenga definido una propiedad de la entidad relacionada.

```
public class Subtarea {  
  
    private int id;  
    private int idTarea;  
    private String nombre;  
    private Tarea tarea;  
}
```

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE hibernate-mapping PUBLIC  
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  
<hibernate-mapping>  
    <class name="org.example.model.Subtarea" table="subtarea">  
        <id name="id" column="id">  
            <generator class="native"></generator>  
        </id>  
        <property name="nombre" type="string" column="nombre"/>  
        <property name="idTarea" type="int" column="idTarea"/>  
        <many-to-one name="tarea" column="idTarea" not-null="true"  
            insert="false" update="false"/>  
    </class>  
</hibernate-mapping>
```

Hibernate - Ficheros de mapeo - Relaciones - Muchos a uno

El elemento <many-to-one> dispone de los siguientes atributos:

- ❖ name: nombre de la propiedad que da acceso.
- ❖ column: nombre de la columna con la que está mapeado.
- ❖ not-null: indica si admite o no valores nulos.
- ❖ insert: valor boolean que indica si la entidad relacionada se insertará en la base de datos a la hora de crear la entidad.
- ❖ update: valor boolean que indica si la entidad relacionada se actualizará en la base de datos a la hora de actualizar la entidad.

Hibernate - Ficheros de mapeo - Relaciones - Muchos a muchos

Este tipo de relaciones, en base de datos tienen una tabla intermedia. Por ejemplo, un empleado puede realizar muchas tareas y una tarea puede tener muchos empleados asignados. En ese caso, se crea una tabla intermedia empleadoTarea con ambos ids para guardar las relaciones. Esta tabla intermedia, no se creará como entidad, basta con indicarla en el mapeo.

Se deberá emplear el elemento `<set>` o `<bag>` en lugar de `<property>`. Si usamos `<set>` debemos asegurarnos que nuestra clase de mapeo use la clase Set para la entidad relacionada, si usamos `<bag>` debemos usar la clase List.

```

dbo.empleadoTarea
├── Columns
│   ├── id (PK, int, not null)
│   ├── idTarea (int, not null)
│   └── idEmpleado (int, not null)

```

```

public class Empleado {

    private int id;
    private String nombre;
    private Set<Tarea> tareas;
}

```

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="org.example.model.Empleado" table="empleado">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="nombre" type="string" column="nombre"/>
        <set name="tareas" table="empleadoTarea" fetch="select" cascade="all">
            <key column="idEmpleado"/>
            <many-to-many class="org.example.model.Tarea" column="idTarea"/>
        </set>
    </class>
</hibernate-mapping>

```

Hibernate - Ficheros de mapeo - Relaciones - Muchos a muchos

El elemento `<set>/<bag>` deberá definir los siguientes atributos y subelementos:

- ❖ `name`: nombre de la propiedad que da acceso.
- ❖ `table`: nombre de la tabla intermedia de la relación.
- ❖ `cascade`: si queremos que una operación de creación, eliminación o actualización de una entidad sea propagada a las entidades relacionadas, habrá que indicarlo en este atributo. Es opcional, si no se especifica no se propagará ninguna operación.
- ❖ `lazy`: determina la forma en la que serán cargados los objetos relacionados en la colección.
- ❖ `fetch`: indica que tipo de operación se va a lanzar sobre la base de datos para recuperar la colección.
- ❖ `<key>`: a través de su atributo `column`, se indica el campo que referencia a nuestra entidad.
- ❖ `<many-to-many>`: en su atributo `class` se indica la clase de entidad relacionada.

Hibernate - Gestión de transacciones

En Hibernate, los límites de las transacciones de la base de datos o el sistema son siempre necesarios. Ninguna comunicación con la base de datos puede darse fuera de una transacción de la base de datos (no existe modo auto-commit).

Es aconsejable siempre usar límites de transacción claros, incluso para las operaciones de sólo lectura.

Para crear una transacción en Hibernate, llamaremos al método `beginTransaction()` del objeto `Session`. Una vez realizadas las operaciones que abarque la transacción, podremos invocar al método `commit()` o `rollback()` del objeto `Session` anterior.

```
Transaction transaction = session.beginTransaction();  
session.save(subtarea);  
transaction.commit();
```


Hibernate - Almacenamiento de objetos

Para guardar un objeto (hacerlo persistente) se usa el método `save(objeto)`. Los pasos son:

1. Abrir una transacción
2. Crear un objeto
3. Hacerlo persistente con el método `save`
4. Confirmar transacción

```
public static void crearTarea() {  
    Session session = sessionFactory.getCurrentSession();  
  
    DateFormat dateFormat = new SimpleDateFormat( pattern: "dd/MM/yyyy HH:mm");  
    Date date = null;  
    try {  
        date = dateFormat.parse( source: "23/01/2023 12:30");  
    } catch (ParseException e) {  
        throw new RuntimeException(e);  
    }  
    long time = date.getTime();  
    Tarea tarea = new Tarea( nombre: "Obra Aula Nosa",  
                             descripcion: "Obra en clase 0 Aula Nosa",  
                             new Timestamp(time), estado: "Pendiente");  
  
    Transaction transaction = session.beginTransaction();  
    session.save(tarea);  
    transaction.commit();  
  
    session.close();  
}
```

Hibernate - Lectura de objetos

Para recuperar un objeto de la base de datos se usa el método `get(Class, clave privada objeto)`.

```
public static void leerTarea() {  
    Session session = sessionFactory.getCurrentSession();  
  
    Transaction transaction = session.beginTransaction();  
    Tarea tarea = session.get(Tarea.class, serializable: 1);  
    System.out.println(tarea);  
    for(Subtarea subtarea : tarea.getSubtareas()) {  
        System.out.println(subtarea);  
    }  
    transaction.commit();  
  
    session.close();  
}
```

Hibernate - Modificar objetos

Para modificar un objeto existente en la base de datos se usa el método `update(Objeto)`. Para que un objeto pueda ser modificado, debe existir con anterioridad. Session ofrece un método llamado `saveOrUpdate(Objeto)` para los casos en que no conocemos si el objeto se guardó con anterioridad, si no se guardó realizará un save y si ya existe realizará un update.

```
public static void modificarTarea() {  
    Session session = sessionFactory.getCurrentSession();  
  
    Transaction transaction = session.beginTransaction();  
    Tarea tarea = session.get(Tarea.class, serializable: 1);  
    tarea.setEstado("Realizada");  
    session.update(tarea);  
    transaction.commit();  
  
    session.close();  
}
```

Hibernate - Eliminar objetos

Para borrar un objeto desde la base de datos el método que se utiliza es delete(Objeto).

```
public static void eliminarTarea() {  
    Session session = sessionFactory.getCurrentSession();  
  
    Transaction transaction = session.beginTransaction();  
    Tarea tarea = session.get(Tarea.class, serializable: 2);  
    session.delete(tarea);  
    transaction.commit();  
  
    session.close();  
}
```

Hibernate - Consultas SQL

Hibernate también nos permitirá realizar consultas nativas en SQL.

Las consultas SQL nativas son representadas con una instancia de `org.hibernate.Query`. Esta interfaz ofrece métodos para ligar parámetros, manejo del conjunto de resultado, y para la ejecución de la consulta real. Siempre se debe obtener la Query utilizando el objeto Session actual.

Para realizar una consulta nativa usaremos el método `createNativeQuery()` al cual se le pasará en un String la consulta HQL.

Ejemplo:

```
Query<Tarea> q = session.createNativeQuery("Select * from Tareas",Tarea.class);  
List<Tarea> lista = q.list();
```

Hibernate - Consultas SQL - Ejemplo

```
public static void consultarTareasSQL() {  
    Session session = sessionFactory.getCurrentSession();  
  
    Transaction transaction = session.beginTransaction();  
    Query<Tarea> query = session.createNativeQuery("Select * from Tareas", Tarea.class);  
    List<Tarea> listaTareas = query.list();  
    for(Tarea tarea : listaTareas) {  
        System.out.println(tarea);  
        for(Subtarea subtarea : tarea.getSubtareas()) {  
            System.out.println(subtarea);  
        }  
    }  
    transaction.commit();  
    session.close();  
}
```

Hibernate - Lenguajes propios de la herramienta ORM - HQL

Hibernate soporta un lenguaje de consulta orientado a objetos denominado HQL que es fácil de usar y potente. Este lenguaje es una extensión orientada a objetos de SQL.

Las consultas HQL y SQL nativas son representadas con una instancia de `org.hibernate.Query`. Esta interfaz ofrece métodos para ligar parámetros, manejo del conjunto de resultado, y para la ejecución de la consulta real. Siempre se debe obtener la Query utilizando el objeto Session actual.

Para realizar una consulta usaremos el método `createQuery()` al cual se le pasará en un String la consulta HQL.

Ejemplo:

```
Query q = session.createQuery("from Departamento");  
List<Departamentos> lista = q.list();
```

Hibernate - Lenguajes propios de la herramienta ORM - HQL

Hibernate soporta parámetros con nombre y parámetros del estilo JDBC (?) en las consultas HQL, aunque siempre es recomendable referirse a ellos por el nombre para no ser dependientes del orden de los mismos. Los parámetros con nombre son identificadores de la forma :nombre en la cadena de consulta.

Para asignar valores a los parámetros se utilizan los métodos setXXX de la clase Query.

Ejemplo:

```
Query q = session.createQuery("from Empleado  
                               where nombre = :nombre and edad = :edad")  
q.setString("nombre","Pepe");  
q.setInteger("edad",30);
```


Hibernate - Lenguajes propios de la herramienta ORM - HQL

Hibernate soporta parámetros con nombre y parámetros del estilo JDBC (?) en las consultas HQL, aunque siempre es recomendable referirse a ellos por el nombre para no ser dependientes del orden de los mismos. Los parámetros con nombre son identificadores de la forma :nombre en la cadena de consulta.

Para asignar valores a los parámetros se utilizan los métodos `setParameter` de la clase `Query`.

Ejemplo:

```
Query<Tarea> q = session.createQuery("from Tareas where estado = :estado",Tarea.class)
q.setParameter("Estado","Pendiente", org.hibernate.type.StringType.INSTANCE);
```

Hibernate - Lenguajes propios de la herramienta ORM - HQL - Ejemplo

```
public static void consultarTareasHQL() {  
    Session session = sessionFactory.getCurrentSession();  
  
    Transaction transaction = session.beginTransaction();  
    Query<Tarea> query = session.createQuery( s: "from Tarea where estado = :estado", Tarea.class);  
    query.setParameter( s: "estado", o: "Realizada", org.hibernate.type.StringType.INSTANCE);  
    List<Tarea> listaTareas = query.list();  
    for(Tarea tarea : listaTareas) {  
        System.out.println(tarea);  
        for(Subtarea subtarea : tarea.getSubtareas()) {  
            System.out.println(subtarea);  
        }  
    }  
    transaction.commit();  
    session.close();  
}
```

Hibernate - HibernateUtil

El uso de Hibernate nos obliga a tener siempre disponible una referencia al objeto `SessionFactory` para que cualquier clase pueda tener acceso al objeto `Session` y por lo tanto a todas las funcionalidades de Hibernate.

El problema de acceder al objeto `SessionFactory` es que cualquier clase podría necesitarlo por lo que deberemos hacérselo llegar de alguna forma.

En cualquier libro o tutorial que hable sobre Hibernate acabará hablándose de la clase `HibernateUtil`. Esta clase que debemos crearnos nosotros y que no está incluida en Hibernate contiene código estático que inicializa Hibernate y crea el objeto `SessionFactory`. Se incluye además un método estático que da acceso al objeto `SessionFactory` que se ha creado.

Hibernate - HibernateUtil - Ejemplo

HibernateUtil.java

```
package org.example;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            sessionFactory = new Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            System.err.println("Error al crear SessionFactory : " + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

Cómo usarlo

```
public static void eliminarTarea() {
    Session session = HibernateUtil.getSessionFactory().getCurrentSession();

    Transaction transaction = session.beginTransaction();
    Tarea tarea = session.get(Tarea.class, serializable: 2);
    session.delete(tarea);
    transaction.commit();

    HibernateUtil.closeSessionFactory();
}
```

Hibernate - Ejercicio

Crear aplicación que permita realizar las siguientes operaciones sobre la base de datos Northwind:

- Operaciones CRUD de proveedores (Suppliers)
- Listar proveedores de una ciudad concreta mediante SQL nativo
- Listar todos los proveedores mediante HQL
- Operaciones CRUD de categorías (Categories)
- Listar todas las categorías mediante SQL nativo
- Operaciones CRUD de productos (Products)
- Obtener cuantos productos tenemos en stock (UnitsInStock) mediante SQL nativo
- Listar todos los productos, mostrando toda la información de su proveedor y categoría

Hibernate - Mapeo por anotaciones

Desde la versión 3 Hibernate soporta usar anotaciones de Java para especificar la relación entre las clases y las tablas de la base de datos.

Si hacemos uso de las anotaciones, ya no es necesario un fichero de configuración ORM por cada clase de nuestro proyecto, pues los mapeos se indicarán en la propia clase mediante las anotaciones. De esta manera tendremos que mantener menos ficheros en nuestro proyecto, además de reducir el tiempo de codificación, ya que la mayoría de las opciones ya tienen valores por defecto en los cuales nos ahorramos ciertas anotaciones.

El fichero de configuración de Hibernate (hibernate.cfg.xml) debemos mantenerlo en el proyecto igualmente, y además debemos igualmente indicar que clases debe mapear Hibernate, lo que cambia es que en lugar de indicar en el mapping el atributo resource debemos indicar el atributo class.

```
<mapping class="org.example.Subtarea"/>
```

Hibernate - Mapeo por anotaciones (ejemplo)

```

import javax.persistence.*;
import java.io.Serializable;
import java.sql.Timestamp;
import java.util.Set;

@Entity
@Table(name="Tareas")
public class Tarea implements Serializable {

    private int id;
    private String nombre;
    private String descripcion;
    private Timestamp fecha;
    private String estado;
    private Set<Subtarea> subtareas;

    public Tarea() {
    }

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    @Column(name="nombre")
    public String getNombre() { return nombre; }
    public void setNombre(String nombre) { this.nombre = nombre; }

```

```

    @Column(name="descripcion")
    public String getDescripcion() { return descripcion; }
    public void setDescripcion(String descripcion) { this.descripcion = descripcion; }

    @Column(name="fecha")
    public Timestamp getFecha() { return fecha; }
    public void setFecha(Timestamp fecha) { this.fecha = fecha; }

    @Column(name="estado")
    public String getEstado() { return estado; }
    public void setEstado(String estado) { this.estado = estado; }

    @OneToMany(mappedBy="tarea")
    public Set<Subtarea> getSubtareas() { return subtareas; }
    public void setSubtareas(Set<Subtarea> subtareas) { this.subtareas = subtareas; }

    @Override
    public String toString() {
        return "Tarea{" +
            "id=" + id +
            ", nombre='" + nombre + '\'' +
            ", descripcion='" + descripcion + '\'' +
            ", fecha=" + fecha +
            ", estado='" + estado +
            "'";
    }
}

```

Hibernate - Mapeo por anotaciones (ejemplo)

```

package org.example.model;

import javax.persistence.*;
import java.io.Serializable;

@Entity
@Table(name = "subtarea")
public class Subtarea implements Serializable {

    private int id;
    private int idTarea;
    private String nombre;
    private Tarea tarea;

    public Subtarea() {
    }

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public int getId() { return id; }

    public void setId(int id) { this.id = id; }

    @Column(name="nombre")
    public String getNombre() { return nombre; }

    public void setNombre(String nombre) { this.nombre = nombre; }

```

```

    public int getIdTarea() {
        return idTarea;
    }

    public void setIdTarea(int idTarea) {
        this.idTarea = idTarea;
    }

    @ManyToOne()
    @JoinColumn(name = "idTarea", insertable = false, updatable = false)
    public Tarea getTarea() { return tarea; }

    public void setTarea(Tarea tarea) { this.tarea = tarea; }

    @Override
    public String toString() {
        return "Subtarea{" +
            "id=" + id +
            ", idTarea=" + idTarea +
            ", nombre=" + nombre + '\n' +
            // "tarea=" + tarea +
            '}';
    }
}

```


Hibernate - Anotaciones

- ❖ **@Entity** : la anotación declara esta clase como una clase persistente. Al declarar una clase Javabeen como una clase de mapeo de tabla de base de datos de entidad, es mejor implementar la serialización. En este momento, de forma predeterminada, todos los atributos de clase son campos persistentes asignados a la tabla de datos.
- ❖ **@Table**(name = "XXX") : es una anotación de nivel de clase, definida en @Entity, para mapear el nombre de la tabla, directorio y esquema del bean de entidad.
- ❖ **@Column**(name = "XXX") : para indicar el nombre de la columna en base de datos.
- ❖ **@Id** : es el identificador único en la clase (es decir, equivalente a la clave primaria de la tabla de datos).
- ❖ **@GeneratedValue** : define la estrategia de generación de identificador.
- ❖ **@Transient** : estos campos y atributos se ignorarán y no se mantendrán en la base de datos

Hibernate - Anotaciones

- ❖ **@Temporal** (**TemporalType.TIMESTAMP**) : los campos de tipo Timestamp deben llevar esta anotación.
- ❖ **@ManyToOne**: para establecer asociaciones de muchos a uno entre entidades.
- ❖ **@JoinColumn**: para establecer el nexo de unión al establecer una anotación @ManyToOne.
- ❖ **@OneToMany**: para establecer asociaciones de uno a muchos entre entidades. En el atributo mappedBy se indica el atributo del mapeo.
- ❖ **@ManyToMany**: para establecer asociaciones de muchos a muchos entre entidades.
- ❖ **@JoinTable**: para establecer el nexo de unión al establecer una anotación @ManyToMany.

Hibernate - Ejercicio

Crear aplicación (mediante anotaciones) que permita realizar las siguientes operaciones sobre la base de datos Northwind:

- Operaciones CRUD de regiones (Region)
- Operaciones CRUD de territorios (Territories)
- Listar todos los territorios, mostrando toda la información de su región

Hibernate - Conversiones

Al codificar un documento de configuración ORM, debemos asignar los tipos de datos Java de nuestras entidades a tipos de datos Hibernate.

Los tipos declarados y utilizados en los archivos de asignación no son tipos de datos Java; tampoco son tipos de base de datos SQL.

Estos tipos se denominan tipos de asignación Hibernate, que pueden traducirse de Java a tipos de datos SQL y viceversa.

En la siguiente diapositiva podemos ver las conversiones habituales.

Nota: Recordad que los tipos de Java tampoco son idénticos a los de las bases de datos, y debemos respetar las equivalencias de los mismos también entre nuestras entidades y tablas.

Hibernate - Conversiones

Mapping type	Java type
integer	int or java.lang.Integer
long	long or java.lang.Long
short	short or java.lang.Short
float	float or java.lang.Float
double	double or java.lang.Double
big_decimal	java.math.BigDecimal
character	java.lang.String
string	java.lang.String
byte	byte or java.lang.Byte
boolean	boolean or java.lang.Boolean
yes/no	boolean or java.lang.Boolean
true/false	boolean or java.lang.Boolean

Mapping type	Java type
date	java.util.Date or java.sql.Date
time	java.util.Date or java.sql.Time
timestamp	java.util.Date or java.sql.Timestamp
calendar	java.util.Calendar
calendar_date	java.util.Calendar

Mapping type	Java type
binary	byte[]
text	java.lang.String
serializable	any Java class that implements java.io.Serializable
clob	java.sql.Clob
blob	java.sql.Blob

Hibernate - Conversiones Java y SQL Server

SQL data type	Java data type
Bit	boolean
Tinyint	short
Smallint	short
Int	int
Real	float
Bigint	long
float	double
nchar(n)	String
nvarchar(n)	String
binary(n)	byte[]

SQL data type	Java data type
varbinary(n)	byte[]
nvarchar(max)	String
varbinary(max)	byte[]
uniqueidentifier	String
char(n)	String
	Only UTF8 Strings supported
varchar(n)	String
	Only UTF8 Strings supported
varchar(max)	String
	Only UTF8 Strings supported
date	java.sql.date
numeric	java.math.BigDecimal
decimal	java.math.BigDecimal
money	java.math.BigDecimal

SQL data type	Java data type
smallmoney	java.math.BigDecimal
smalldatetime	java.sql.timestamp
datetime	java.sql.timestamp
datetime2	java.sql.timestamp

BIBLIOGRAFÍA

- ❖ Acceso a Datos, 2ª edición. Alicia Ramos Martín, María Jesús Ramos Martín. Garceta Grupo Editorial.
- ❖ Acceso a Datos. José Eduardo Córcoles y Francisco Montero Simarro. RA-MA Editorial.
- ❖ Curso de Hibernate con Java. Antonio J. Martín, Ramón Egidio.