

Herramientas de mapeo objeto-relacional

Índice

Herramientas de mapeo objeto-relacional.....	1
1. Concepto de Mapeo Objeto-Relacional (ORM, Object-Relational Mapping)	2
2. Características de las Herramientas ORM. Herramientas ORM más utilizadas	2
3. Instalación de una Herramienta ORM: Configuración de Hibernate	7
4. Clases Persistentes.....	11
5. Sesiones; Estados de un Objeto	11
6. Carga, Almacenamiento y Modificación de Objetos. Guardar un Objeto.....	12
7. Gestión de Transacciones	13
8. Estructura de un Fichero de Mapeo: Elementos, Propiedades. Mapeo Basado en Anotaciones.....	14
Webgrafía.....	19
Anexo I - Anotaciones	20

1. Concepto de Mapeo Objeto-Relacional (ORM, Object-Relational Mapping)

El mapeo objeto-relacional (ORM) es una técnica que facilita la conversión de datos entre los sistemas de tipos de lenguajes de programación orientados a objetos (como Java) y las bases de datos relacionales. ORM permite que los desarrolladores trabajen con datos en forma de objetos en lugar de escribir consultas SQL tradicionales, simplificando el acceso y manipulación de datos en bases de datos como Oracle.

Este tipo de mapeo es característico de la programación orientada a objetos, y los lenguajes más conocidos que usan este tipo de herramienta son Java, PHP, Python, Ruby y .Net, ya que son lenguajes orientados a objetos.

Esta herramienta será la encargada de coger todos los atributos del objeto que queramos persistir en base de datos, identificará de qué tipo de atributo se trata y lo guardará en su correspondiente columna en la tabla que esté indicada en este mapeo.

2. Características de las Herramientas ORM. Herramientas ORM más utilizadas

Las principales **características** de las herramientas ORM son:

- **Seguridad:** es una de las características más importantes, ya que permite proteger los datos y mantener la privacidad. El acceso a esos datos será a través de la herramienta ORM y será impenetrable de otro modo.
- **Agilidad y rapidez:** ayuda a gestionar automáticamente las transacciones entre el objeto y la base de datos sin tener que realizar ningún tipo de operación con la base de datos. Agiliza la construcción del código porque nos evita tener que realizar consultas SQL. Los programadores no necesitan tener nociones de SQL para poder gestionar una herramienta ORM.
- **Abstracción:** es una metodología que permite crear estructuras en la base de datos que nos permitirán seleccionar los datos necesarios sin tener la necesidad de conocer los detalles de esa información.
- **Independiente:** se trata de una aplicación independiente de la base de datos. En caso de migrar a otra base de datos, es bastante bueno tener ORM implementado en el proyecto.
- **Menos restricción de datos:** los cambios en base de datos se pueden realizar mediante la herramienta ORM, permitiendo ser menos restrictivo a la hora de tratar los datos si lo comparamos con JDBC.
- **Robustez:** la conexión con la base de datos se convierte en mucho más robusta, ya que tenemos mucho menos código de por medio. A través del ORM, podremos realizar todas las configuraciones necesarias para mapear los objetos Java. Además, gracias a ello, nuestra aplicación será mucho más segura.

Viendo las características y los beneficios de un ORM, podemos destacar las principales **ventajas**:

1. Se encarga de realizar todas las operaciones con la base de datos, por tanto, no es necesario codificar las sentencias SQL.
1. Generar automáticamente el código SQL usando un mapeo objeto-relacional, el cual se especifica en un documento xml o en anotaciones.
2. Las consultas a través de ORM se pueden escribir independientemente de la base de datos que se esté utilizando, lo que proporciona mucha flexibilidad al codificador. Esta es una de las mayores ventajas que ofrecen los ORM.
3. Los ORM están disponibles para cualquier lenguaje orientado a objetos, por lo que no solo son específicos de un idioma.
4. Permitir crear, modificar, recuperar y eliminar objetos persistentes.

¿Todo son ventajas? ¿Ves algún inconveniente?

... Las aplicaciones son más lentas: todas las consultas que se hagan sobre la base de datos, el sistema primero deberá transformarlas al lenguaje propio de la herramienta, luego leer los registros y por último crear los objetos.

Según el lenguaje de programación, utilizaremos un **ORM** u otro. Los más conocidos son:

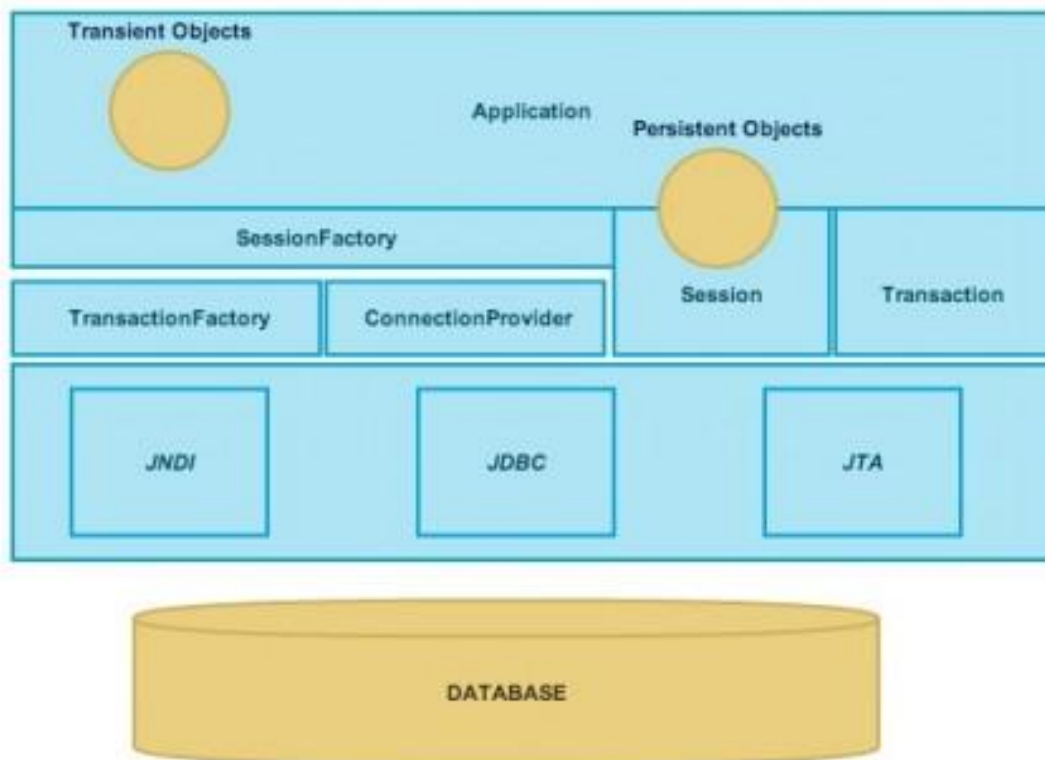
- Para lenguaje **Java** tenemos:
 - **Hibernate**: es el más conocido, y es de *software* libre. Se caracteriza por un mapeo a base de XML que ayuda a convertir los datos de los objetos Java definidos a las tablas y atributos correspondientes en base de datos. También es conocido por tener un lenguaje propio de generación de consultas de base de datos.
 - **iBatis**: también es un *framework* de código abierto, menos conocido que el anterior. Está basado en Apache Software Foundation y, como Hibernate, utiliza ficheros XML para definir la configuración que convertirá los datos de los objetos en sentencias SQL para hacer más sencilla la conexión con la base de datos.
 - **EclipseLink**: Implementación de JPA, desarrollada por Oracle.
- Para **Python**:
 - **Django**: es una herramienta de código abierto para el desarrollo de aplicaciones que sigan con el modelo MVC (modelo-vista-controlador). Es una herramienta ORM muy conocida y utilizada en este lenguaje.
 - **Storm**: otro ejemplo de herramienta ORM para Python que utiliza lenguaje SQL.
- Para **PHP**:
 - **Laravel**: es uno de los *frameworks* más conocidos para el desarrollo de aplicaciones PHP. Es gratuito.
 - **Yii**: otro ejemplo de *framework* para el desarrollo PHP, basado en el modelo típico de la programación orientada a objetos, el modelo MVC. Es uno de los *frameworks* más utilizado por ser muy rápido.

Hibernate

Centrándonos en el ORM con el que trabajaremos en este tema, **Hibernate** es un framework ORM para Java:

- Mapeo de clases Java a tablas y viceversa.
- Gestiona transacciones y garantiza consistencia de los datos.
- Gestiona los recursos de BD de forma eficiente.
- Proporciona una capa de abstracción entre la aplicación y la BD.
- Mapeos para joins, colecciones, herencia...
- HQL lenguaje similar a SQL pero orientado a objetos: soporta herencia, polimorfismo y asociación.

Hibernate - Arquitectura:



- **SessionFactory** (`org.hibernate.SessionFactory`): Factoría para crear sesiones para una BD.
- **Session** (`org.hibernate.Session`): objeto envoltorio para una conexión JDBC. Es el objeto que permite a la aplicación comunicarse con la BD.
- **Persistent objects: POJOs** que se hacen persistentes o viceversa en una sesión. Veremos más adelante qué es un POJO.
- **Transient objects**: objetos instanciados en la aplicación, pero no asociados a una sesión: no se hacen persistentes o no se recuperan a partir de tablas en una sesión.

- **Transaction (org.hibernate.Transaction):** Gestiona transacciones aislando de JDBC.
- **ConnectionProvider (org.hibernate.connection.ConnectionProvider):** Factoría para conexiones JDBC. Gestiona DataSource y DriverManager de forma transparente.
- **SessionFactory (org.hibernate.SessionFactory):** Factoría para instancias de transacción.

Hibernate – otros conceptos

Hibernate – JPA

JPA es el estándar de persistencia en Java y para aunar más aún a JPA, **Hibernate implementa la especificación de JPA**. Así que ahora podemos usar Hibernate siguiendo un estándar por lo que podríamos cambiar de Hibernate a otra implementación sin problemas.

JPA es una abstracción sobre **JDBC** que nos permite realizar la correlación entre el sistema orientado a objetos de Java y el sistema relacional de la base de datos de forma sencilla, ya que realiza por nosotros toda la conversión entre nuestros objetos y las tablas de una base de datos. Todas las clases de ésta API están en el paquete **javax.persistence**.

¿Y qué tiene que ver yakarta con javax?

Jakarta y javax están relacionados debido a una transición histórica en el ecosistema Java.

- Java EE (Java Enterprise Edition) fue una plataforma de desarrollo de aplicaciones empresariales para Java, inicialmente gestionada por Sun Microsystems y luego por Oracle cuando adquirió Sun en 2010.
- Los paquetes y bibliotecas de Java EE usaban el prefijo javax (como javax.servlet, javax.persistence, etc.) para estandarizar API críticas para aplicaciones web, EJB, etc.

Transición a Eclipse Foundation: Nacimiento de Jakarta EE

1. **Transferencia de Java EE a Eclipse Foundation:** En 2017, Oracle decidió ceder el control de Java EE a la **Eclipse Foundation**, una organización de software de código abierto. Esto marcó el inicio de lo que hoy conocemos como **Jakarta EE**.
2. **Restricciones de marca sobre javax:** Aunque Oracle permitió que la Eclipse Foundation continuara el desarrollo de la plataforma, impuso una restricción legal: el paquete **javax** no podía modificarse en futuras versiones de la plataforma bajo la nueva administración.
3. **Cambio a jakarta:** Debido a esta restricción, la Eclipse Foundation comenzó a usar el prefijo **jakarta** en lugar de javax en nuevas versiones de la plataforma, así los paquetes y clases de **Jakarta EE 9** (y versiones posteriores) pasaron a jakarta.* en lugar de javax.*.

Este cambio implica que en las nuevas versiones de Jakarta EE (es decir, Java EE renombrado y gestionado por Eclipse):

- Las bibliotecas y API ahora están bajo el prefijo **jakarta** en lugar de javax.

Entonces, Java Persistence API es una API de persistencia creada originalmente para Java EE. Se basa en el uso de objetos **POJO** y se utiliza para el mapeo de datos en modelos relacionales. Para marcar los diferentes elementos, se utilizan **decoradores o anotaciones o ficheros de mapeos**.

Un **POJO** (Plain Old Java Object, por sus siglas en inglés) es un objeto simple de Java que no depende de ninguna clase, framework o biblioteca externa. Este tipo de objeto se utiliza principalmente para representar datos de forma clara y estructurada sin lógica adicional, restricciones o dependencias.

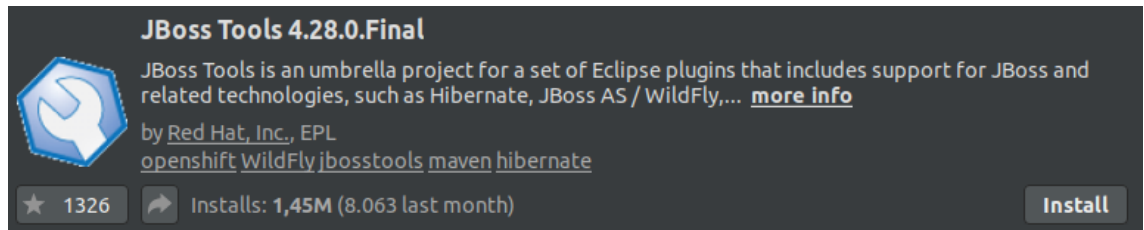
Características de un POJO

1. **Simple y sin dependencias:** Los POJO no implementan interfaces específicas ni heredan de clases particulares. Son objetos puros, independientes de cualquier marco o tecnología.
2. **Atributos privados con getters y setters:** Tienen variables de instancia (atributos) con acceso privado y métodos públicos para acceder a estos valores (getters) y modificarlos (setters).
3. **Constructores:** Usualmente tienen un constructor sin parámetros (constructor por defecto) y, en ocasiones, otros constructores para inicializar atributos.
4. **Sin métodos especiales o anotaciones requeridas:** No utilizan anotaciones o comportamientos específicos de bibliotecas externas.

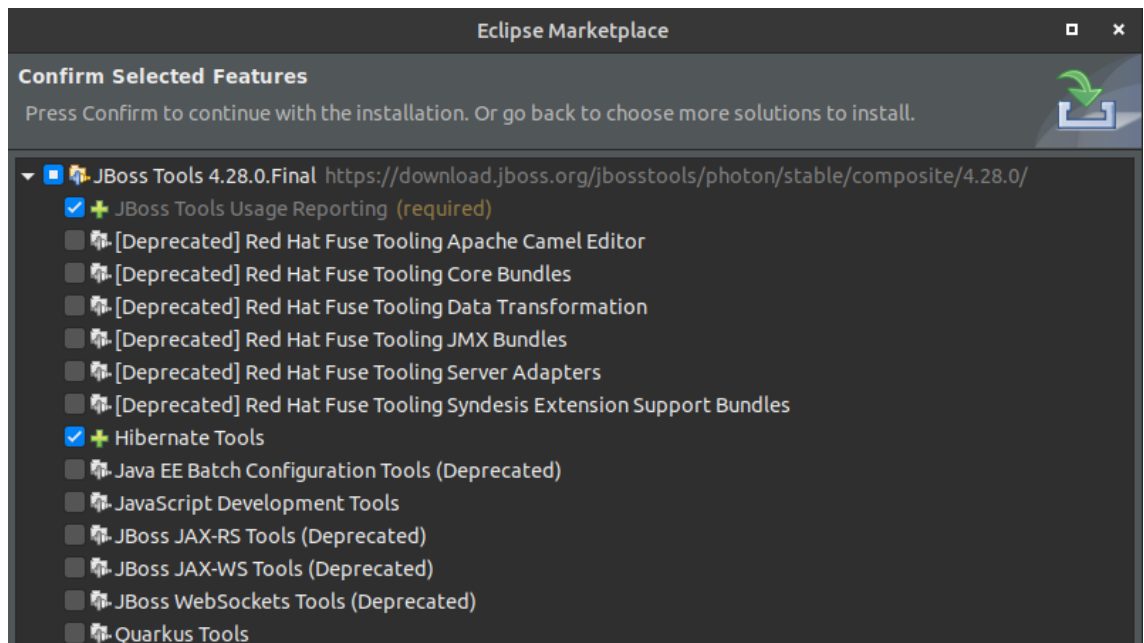
3. Instalación de una Herramienta ORM: Configuración de Hibernate

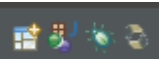
Instalación Jboss Hibernate Tools en eclipse:

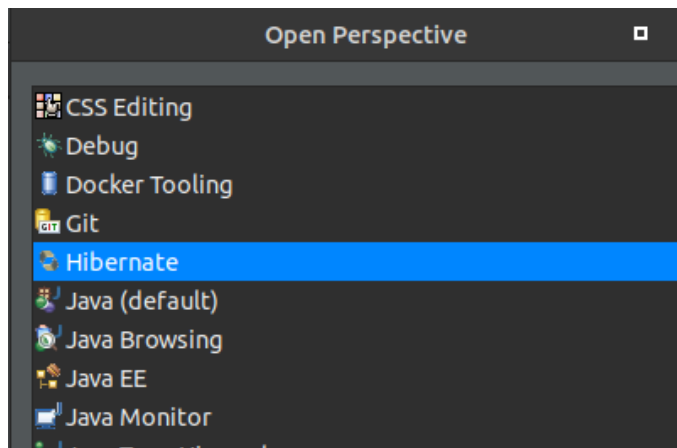
1. Ir a Help -> Eclipse Marketplace
2. Buscar Jboss Tools
3. Click en el enlace Hibernate de Jboss Tools



4. En la siguiente página seleccionamos Hibernate Tools y pulsamos Confirmar. Aceptamos los términos de licencia y pulsamos finalizar.



5. Comprobamos que se ha instalado -> Window -> Open Perspective -> Other -> Hibernate. Está también disponible la perspectiva , es el primero de los que hay arriba a la derecha. Debería aparecer una perspectiva de Hibernate:



¡Ya tenemos Eclipse preparado!

Proyecto Maven

Para crear en un proyecto Java con Hibernate, usaremos **Maven** y agregaremos las dependencias necesarias en el archivo **pom.xml** para usar hibernate,jpa con Oracle

```
<dependencies>
```

```
    <dependency>
```

```
        <groupId>org.junit.jupiter</groupId>
```

```
        <artifactId>junit-jupiter-api</artifactId>
```

```
        <scope>test</scope>
```

```
    </dependency>
```

```
    <!-- Optionally: parameterized tests support -->
```

```
    <dependency>
```

```
        <groupId>org.junit.jupiter</groupId>
```

```
        <artifactId>junit-jupiter-params</artifactId>
```

```
        <scope>test</scope>
```

```
    </dependency>
```

```
    <dependency>
```

```
        <groupId>com.oracle.database.jdbc</groupId>
```

```
        <artifactId>ojdbc11</artifactId>
```

```
        <version>23.2.0.0</version>
```

```
    </dependency>
```



```
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.1.7.Final</version>
</dependency>

<!-- Jakarta Persistence (JPA 3.0) -->
<dependency>
    <groupId>jakarta.persistence</groupId>
    <artifactId>jakarta.persistence-api</artifactId>
    <version>3.0.0</version>
</dependency>
</dependencies>
```

Configuración en hibernate.cfg.xml

Este archivo define la conexión y propiedades de Hibernate. Colócalo en **src/main/resources**, si no existe la ruta en tu proyecto créala primero.

Para crear el fichero pulsamos: File -> New -> Other -> Hibernate Configuration File

En la siguiente pantalla rellenamos todos los datos de acceso a BD y pulsamos finish. Fíjate que **no esté marcada la casilla “Create a console configuration”** antes de pulsar finish.

Hibernate Configuration File (cfg.xml)

This wizard creates a new configuration file to use with Hibernate.

Container: /MiProyecto/src/main/resources

File name: hibernate.cfg.xml

Hibernate version: 6.5

Session factory name:

[Get values from Connection](#)

Database dialect: Elige una opción de Oracle para que se rellene el combo de abajo, después borra esta información

Driver class: oracle.jdbc.driver.OracleDriver

Connection URL: jdbc:oracle:thin:@localhost:1521:xe

Default Schema:

Default Catalog:

Username: C##DAM

Password: DAM

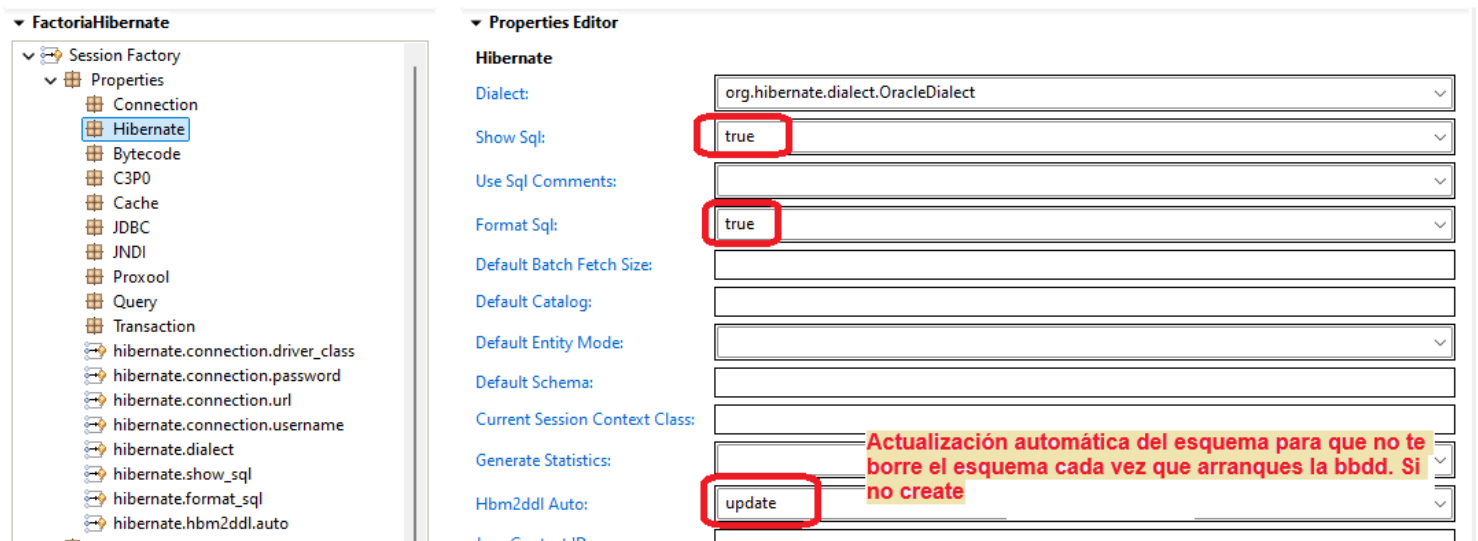
☐ Create a console configuration

< Back Next > Finish Cancel

El fichero de configuración se genera automáticamente, podemos explorarlo en modo gráfico a través de la pestaña *Session Factory* o ver el código xml en la pestaña *Source*.

Con el modo gráfico podemos aprovechar para configurar unos parámetros que son útiles:

Hibernate Configuration 3.0 XML Editor



Con el modo Source, al final del fichero añadimos el mapeo, en este caso de ejemplo con anotaciones será el mapeo, pero sería igual para el fichero:

```
<!-- Mapeo de las clases paquete.Clase-->
```

```
<mapping class="clases.Departamento" />
```

4. Clases Persistentes

Una **clase persistente** es aquella que se mapea a una tabla de base de datos y cuyas instancias se pueden almacenar y recuperar de la base de datos. En Hibernate, las clases persistentes deben:

- Tener un **constructor sin parámetros**.
- Definir un **atributo clave primaria** (@Id en anotaciones).
- Ser gestionadas mediante una sesión (Session).

5. Sesiones; Estados de un Objeto

Hibernate gestiona tres estados de objeto:

1. **Transitorio**: Objeto que se ha instanciado utilizando el operador **new** pero no guardado en la base de datos y no está asociado a una **Session** de Hibernate.
2. **Persistente**: una instancia persistente tiene una representación en la base de datos y un identificador. Puede haber sido guardado (save) o cargado (load), sin embargo, por definición, se encuentra en el ámbito de una Session. Hibernate detectará cualquier cambio realizado a un objeto en estado persistente y sincronizará el estado con la base de datos cuando se complete la transacción.
3. **Separado (detached)**: un objeto que se ha hecho persistente, pero su Session ha sido cerrada. La referencia al objeto todavía es válida, por

supuesto, y la instancia desvinculada (detached) podría incluso ser modificada en este estado. Una instancia detached puede ser re-vinculada (attached) a una nueva Session más tarde, haciéndola persistente de nuevo (con todas las modificaciones). Este aspecto habilita un modelo de programación para transacciones largas que requieren tiempo-para-pensar por parte del usuario.

El objeto Session de Hibernate se usa para interactuar con la base de datos, administrar estos estados y realizar operaciones CRUD.

6. Carga, Almacenamiento y Modificación de Objetos. Guardar un Objeto.

Haciendo los objetos persistentes:

- Las instancias recién instanciadas son **transient**.
- Podemos hacer una instancia persistente asociándola con una sesión e invocando el método **save** (deprecated), actualmente **persist()**.
- También podemos hacer un objeto persistente actualizándolo, borrándolo o incluso podemos delegar en Hibernate si la operación necesaria es guardar o actualizar. El método **update()** está deprecated, actualmente **merge()**.

Recuperando un objeto:

- Los métodos **load()** de Session le proporcionan una forma de recuperar una instancia persistente si ya conoce su identificador. **load()** toma un objeto clase y carga el estado dentro de una instancia recién instanciada de esa clase, en un estado persistente. El método **load()** está deprecated, actualmente **getReference()**.

```
Cat fritz = (Cat) sess.load(Cat.class, generatedId);
```

load() lanzará una excepción si no hay una fila correspondiente en la base de datos

- Si no tienes la certeza de que existe una fila correspondiente, debe utilizar el método **get()**, que llama a la base de datos inmediatamente y devuelve nulo si no existe una fila correspondiente.

```
Cat cat = (Cat) sess.get(Cat.class, id);  
  
if (cat==null) {  
    cat = new Cat();  
    sess.persist(cat, id);  
}  
  
return cat;
```

7. Gestión de Transacciones

La **gestión de transacciones** asegura que las operaciones se ejecuten de manera consistente. Hibernate permite manejar transacciones con Transaction.

Veamos algo de información de las clases que intervienen:

- **SessionFactory**: es la factoría que permite crear objetos sesión. Es un objeto costoso de crear (thread-safe), pensado para compartir por todos los hilos de la aplicación. Se debe crear una sola vez al arrancar la aplicación.
 - La **seguridad en hilos** es un concepto de [programación](#) aplicable en el contexto de los [programas multihilos](#). Una pieza de código es segura en cuanto a los hilos si funciona correctamente durante la ejecución simultánea de múltiples hilos. En particular, debe satisfacer la necesidad de que múltiples [hilos](#) accedan a los mismos datos compartidos, y la necesidad de que una pieza compartida de datos sea accedida por solo un hilo en un momento dado.
- **Session**: es el objeto que permite comunicar con la BD. No es costoso de crear, debe usarse una sola vez (para una petición o transacción) y descartarse. Un objeto sesión no obtiene una conexión JDBC hasta que no sea necesario, hasta que no se use.
- Hibernate deshabilita el modo **auto-commit**, o espera que el servidor de aplicaciones lo deshabilite inmediatamente. Auto-commit tiene más sentido para trabajo ad-hoc en consola SQL.
 - El comportamiento auto-commit para leer datos se debe evitar, ya que hay muy poca probabilidad de que las transacciones pequeñas funcionen mejor que una unidad de trabajo definida claramente. La última es mucho más sostenible y extensible.
- **“Ninguna comunicación con la base de datos puede darse fuera de una transacción de la base de datos”**
- Siempre usar límites de transacción claros, incluso para las operaciones de sólo lectura. Dependiendo de las capacidades de la base de datos, esto podría requerirse o no, pero no habrá ningún problema si siempre se marca explícitamente la transacción. Con seguridad, una transacción única de base de datos va a funcionar mejor que muchas transacciones pequeñas, inclusive para leer datos.

Esta es la forma en la que tendremos para trabajar con sesiones y transacciones.

Session sess = factory.openSession(); //No consume recursos de BD hasta que no es necesario

Transaction tx = null;

try {

tx = sess.beginTransaction();

// realizamos operaciones

...

tx.commit(); //Se hacen efectivos los cambios en BD

}

catch (RuntimeException e) {

if (tx != null) tx.rollback();

throw e; // o visualizamos mensaje de error

}

finally {

sess.close();

}

8. Estructura de un Fichero de Mapeo: Elementos, Propiedades. Mapeo Basado en Anotaciones.

Hibernate permite dos métodos de mapeo:

1. **Mapeo mediante XML:** Configuración en XML, donde se especifican las clases, tablas, y relaciones.
2. **Mapeo basado en anotaciones.**

El mapeo entre POJO's y tablas de BD se puede definir mediante:

- Archivos xml .hbm.xml
- En el POJO mediante anotaciones JPA (**más simple**).

El mapeo es “centrado en Java”: declaramos clases para hacerlas persistentes en tablas (no viceversa). Los principales conceptos de mapeo son:

- Entidades
- Identificadores
- Propiedades
- Componentes: son para almacenar objetos relacionados en una única tabla de BD. **No los vemos**
- Asociaciones

Entidad

- Una entidad representa una clase Java (POJO) que vamos a hacer persistente.
- Utilizamos la anotación **@Entity** para marcar una clase como persistente.
- **@Table** nos permite definir en que tabla se hará persistente y restricciones de unicidad.

```
@Table(name="TBL_FLIGHT",
```

```
uniqueConstraints=
```

```
@UniqueConstraint(columnNames={"comp_prefix", "flight_number"}) ) )
```

- El mapeo equivalente en xml se realiza con el tag **class**

```
<class name="com.hibernate.modelo.Empleado"
table="EMPLEADO"></class>
```

- El atributo **name** es el nombre cualificado completo de la clase.

Identificadores

- El identificador declara el atributo que representa la **clave primaria de la tabla**.
- Se define con la anotación **@id**.
- Para indicar una clave primaria autoincremental utilizamos **@GeneratedValue** para indicar que es autogenerado y **GenerationType.IDENTITY** strategy se usa para mapear el valor autogenerado "id" y que pueda ser recuperado en el POJO por java.
- Existen otros generadores de clave según el gestor de BD que usemos, ver apartado “5.1.2.2. Identifier generator” de <http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/ch05.html>
- En xml utilizamos el tag **<id>**

```
<id name="id" type="int"> (El nombre del atributo)
```

<column name="ID" /> (opcional: nombre de la columna de la clave primaria)

<generator class="identity" /> (clave autogenerada)

</id>

- Una columna de identidad en una tabla de SQL Server o MySQL actúa como una clave principal o única. Contiene un entero que se incrementa automáticamente en uno cada vez que se inserta un nuevo registro en la tabla.

Propiedades

Las propiedades definen qué atributos del POJO se harán persistentes.

- **@Transient** indica que la propiedad no se hará persistente. Uno de los modificadores que podemos aplicar a una **variable Java es el transient**. Las **variables Java transient** sirven para remarcar el carácter temporal o transitorio de dicha variable, es decir, que no siempre se tendrá acceso al valor de la variable.
- **@Basic** indica que la propiedad se hará persistente. Se puede omitir. Una propiedad sin anotación se hará persistente igualmente. También sirve para indicar **fetch lazy**, pero esto es avanzado y no lo veremos.

Hibernate:

- **@Enumerated** la utilizaremos para indicar que un tipo enumerado se almacene como String. Si no aparece la anotación se almacena como ordinal:

`@Enumerated(Enumtype.String)`

- **@Temporal** para definir la precisión con la que almacenar datos temporales en BD (DATE, TIME o TIMESTAMP).
- **@Lob** para indicar que se haga persistente en Clob si el tipo es java.sql.Clob, Character[], char[], java.lang.String o en Blob si el tipo es java.sql.Blob, Byte[], byte[] y Serializable.
- **Los atributos estáticos no se hacen persistentes.**
- Podemos especificar manualmente los tipos de las propiedades con la anotación **@Type**, y el tipo puede ser:
 - Un tipo básico hibernate: integer, string, character, date, timestamp, float, binary, serializable, object, blob etc.
 - Un tipo básico Java: int, float, char, java.lang.String, java.util.Date, java.lang.Integer, java.sql.Clob, etc.
 - El nombre de una clase Java Serializable.
 - El nombre de una clase definida por el usuario (tipo Personalizado).

- Si no se especifica un tipo, Hibernate intenta automáticamente deducir el tipo hibernate asociado, suele ser lo habitual.
- **@column** nos permite definir con qué columnas de la tabla mapear las propiedades, podemos utilizar esta anotación sobre propiedades:
 - No anotadas.
 - Anotadas con @Basic, @Temporal, @Lob.
 - Los atributos más comunes en la anotación **@column** son:
 - **name** (opcional): El nombre de la columna (default nombre de la propiedad).
 - **unique** (opcional): establecer restricción (default false).
 - **nullable** (opcional): establecer como nullable (default true).
 - **insertable** (opcional): la columna será parte de la sentencia de inserción (default true).
 - **updatable** (opcional): la columna será parte de la sentencia de update (default true).
 - **length** (opcional): longitud (default 255).
 - **precision** (opcional): precision decimal (default 0).
 - **scale** (opcional): escala decimal (default 0).

`@Column(name="ID", nullable=false, unique=true, length=11)`

- **@formula** nos permite definir un cálculo al obtener un valor de una propiedad. Es una propiedad de lectura.
 - `@formula ("obj_length * obj_height * obj_width")`
- **Si el nombre de la columna no coincide con el de la propiedad hay que definirlo para que no de error.**

- En xml las propiedades se definen diferente con el tag **property**:

```
<property>
```

```
    name="propertyName"
```

```
    column="column_name"
```

```
    type="typename"
```

```
    update="true|false"
```

```
    insert="true|false"
```

```
    formula="SQL expression"
```

```
    unique="true|false"
```

```
    not-null="true|false"
```

```
    length="L"
```

```
    precision="P"
```

```
    scale="S"
```

```
/>
```

```
<property name="totalPrice"
```

```
    formula="( SELECT SUM (li.quantity*p.price) FROM LineItem li, Product p
```

```
        WHERE li.productId = p.productId
```

```
        AND li.customerId = customerId
```

```
        AND li.orderNumber = orderNumber )"/>
```

Webgrafía

1. Documentación Hibernate, <http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/ch01.html>
2. Hibernate Tools, https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Developer_Studio/7.0/html/Hibernate_Tools_Reference_Guide/introduction.html
3. Tutorial on line, <http://www.cursohibernate.es/>
4. Más Ejercicios en http://cursohibernate.es/doku.php?id=ejercicios:00_start

Anexo I - Anotaciones

CLASE

- **@Entity**: indica que la clase decorada es una entidad.
- **@MappedSuperClass**: aplicado sobre una clase, indica que se mapeará como cualquier otra clase, pero que se aplicará únicamente a sus subclases, puesto que esta entidad no tiene una tabla asociada.
- **@Table**: especifica el nombre de la tabla relacionada con la entidad. Admite el parámetro:
 - name: nombre de la tabla

PROPIEDAD

- **@Column**: indica el nombre de la columna en base de datos. Admite el parámetro:
 - name: especifica el nombre de la columna
- **@Enumerated**: indica que los valores de la propiedad van a estar dentro del rango de un objeto enumerador. Admite el parámetro:
 - value: indica el tipo valor que se va a utilizar en la persistencia en Base de Datos. Puede utilizarse el enumerador EnumType.
- **@Id**: aplicado sobre una propiedad, indica que es la clave primaria de una entidad. La propiedad a la que hace referencia puede ser de los siguientes tipos: cualquier tipo primitivo, String, java.util.Date, java.sql.Date, java.math.BigDecimal, java.math.BigInteger
- **@GeneratedValue**: especifica la estrategia de generación de la clave primaria. Admite el parámetro:
 - strategy: indica la estrategia a seguir para la obtención de un nuevo identificador. Permite utilizar el enum GenerationType.
- **@Lob**: aplicado sobre una propiedad, indica que es un objeto grande (Large Object). Por ejemplo, en el caso de String, si no se especifica @Lob, se le asigna un tamaño máximo de 255 caracteres.
- **@NotEmpty**: validación de restricción, indica que la propiedad no puede tener un valor vacío.