

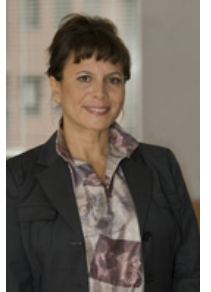
## AD04.- Mapeo objeto relacional.

### Caso práctico

En BK todo el desarrollo de software se hace siguiendo el paradigma de la Programación Orientada a Objetos. Sin embargo, la mayoría de las bases de datos con las que interaccionan sus aplicaciones son bases de datos relacionales, aunque también las hay orientadas a objetos.

**Ada** constata que parte de sus trabajadores se van a enfrentar con un dilema: ¿cómo usar objetos y clases para interactuar con bases de datos relacionales? Si en éstas sólo se pueden registrar tipos de datos simples, ¿cómo guardar ahí un objeto?

Este es el punto de partida del mapeo objeto-relacional, y los trabajadores de BK van a aprender cómo se trabaja en el sistema de almacenamiento de componentes propios de la Programación orientada a Objetos.



## 1.- Concepto de Mapeo objeto-relacional.

### Caso práctico

**María** percibe que sus aplicaciones, basadas en Programación Orientada a Objetos, no son almacenables de forma directa en los Sistemas Gestores de Bases de Datos que tienen en la empresa. ¿Cómo guardar un objeto en un sistema de almacenamiento basado en tablas?



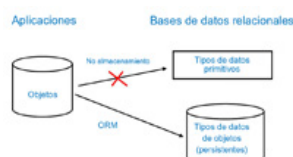
A la hora de almacenar los datos de un programa orientado a objetos en una **base de datos relacional**, surge un inconveniente debido a incompatibilidad de sistemas de **tipos de datos**. En el software orientado a objetos, la información se representa como clases y objetos. En las bases de datos relacionales, como tablas y sus restricciones. Por tanto, para almacenar la información tratada en un programa orientado a objetos en una base de datos relacional es necesaria una traducción entre ambas formas.

El mapeo objeto-relacional (ORM) soluciona este problema. Es una técnica de programación que se utiliza con el propósito de convertir datos entre el utilizado en un lenguaje de programación orientado a objetos y el utilizado en una base de datos relacional, gracias a la **persistencia**. Esto posibilita el uso en las bases de datos relacionales de las características propias de la programación orientada a objetos (básicamente **herencia** y **polimorfismo**).

**La mayoría de las aplicaciones se construyen usando técnicas de programación orientada a objetos; sin embargo, los sistemas de bases de datos más extendidos son de tipo relacional.**

Las bases de datos más extendidas son del tipo relacional y estas sólo permiten guardar tipos de datos primitivos (enteros, cadenas de texto, etc.) por lo que no se puede guardar de forma directa los objetos de la aplicación en las tablas. Por tanto, se debe convertir los valores de los objetos en valores simples que puedan ser almacenados en una base de datos (y poder recuperarlos más tarde).

El mapeo objeto-relacional surge, pues, para dar respuesta a esta problemática: traducir los objetos a formas que puedan ser almacenadas en bases de datos preservando las propiedades de los objetos y sus relaciones; estos objetos se dice entonces que son persistentes.



[\(link: AD04\\_CONT\\_R03\\_EsquemaORM.jpeg.\)](#)

El ORM se encarga, de forma automática, de convertir los **objetos** en **registros** y viceversa, simulando así tener una base de datos orientada a objetos.

## 2.- Herramientas ORM. Características y herramientas más utilizadas.

### Caso práctico

En la empresa se ponen manos a la obra para estudiar las herramientas que permitan almacenar objetos y clases en bases de datos relacionales. ¿Qué son y qué características tienen las herramientas ORM?



Ya hemos dicho que las herramientas ORM se utilizan para dar solución al problema de que, en la programación orientada a objetos, la gestión de datos se implementa usando objetos; sin embargo, en los sistemas de gestión de bases de datos SQL sólo se pueden almacenar y manipular valores escalares organizados en tablas relacionales.

Object Relational Mapping (ORM) es la herramienta que nos sirve para transformar representaciones de datos de los Sistemas de Bases de Datos Relacionales, a representaciones (Modelos) de objetos. Dado a que los RDBMS carecen de la flexibilidad para representar datos no escalares, la existencia de un ORM es fundamental para el desarrollo de sistemas de software robustos y escalables.

**En el modelo relacional, cada fila en la tabla se mapea a un objeto y cada columna a una propiedad.**

Las herramientas ORM pues, actúan como un puente que conecta las ventajas de los RDBMS con la buena representación de estos en un lenguaje Orientado a Objetos, o, dicho en otras palabras, nos lleva de la base de datos al lenguaje de programación.

### Autoevaluación

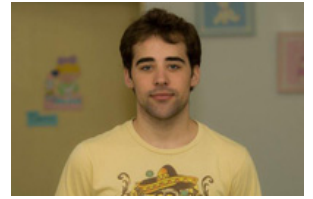
Utilizando una herramienta ORM podemos recuperar los objetos de un programa en formas compatibles con el sistema de almacenamiento de los gestores de bases de datos relacionales.

- ☐ [\(link: \)](#)  
No.
- ☐ [\(link: \)](#)  
Si.

## 2.1.- Características.

### Caso práctico

**Antonio** no sabe exactamente qué tiene que utilizar ni qué ventajas obtendrá con el uso de las herramientas de mapeo ¿Cuáles son las características de las herramientas ORM?



Las herramientas ORM facilitan el mapeo de atributos entre una base de datos relacional y el modelo de objetos de una aplicación, mediante archivos declarativos **XML o anotaciones** que permiten establecer estas relaciones. Gracias a las ORM, podemos conectar con una base de datos relacional para extraer la información contenida en objetos de programas que están almacenados. Para ello, sólo tendremos que definir la forma en la que establecer la correspondencia entre las clases y las tablas una sola vez (indicando qué propiedad se corresponde con cada columna, qué clase con cada tabla, etc.). Una vez hecho esto, podremos utilizar **POJO's** de nuestra aplicación e indicar a la ORM que los haga persistentes, consiguiendo que una sola herramienta pueda leer o escribir en la base de datos utilizando **VO's** directamente.

**Una herramienta ORM permite tomar un objeto Java y hacerlo persistente, carga el objeto de la base de datos a memoria y permite hacer consultas a las tablas de la base de datos.**

#### Ventajas de ORM.

- ✓ Ayudan a reducir el tiempo de desarrollo de software. La mayoría de las herramientas ORM disponibles, permiten la creación del modelo a través del esquema de la base de datos, es decir, el usuario crea la base de datos y la herramienta automáticamente lee el esquema de tablas y relaciones y crea un modelo ajustado.
- ✓ Abstracción de la base de datos.
- ✓ Reutilización.
- ✓ Permiten persistir objetos a través de un método `save` o `persist` y generar el SQL correspondiente.
- ✓ Permiten recuperar los objetos persistidos a través de un método `load` o `get`.
- ✓ Lenguaje propio para realizar las consultas.
- ✓ Independencia de la base de datos.
- ✓ Incentivan la **portabilidad** y **escalabilidad** de los programas de software.

#### Desventajas de ORM.

- ✓ Tiempo utilizado en el aprendizaje. Este tipo de herramientas suelen ser complejas por lo que su correcta utilización lleva un tiempo que hay que emplear en ver el funcionamiento correcto y ver todo el partido que se le puede sacar.
- ✓ Menor rendimiento (aplicaciones algo más lentas). Esto es debido a que todas las consultas que se hagan sobre la base de datos, el sistema primero deberá de transformarlas al lenguaje propio de la herramienta, luego leer los registros y por último crear los objetos.
- ✓ Sistemas complejos. Normalmente la utilidad de ORM desciende con la mayor complejidad del sistema relacional.

## 2.2.- Herramientas ORM más utilizadas.

### Caso práctico

Sabiendo ya la función de las herramientas ORM, el siguiente punto es el análisis de las ORM existentes en el mercado, junto a sus características más destacables, para decidir cuál es la que mejor se adapta a nuestras necesidades.



Entre las herramientas ORM más relevantes encontramos las siguientes:

#### Hibernate:

Hibernate es una herramienta de Mapeo objeto-relacional (ORM) 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. Utiliza archivos declarativos (XML) o anotaciones en los beans de las entidades que permiten establecer estas relaciones.

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



#### Java Persistence Api (JPA):

El Java Persistence **API** (JPA) es una especificación de Sun Microsystems para la persistencia de objetos Java a cualquier base de datos relacional. Esta API fue desarrollada para la plataforma JEE e incluida en el estándar de EJB 3.0, formando parte de la Java Specification Request JSR 220.

#### iBatis:

iBatis es un framework de persistencia desarrollado por la Apache software Foundation. Al igual que el resto de los proyectos desarrollados por la ASF, iBatis es una herramienta de código libre.

iBatis sigue el mismo esquema de uso que Hibernate; se apoya en ficheros de mapeo XML para persistir la información contenida en los objetos en un **repositorio** relacional.



En el siguiente enlace encontrarás la guía de iniciación de la herramienta Hibernate.

[Hibernate Getting Started Guide . \(link: https://docs.jboss.org/hibernate/orm/current/quickstart/html\\_single/\).](https://docs.jboss.org/hibernate/orm/current/quickstart/html_single/)

### Autoevaluación

La herramienta ORM Hibernate se caracteriza por:

- ☐ (link: )  
No ser compatible con la plataforma .Net.
- ☐ (link: )  
Ser software libre.
- (link: )

- ☐ Ser una especificación de Sun Microsystems .

### 3.- Instalación y configuración de Hibernate

## Caso práctico

Una vez que el equipo de BK ha estudiado las diferentes herramientas ORM, ya han decidido que es Hibernate la que mejor se adapta a sus requerimientos. Por ello, el paso siguiente es su instalación y configuración en los equipos para empezar a utilizarla.



La instalación de Hibernate sobre el IDE Eclipse requiere tener instalado previamente este entorno de desarrollo, junto al JDK.

Para su utilización tenemos dos opciones:

1. Visitamos la [web de Hibernate ORM \(link: https://hibernate.org/orm/\)](https://hibernate.org/orm/) y nos descargamos la última versión estable del programa en un fichero .zip. Lo descomprimos y añadimos los .jar que se encuentran en la carpeta *lib/required* a nuestro proyecto.
2. Creamos un proyecto Maven en Eclipse y en el fichero *pom.xml* añadimos las dependencias que podemos consultar en el [Repositorio Maven. \(link: https://mvnrepository.com/artifact/org.hibernate/hibernate-core\)](https://mvnrepository.com/artifact/org.hibernate/hibernate-core).

Hibernate puede configurarse y ejecutarse en la mayoría de aplicaciones Java y entornos de desarrollo. El archivo de configuración de Hibernate recibe el nombre de *Hibernate.cfg.xml* y contiene información sobre la conexión de la base de datos y otras propiedades. Al crearlo, hay que especificar la conexión a la base de datos.

En los siguientes puntos del tema veremos más detenidamente el proceso de configuración.

## 4.- Ficheros de configuración y anotaciones. Estructura y propiedades.

### Caso práctico

En el momento de empezar a trabajar, **Juan** se enfrenta a aspectos esenciales tras la instalación de la herramienta ORM: los ficheros de configuración y de mapeo para el correcto funcionamiento de Hibernate.



Para empezar a trabajar con Hibernate es necesario configurar la herramienta para que conozca qué objetos debe recuperar de la base de datos relacional y en qué lugar los hará persistir. Por tanto, el primer paso será tener una base de datos relacional con la que poder trabajar.

En nuestro caso trabajaremos con MySQL por lo que en primer lugar nos conectaremos mediante MySQL Workbench y crearemos el esquema y un usuario con privilegios suficientes sobre ese esquema.

### Para saber más

En el siguiente enlace encontrarás un documento donde se expone con más detalle aspectos de la configuración de Hibernate.

Guía de usuario de Hibernate. (link: [https://docs.jboss.org/hibernate/orm/5.4/userguide/html\\_single/Hibernate\\_User\\_Guide.html](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html) )



### Caso práctico

**Ana** ya ha instalado Hibernate sobre el IDE Eclipse y ahora surge el problema de configurar correctamente todos sus parámetros para trabajar de la forma más adecuada. Hay varias formas de trabajar con los archivos de configuración de Hibernate; habrá que decidir de qué forma se va a trabajar.



El archivo de configuración de Hibernate es el `Hibernate.cfg.xml` y contiene información sobre la conexión de base de datos, las asignaciones de recursos y otras propiedades de conexión. A continuación se muestra un ejemplo:

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- Configuración de la conexión JDBC -->
        <property name="connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost:3306/hb_student?useSSL=false&serverTimezone=UTC</property>
        <property name="connection.username">hbstudent</property>
        <property name="connection.password">hbstudent</property>
        <!-- Configuración del pool de conexiones JDBC... utilizamos el pool de test integrado -->
        <property name="connection.pool_size">1</property>
        <!-- Seleccionamos el dialecto SQL -->
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <!-- Configurar SQL para salida estándar -->
        <property name="show_sql">true</property>
        <!-- Establece el contexto de la sesión -->
        <property name="current_session_context_class">thread</property>
    </session-factory>
</hibernate-configuration>
```

Algunas de las propiedades más importantes del fichero `Hibernate.cfg.xml` son:

- ✓ `Hibernate.dialect`: Dialecto o lenguaje empleado. Por ejemplo, MySQL.
- ✓ `Hibernate.connection.driver_class`. Driver utilizado para la conexión con la base de datos.
- ✓ `Hibernate.connection.url`. Dirección de la base de datos con la que se va a conectar Hibernate.
- ✓ `Hibernate.connection.username`. Nombre del usuario que va a realizar la extracción de información. Por defecto, el nombre de usuario es root.
- ✓ `Hibernate.connection.password`. Contraseña del root o del usuario con privilegios que se conectará a la base de datos.
- ✓ `Hibernate.show_sql`. Para mostrar la herramienta. Por defecto, su valor es true.

### Autoevaluación

El archivo `Hibernate.cfg.xml` contiene información sobre:

- *(link: )*  
La conexión, recursos y otras propiedades como nombre de usuario, contraseña, etc.
- *(link: )*  
La conexión a la base de datos.
- *(link: )*  
La dirección de la base de datos y la versión del IDE utilizada.

# Caso práctico

Ya se ha configurado Hibernate y todo parece estar listo. La cuestión ahora es relacionar los objetos del programa orientado a objetos con las tablas de la base de datos relacional. Entran en juego los ficheros de mapeo y las anotaciones.

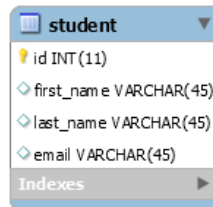


Hibernate utiliza ficheros de mapeo (en formato XML que tienen extensión `.hbm.xml`.) o anotaciones JPA para relacionar tablas con objetos Java.

Entre estas dos opciones, **se recomienda utilizar las anotaciones JPA**. JPA es una especificación estándar e Hibernate es una implementación de esa especificación JPA. Hibernate implemente todas las anotaciones JPA y el propio equipo de Hibernate recomienda utilizar anotaciones JPA como buena práctica.

Estas anotaciones se realizan directamente en los POJOs junto con el código y es ahí donde se especifica la correspondencia con las tablas de la base de datos.

Un ejemplo de mapeo simple de la tabla *student* podría ser el siguiente:



...

```
@Entity
@Table(name="student")
public class Student {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY) //La opción más usada con MySQL
    @Column(name="id")
    private int id;

    @Column(name="first_name")
    private String firstName;

    @Column(name="last_name")
    private String lastName;

    @Column(name="email")
    private String email;

    ...
```

Seguido de los constructores, getters y setters y toString del POJO.

## 5.- Mapeo de relaciones.

### Caso práctico

**María** necesita ahora aprender la verdadera utilidad de Hibernate: el mapeo de elementos, tales como relaciones, que le permita avanzar sobre el objetivo perseguido de almacenar estas características de la Programación Orientada a Objetos en las bases de datos relacionales.



#### 1. Mapeo de relaciones.

Para persistir, las relaciones usan las denominadas transacciones, ya que los cambios pueden incluir varias tablas. Una regla general para el mapeo es respetar el tipo de relación en el modelo de objetos y en el modelo relacional: así, una relación 1-1 en el modelo de objetos deberá corresponderse a una relación 1-1 en el modelo relacional.



Para mapear las relaciones, se usan directas anotaciones dependiendo de la asociación que tienen las entidades. Estas pueden ser:

- @OneToOne
- @ManyToOne
- @OneToMany
- @ManyToMany

### Para saber más

En el siguiente enlace puedes tener acceso a un tutorial en línea de las asociaciones en Hibernate, donde se detallan estos aspectos con más profundidad.

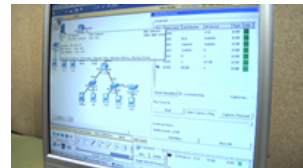
Asociaciones en Hibernate. (link:  
[https://docs.jboss.org/hibernate/orm/5.4/userguide/html\\_single/Hibernate\\_User\\_Guide.html#associations](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#associations) )

### Caso práctico

**Ana** se pregunta si los objetos que se utilizan en una aplicación, para representar entidades de una base de datos, se mantienen siempre en el mismo estado, o si por el contrario, durante todo el proceso de inicio de aplicación, acceso a la base de datos, modificación, etc., estos objetos cambian de estado o funcionalidad.



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.



La sesión está formada internamente por una cola de sentencias SQL que son necesarias ejecutar para poder sincronizar el estado de la sesión con la base de datos.

Asimismo, la **sesión** contiene una lista de objetos persistentes. Una sesión corresponde con el primer nivel de caché.

Si para realizar el acceso a datos, usamos Hibernate, la sesión nos permite definir el alcance de un contexto determinado. Para poder utilizar los mecanismos de persistencia de Hibernate se debe inicializar el entorno Hibernate y obtener un objeto `Session` utilizando la clase `SessionFactory` de Hibernate. Un objeto `Session` Hibernate representa una única unidad-de-trabajo para un almacén de datos dado y lo abre un ejemplar de `SessionFactory`. Se deben cerrar las sesiones cuando se haya completado todo el trabajo de una transacción.

Para obtener un ejemplar de `SessionFactory` realizaremos:

```
protected void setUp() throws Exception {
    final StandardServiceRegistry registry = new StandardServiceRegistryBuilder()
        .configure() // por defecto: hibernate.cfg.xml
        .build();

    try {
        sessionFactory = new MetadataSources( registry ).buildMetadata().buildSessionFactory();
    }
    catch (Exception e) {
        StandardServiceRegistryBuilder.destroy( registry );
    }
}
```

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

- ✓ Transitorio (Transient). En este estado estará un objeto recién creado que no ha sido enlazado con el gestor de persistencia.
- ✓ Persistente: En este caso el objeto está enlazado con la sesión. Todos los cambios que se realicen serán persistentes.
- ✓ Disociado (Detached): En este caso nos encontramos con un objeto persistente que sigue en memoria después de que termine la sesión. En este caso existe en Java y en la base de datos.
- ✓ Borrado (Removed): En este caso el objeto está marcado para ser borrado de la base de datos. Existe en la aplicación Java y se borrará de la base de datos al terminar la sesión.

### Autoevaluación

**Un objeto en estado transitorio:**

- *(link: )*  
El objeto no ha sido enlazado con el gestor de persistencia.
- *(link: )*  
El objeto ha sido marcado para ser borrado.
- *(link: )*  
El objeto sigue en memoria después de finalizada la sesión.

### Caso práctico

Para poder interactuar con una bases de datos, **María** debe definir en su aplicación clases y objetos que puedan representar las entidades de la base de datos con la que interacciona la aplicación. Otra opción posible es utilizar el estandar JPA.



En los capítulos previos hemos utilizado el fichero de configuración específico de Hibernate *hibernate.cfg.xml*.

Sin embargo, JPA define un proceso bootstrap que utiliza su propio archivo de configuración denominado *persistence.xml*. Este debe estar localizado en la siguiente ruta: *META-INF/persistence.xml*. A continuación se muestra un ejemplo:



```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">
  <persistence-unit name="org.hibernate.tutorial.jpa">
    ...
  </persistence-unit>
</persistence>
```

## 6.1.- Ejemplo de uso

En primer lugar obtenemos el `javax.persistence.EntityManagerFactory`:

```
protected void setUp() throws Exception {  
    sessionFactory = Persistence.createEntityManagerFactory( "org.hibernate.tutorial.jpa" );  
}
```

Para guardar las entidades utilizamos el interfaz `javax.persistence.EntityManager` en vez de `org.hibernate.Session` Como vemos existen ligeras variaciones y en vez de *save* se utiliza *persist*.

```
EntityManager entityManager = sessionFactory.createEntityManager();  
entityManager.getTransaction().begin();  
entityManager.persist( new Event( "Our very first event!", new Date() ) );  
entityManager.persist( new Event( "A follow up event", new Date() ) );  
entityManager.getTransaction().commit();  
entityManager.close();
```

Por último, para obtener una lista de las entidades vemos que el código es muy similar al utilizado con Hibernate:

```
entityManager = sessionFactory.createEntityManager();  
entityManager.getTransaction().begin();  
List<Event> result = entityManager.createQuery( "from Event", Event.class ).getResultList();  
for ( Event event : result ) {  
    System.out.println( "Event ( " + event.getDate() + " ) : " + event.getTitle() );  
}  
entityManager.getTransaction().commit();  
entityManager.close();
```



## 8.- Carga, almacenamiento y modificación de objetos.

### Caso práctico

**Juan** se está convirtiendo en un entusiasta de las herramientas de mapeo-objeto relacional. Empieza a entender y comprobar sus ventajas, sin embargo, ahora toca lo más importante: como cargar, almacenar y modificar los objetos que representan elementos de la base de datos.



Para cargar un objeto de acceso a datos en la aplicación Java, el método `load()` de la clase `Session` suministra un mecanismo para capturar una instancia persistente, si conocemos su identificador. El método `load()` acepta un objeto `Class`, y cargará el estado de una nueva instancia de esa clase, inicializada en estado persistente.

El método `load()` lanzará una excepción irrecuperable si no existe la fila de base de datos correspondiente. Si no se está seguro de que exista una fila correspondiente, debe usarse el método `get()`, el cual consulta la base de datos inmediatamente y devuelve `null` si no existe una fila correspondiente.

Existen dos métodos que se encargan de recuperar un objeto persistente por identificador: `load()` y `get()`. La diferencia entre ellos radica en cómo indican que un objeto no se encuentra en la base de datos: `get()` devuelve un nulo y `load()` lanza una excepción `ObjectNotFoundException`.

Aparte de esta diferencia, `load()` intenta devolver un objeto **proxy** siempre y cuando le sea posible (no esté en el contexto de persistencia). Con lo que es posible que la excepción sea lanzada cuando se inicialice el objeto proxy. Esto es conocido como carga perezosa.

En el caso que se obtenga un proxy, no tiene impacto sobre la base de datos (no se ejecuta ninguna consulta), hasta que no se inicializa el mismo. Muy útil cuando se obtiene una referencia de un objeto para asociarlo a otro. (No es necesario obtener el objeto). Se modifica un objeto persistente.

## 8.1.- Almacenamiento y modificando de objetos persistentes.

### Caso práctico

**Juan** empieza a comprender como se pueden cargar objetos persistente. **Juan** quiere enseñar a a **María** el mecanismo de almacenamiento y modificación de objetos persistentes, para que la aplicación que enlaza a bases de datos sea lo más completa posible.



Para almacenar objetos persistentes se proceso siguiendo los siguientes pasos:

1. Se instancia un objeto nuevo (estado transitorio).
2. Se obtiene una sesión y se comienza la transacción, inicializando el contexto de persistencia.
3. Una vez obtenida da la sesión, se llama al método `save()`, el cual introduce el objeto en el contexto de persistencia. Este método devuelve el identificador del objeto persistido.
4. Para que los cambios sean sincronizados en las bases de datos, es necesario realizar el `commit` de la transacción. Dentro del objeto sesión se llama al método `flush()`. Es posible llamarlo explícitamente. En este momento, se obtiene la conexión JDBC a la bases de datos para poder ejecutar la oportuna sentencia.
5. Finalmente, la sesión se cierra, con el objeto de liberar el contexto de persistencia, y por tanto, devolver la referencia del objeto creado al estado disociado.

Los objetos cargados, grabados, creados o consultados por las sesión pueden ser manipulados por la aplicación, y cualquier cambio a su estado de persistencia será persistido cuando se le aplique "flush" a la sesión.

No hay que invocar ningún método en particular para que las modificaciones se vuelvan persistentes. Así que la manera más sencilla y directa de actualizar el estado de un objeto es cargarlo con `load()`, y luego manipularlo directamente, mientras la sesión esté abierta.

Para borrar objetos persistentes, podemos ejecutar `Session.delete()`, que quitará el estado de un objeto de la base de datos. Por supuesto, su aplicación podría aún contener una referencia al objeto quitado. Se puede borrar objetos en cualquier orden, no se van producir violaciones de llave externa, pero sí es posible violar `constraints NOT NULL` aplicadas a la columna de llave externa.

Muchas aplicaciones necesitan capturar un objeto en una transacción, mandarlo a la capa de interfaz de usuario para su manipulación, y grabar sus cambios en una nueva transacción. Las aplicaciones que usan este tipo de estrategia en entornos de alta concurrencia, normalmente usan datos versionados para garantizar aislamiento durante la "larga" unidad de trabajo.

Hibernate soporta este modelo, proveyendo "reasociación" de entidades desprendidas usando los métodos `Session.update()` o `Session.merge()`.



### Autoevaluación

**Para sincronizar los datos con la base de datos es necesario realizar:**

- *(link: )*  
Un inicio de sesión.
- *(link: )*  
Se invoca el método `save()` .
- *(link: )*  
Se invoca al método `flush()` .

### Caso práctico

**Ana** ya ha aprendido como poder utilizar una base de datos relacional en una aplicación que utiliza clases persistentes. La pregunta que le surge ahora es ¿Cómo puedo realizar las consultas SQL con esta forma de acceder a los datos? ¿Puedo utilizar el select, insert, etc.?



Usando Hibernate, la ejecución de **consultas SQL nativas** se controla por medio de la interfaz `NativeQuery`, la cual se obtiene llamando a `Session.createNativeQuery()`. Las siguientes secciones describen cómo utilizar esta API para consultas.

El equivalente en JPA sería: `entityManager.createNativeQuery()`

Los tipos de consulta SQL más básicas para obtener una lista de escalares (valores) son los siguientes:

- ✓ `sess.createNativeQuery("SELECT * FROM Personas").list();`
- ✓ `entityManager.createNativeQuery("SELECT ID,NOMBRE, EDAD FROM PERSONAS FROM Personas" ).getResultList();`

Estas retornarán una lista de objetos arrays (`Object[]`) con valores escalares para cada columna en la tabla `PERSONAS`. Hibernate utilizará `ResultSetMetadata` para deducir el orden real y los tipos de los valores escalares retornados.

Otro tipo de consulta más compleja, es la consulta de entidades. Para obtener los objetos entidades desde una consulta sql nativa, se utiliza por medio de `addEntity()`.

- ✓ `sess.createNativeQuery("SELECT * FROM PERSONAS").addEntity(Persona.class).list;`
- ✓ `entityManager.createNativeQuery("SELECT * FROM PERSONAS", Persona.class).getResultList();`

Se especifica esta consulta:

- ✓ La cadena de consulta SQL.
- ✓ La entidad devuelta por la consulta.

Asumiendo que `Persona` es mapeado como una clase con las columnas `IDENTIFICACION`, `NOMBRE` y `FECHA DE NACIMIENTO` las consultas anteriores devolverán una Lista en donde cada elemento es una entidad `Persona`.



### Para saber más

Puedes consultar más información acerca de Native SQL Queries en el siguiente enlace:

NativeSQLQueries (link: [https://docs.jboss.org/hibernate/orm/5.4/userguide/html\\_single/Hibernate\\_User\\_Guide.html#sql](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#sql))

### Autoevaluación

¿En qué estado se encuentra un objeto persistente en memoria, una vez finalizada la sesión?

- *(link:)*  
Transitorio.
- *(link:)*  
Borrado.
- *(link:)*  
Disociado.

## 10.- Lenguajes propios de la herramienta ORM.

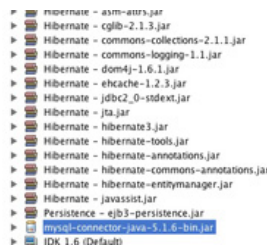
### Caso práctico

**María** conoce el lenguaje SQL, para poder realizar consultas a una base de datos relacional. Sin embargo están para poder interactuar con la base de datos desde Java, en BK está usando Hibernate. ¿Dispondrá Hibernate de un lenguaje de consulta propio? Ada le comenta que existe el **HQL**, que le va a facilitar el trabajo de consulta.



Hibernate utiliza un lenguaje de consulta potente (HQL) que se parece a SQL. Sin embargo, comparado con SQL, HQL es completamente orientado a objetos y comprende nociones como herencia, polimorfismo y asociación. Las consultas se escriben en HQL y Hibernate se encarga de convertirlas al SQL usado por la base de datos con la que estemos trabajando y ejecutarla para realizar la operación indicada.

HQL es case-insensitive, o sea que sus sentencias pueden escribirse en mayúsculas y minúsculas. Por lo tanto "SeLeCt", "seleCT", "select", y "SELECT" se entienden como la misma cosa. Lo único con lo que debemos tener cuidado es con los nombres de las clases que estamos recuperando y con sus propiedades, ahí si se distinguen mayúsculas y minúsculas. O sea, en este caso "pruebas.Hibernate.Usuario" NO ES LO MISMO que "PrueBAs.Hibernate.UsuArio".



[\(link: AD04 CONT R26 libreriaHibernate.jpg.\)](#)

Entre las características más importantes de HQL.

- ✓ Soporte completo para operaciones relacionales: HQL permite representar consultas SQL en forma de objetos. HQL usa clases y atributos o propiedades en vez de tablas y columnas.
- ✓ Regresa sus resultados en forma de objetos: Las consultas realizadas usando HQL regresan los resultados de las mismas en la forma de objetos o listas de objetos, que son más fáciles de usar.
- ✓ Consultas Polimórficas: Podemos declarar el resultado usando el tipo de la superclase e Hibernate se encargara de crear los objetos adecuados de las subclases correctas de forma automática.
- ✓ Soporte para características avanzadas: HQL contiene muchas características avanzadas que son muy útiles y que no siempre están presentes en todas las bases de datos, o no es fácil usarlas, como paginación, fetch joins con perfiles dinámicos, inner y outer joins, etc. Además soporta proyecciones, funciones de agregación (max, avg), y agrupamientos, ordenamientos, y subconsultas.
- ✓ Independiente del manejador de base de datos: Las consultas escritas en HQL son independientes de la base de datos (siempre que la base de datos soporte la característica que estamos intentando utilizar).

### Autoevaluación

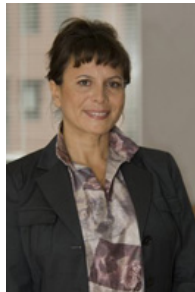
El lenguaje de consulta propio de Hibernate es:

- ☐ (link: )  
HQL .
- ☐ (link: )  
SQL .
- ☐ (link: )  
Java .



### Caso práctico

Para poder sacarle el máximo partido a las herramienta Hibernate, **Ada** es consciente que el equipo de BK debe conocer la sintaxis básicas del lenguaje HQL. Conociendo la forma de construir consultas con HQL, se abre un abanico muy grande de posibilidades, para la implementación de consultas de ciertas complejidad.



**Clausula from:** La consulta más simple que se puede realizar con Hibernate, es utilizando la cláusula `from`, la siguientes sería una consulta que mostraría todos los datos de una tabla de nombre Alumnos: `from Alumnos`

**Cláusula select:** La cláusula `select` escoge qué objetos y propiedades devolver en el conjunto de resultados de la consulta. Un ejemplo de consulta podría ser `select alumno.nombre from Alumnos alumno where alumno.nombre like 'A%'`

**La cláusula where:** La cláusula `where` nos permite refinar la lista de instancias retornadas. Si no existe ningún alias, puede referirse a las propiedades por nombre: `from Alumnos where nombre='Francisco'`. Si existe un alias, usaremos un nombre de propiedad calificado: `from Alumnos as alumnos where alumnos.nombre='Francisco'`. Esto retorna instancias de Alumnos llamados "Francisco".

**Funciones de agregación.** Las consultas HQL pueden retornar resultados de funciones de agregación sobre propiedades: `select avg(alumnos.nota), sum(alumnos.nota), max(alumnos.nota), count(alumnos) from Alumnos alumnos`.

**Expresiones.** Las expresiones utilizadas en la cláusula `where` incluyen lo siguiente: operadores matemáticos, operadores de comparación binarios, operadores lógicos, paréntesis ( ) que indican agrupación, funciones Java, etc.

**La cláusula order by.** La lista retornada por una consulta se puede ordenar por cualquier propiedad de una clase retornada o componentes. La palabra `asc` o `desc` opcionales indican ordenamiento ascendente o descendente respectivamente.

**La cláusula group by.** Una consulta que retorna valores agregados se puede agrupar por cualquier propiedad de una clase retornada o componentes:

**Subconsultas.** Para bases de datos que soportan subconsultas, Hibernate soporta subconsultas dentro de consultas. Una subconsulta se debe encerrar entre paréntesis (frecuentemente por una llamada a una función de agregación SQL). Incluso se permiten subconsultas correlacionadas (subconsultas que se refieren a un alias en la consulta exterior).

El ejemplo más simple de una sentencia HQL es el siguiente:

```
session.createQuery("from Person" ).getResultList();
```

Su equivalente en JPQL sería el siguiente, donde se requiere la cláusula `select`:

```
entityManager.createQuery("select p from Person p", Person.class ).getResultList();
```



[..\(link: AD04 CONT R26A consultaHQL.jpg.\)](#)



En la siguiente dirección Web, encontrarás una amplia información sobre el lenguaje de consulta HQL.

Tutorial de HQL. (link: [https://docs.jboss.org/hibernate/orm/5.4/userguide/html\\_single/Hibernate\\_User\\_Guide.html#hql](https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html#hql) )

### Caso práctico

**Juan** está muy contento con la implementación de la interacción de la aplicación con la base de datos. El uso de clases y objetos que de forma transparente realizan las operaciones con las base de datos, supone un gran ventaja en la programación de aplicaciones con acceso a bases de datos. Sin embargo le surge una duda, ¿qué ocurre si hay varios accesos simultáneos a un determinado datos, uno para consultar y otro para actualizarlo? ¿Cómo se pueden gestionar la transacciones?



Un transacción es un conjunto de órdenes que se ejecutan formando un unidad de trabajo, en forma indivisible o atómica.

Para la gestión de transacciones en Hibernate, no se produce bloqueo de objetos en la memoria. La aplicación puede esperar el comportamiento definido por el nivel de aislamiento de sus transacciones de las bases de datos. Gracias a la Session, la cual también es un caché con alcance de transacción. Para realizar con éxito la gestión de transacciones, ésta se van a basar en el uso del objeto Session.














El objeto Session se obtiene a partir de un objeto SessionFactory, invocando el método openSession. Un objeto SessionFactory representa una configuración particular de un conjunto de metadatos de mapping objeto/relaciona.

```
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction tx = null;
try
{
    tx = session.beginTransaction();
    // Utilizar la Session para saveOrUpdate/get/delete/...tx.commit();
} catch (Exception e)
{
    if (tx != null)
    {
        tx.rollback();
        throw e;
    }
} finally {
    session.close();
} // Al finalizar la aplicación ...HibernateUtil.shutdown( );
```

Cuando se crea el objeto Session, se le asigna la conexión de la base de datos que va a utilizar. Una vez obtenido el objeto Session, se crea una nueva unidad de trabajo (Transaction) utilizando el método beginTransaction. Dentro del contexto de la transacción creada, se pueden invocar los métodos de gestión de persistencia proporcionados por el objeto Session, para recuperar, añadir, eliminar o modificar el estado de instancias de clases persistentes. También se pueden realizar consultas. Si las operaciones de persistencia no han producido ninguna excepción, se invoca el método commit de la unidad de trabajo para confirmar los cambios realizados. En caso contrario, se realiza un rollback para deshacer los cambios producidos. Sobre un mismo objeto Session pueden crearse varias unidades de trabajo. Finalmente se cierra el objeto Session invocando su método close.

## Anexo.- Licencias de recursos.

Licencias de recursos utilizados en la Unidad de Trabajo.

| Recurso (1)  | Datos del recurso (1)   | Recurso (2)  | Datos del recurso (2)  |
|--|---|--|--|
|    | <p>Autoría: Verónica Cabrerizo.</p> <p>Licencia: GNU GPL.</p> <p>Procedencia: Captura de pantalla de la aplicación NetBeans, propiedad Sun Microsystems, bajo licencia GNU GPL v2.</p>          |    | <p>Autoría: hibernate.org.</p> <p>Licencia: GNU LGPL.</p> <p>Procedencia: <a href="http://www.hibernate.org">http://www.hibernate.org</a>.</p>   |
|    | <p>Autoría: ibatis.apache.org.</p> <p>Licencia: Copyright (cita), se autoriza el uso sin restricciones.</p> <p>Procedencia: <a href="http://ibatis.apache.org">http://ibatis.apache.org</a></p> |    | <p>Autoría: Verónica Cabrerizo.</p> <p>Licencia: GNU GPL.</p> <p>Procedencia: Captura de pantalla de la aplicación NetBeans, propiedad Sun Microsystems, bajo licencia GNU GPL v2.</p>   |
|    | <p>Autoría: Verónica Cabrerizo.</p> <p>Licencia: GNU GPL.</p> <p>Procedencia: Captura de pantalla de la aplicación NetBeans, propiedad Sun Microsystems, bajo licencia GNU GPL v2.</p>          |    | <p>Autoría: Verónica Cabrerizo.</p> <p>Licencia: GNU GPL.</p> <p>Procedencia: Captura de pantalla de la aplicación NetBeans, propiedad Sun Microsystems, bajo licencia GNU GPL v2.</p>   |
|  | <p>Autoría: Verónica Cabrerizo.</p> <p>Licencia: GNU GPL.</p> <p>Procedencia: Captura de pantalla de la aplicación NetBeans, propiedad Sun Microsystems, bajo licencia GNU GPL v2.</p>          |   | <p>Autoría: Verónica Cabrerizo.</p> <p>Licencia: GNU GPL.</p> <p>Procedencia: Captura de pantalla de la aplicación NetBeans, propiedad Sun Microsystems, bajo licencia GNU GPL v2.</p>   |
|  | <p>Autoría: Verónica Cabrerizo.</p> <p>Licencia: GNU GPL.</p> <p>Procedencia: Captura de pantalla de la aplicación NetBeans, propiedad Sun Microsystems, bajo licencia GNU GPL v2.</p>          |  | <p>Autoría: netbeans.org.</p> <p>Licencia: Copyright (cita), se autoriza el uso sin restricciones.</p> <p>Procedencia: <a href="http://netbeans.org/kb/docs/java/hibernate-java-se.html">http://netbeans.org/kb/docs/java/hibernate-java-se.html</a></p> |
|  | <p>Autoría: Verónica Cabrerizo.</p> <p>Licencia: GNU GPL.</p> <p>Procedencia: Captura de pantalla de la aplicación NetBeans, propiedad Sun Microsystems, bajo licencia GNU GPL v2.</p>          |  |  |

