Mapeo Objeto Relacional

Por Lucila Melamed, basado en el seminario de ORM dictado en 2021 por Ezequiel Escobar

En software, utilizamos un modelo de programación orientada a objetos, pero en bases de datos, utilizamos una relacional, que entiende de tablas en vez de objetos. Necesitamos un "traductor" a la hora de pasar un diagrama de clases a un DER, es decir, un ORM, quien se ocupará del **mapeo de objetos relacional.**

¿Qué es un ORM?

Un ORM (object-relational mapping) es un modelo de programación que permite mapear las estructuras de una base de datos relacional (SQL Server, Oracle, MySQL, etc.), sobre una estructura lógica de entidades con el objetivo de simplificar y acelerar el desarrollo de nuestras aplicaciones. Ejemplos son Hibernate en Java, Entity Framework en .NET

Tipos de ORM

- **Active record** → "Envuelven" a las tablas de Bases de Datos en una Clase. A esa clase se le agregan los comportamientos propios de la entidad y los comportamientos propios del manejo de ORM (save, update, remove, find,..)
  - Es decir, todas las clases deben heredar de una, que va a tener los métodos de Save, Update, Remove, Find[1] → Todas las clases dependen de sí mismas para persistirse
- **Data Mapper** → Son capas intermedias de acceso a datos (DAL), que realizan la transferencia de datos entre la herramienta de persistencia, y las entidades que conforman el dominio.
  - En general, hay una clase (*Entity Manager*), que es la que se ocupa de hacer la persistencia, así las clases de dominio quedan desligadas
    - En general esa clase ya viene creada por el ORM
  - En Java, lo único que agregamos manualmente en las clases de dominio, son las anotaciones

Lazy vs Eager Loading

- **Lazy loading** → realiza la carga en memoria de los objetos sólo al momento de su utilización.
- **Eager Loading** → realiza la carga en memoria de los objetos independientemente de si van a ser utilizados o no

¡Cuidado con las *colecciones*! Aplicado en el ejemplo "una persona tiene mascotas":
Si el mapeo entre una persona y sus animales está marcado como **eager**, cuando se quiera traer la tabla persona se va a buscar todos los animales que tiene asociados.
En cambio si esa relación está marcada como **lazy**, el ORM se ocupa de ver si es necesario o no traer toda la información de los animales.

Impedance Mismatch

Este concepto es importante a tener en cuenta cuando se piensan los modelos para un dominio dado, y se deben realizar los diagramas de clases y de entidad relación (DER). El Impedance mismatch (desajuste por impedancia) indica la incompatibilidad entre diagrama de clases y DER, ya sea en la *identidad*, *cardinalidad*, o *herencia*. A continuación se explican en detalle las estrategias posibles para mapear "en tablas" la herencia entre clases.

---

[1] Esas operaciones se conocen también como CRUD (Create, Read, Update, Delete)

Estrategias de herencia

Table per Class

Genera una tabla por cada clase hija, y NO la del padre

Si el padre tuviera atributos en común a las hijas, entonces esos atributos se repiten en cada tabla, por ejemplo:

```
ANIMAL - clase padre:
   -  nombre : string
PERRO - clase hija
GATO - clase hija
```

A la hora de pasar eso a BD, se debería crear solo a `perro` y `gato` y ambas tablas tendrían el atributo `nombre : varchar`

En ese mismo ejemplo, cuando se quiera saber “qué animales tiene una persona” se tendrá que recorrer todas las tablas hijas. Por lo tanto, para esta estrategia, *se harán consultas que tengan tantos left join como clases/tablas hijas haya*

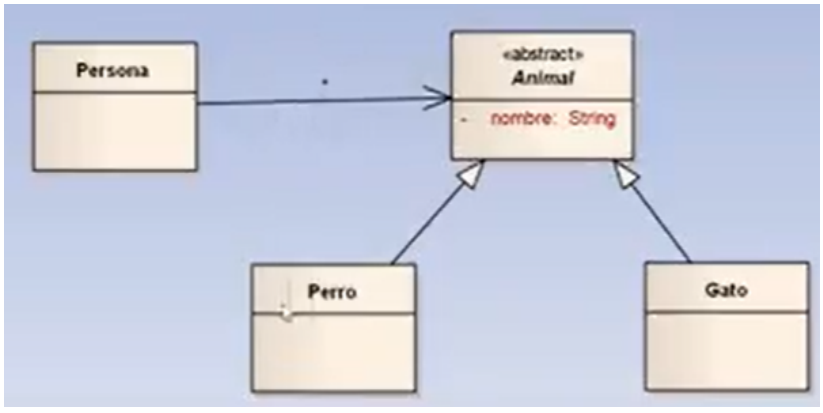
¿Cuándo conviene usarlo?

- Cuando lo único que comparten las clases hijas en el padre, es *comportamiento* (que como sabemos, no se persiste)
- Se utiliza cuando se tienen atributos comunes pero también muchos atributos propios de cada subclase

Table per Sub Class / Joined

Además de generar una tabla por clase concreta, se genera una tabla para la clase padre (o abstracta)

En el ejemplo anterior, se generarían 3 tablas: `perro`, `gato` y `animal`, y van a estar “unidas” por claves foráneas (FK), ya sea que cada clase hija tenga una referencia al padre, o el padre tenga una referencia a las hijas. Para eso, no se pueden repetir ids (PK) entre perro y gato



¿Cuándo conviene usarlo?

- Cuando cada clase hija tiene muchos atributos propios pero también tienen muchos en común

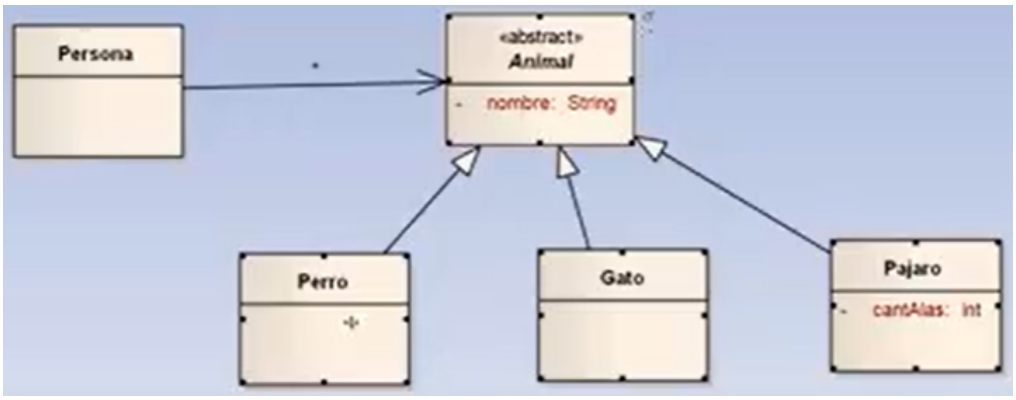
Single Table

Persiste la jerarquía completa en una sola tabla

Es simple, posee buena performance, no genera muchas tablas y simplifica las consultas en la BD

En el ejemplo anterior, se generaría una sola tabla `animal`, con una **columna discriminadora**, para saber a la hora de crear los objetos si se debe instanciar a un `gato` o un `perro.` Hablando de objetos, `animal` sería una clase abstracta, es decir, no la podría instanciar.

Con esta estrategia, puede pasar que queden algunas columnas en `null`, pero con la ventaja de ser más performante. Por ejemplo, si le agregamos una clase hija `pájaro`, que tenga `cantidadDeAlas` como atributo, la única tabla `animal` va a tener ese campo, que para los registros que pertenezcan a perros y gatos, estará nula



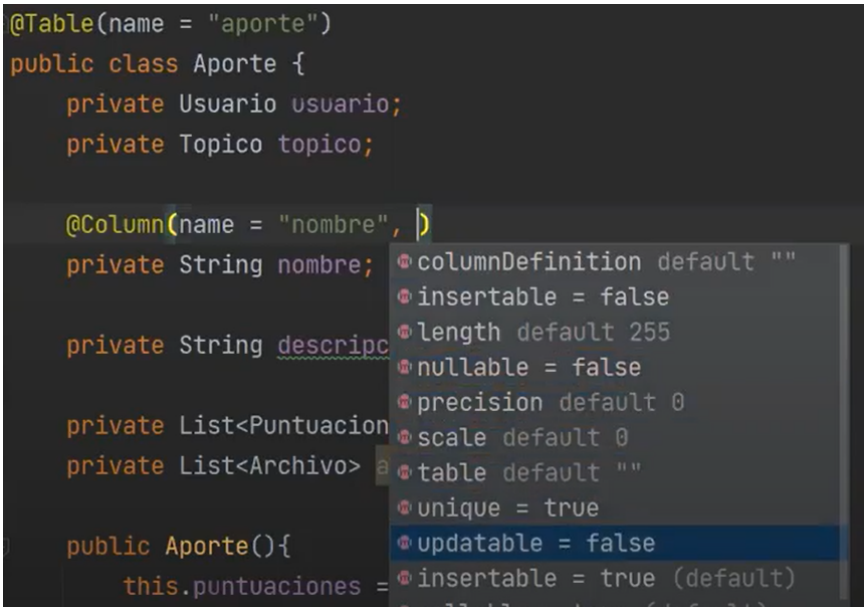
¿Cuándo conviene usarlo?

- Se utiliza cuando las subclases comparten muchos atributos en común. Es decir, cuando en total, sumando todos los atributos de los hijos y el padre, tengo pocos atributos para persistir

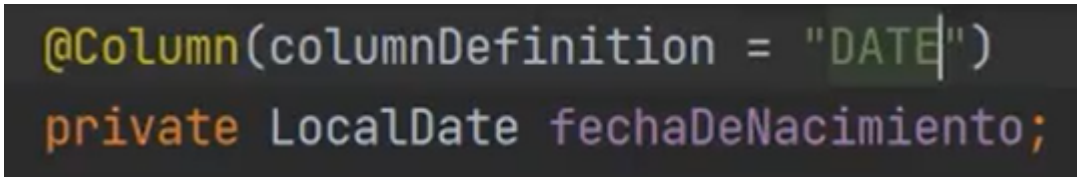
Extra: Algunos Impedance Mismatch en Java

A continuación, algunas sugerencias para solucionar estos desajustes, pero esta vez a nivel código (Java, utilizando Hibernate como ORM)

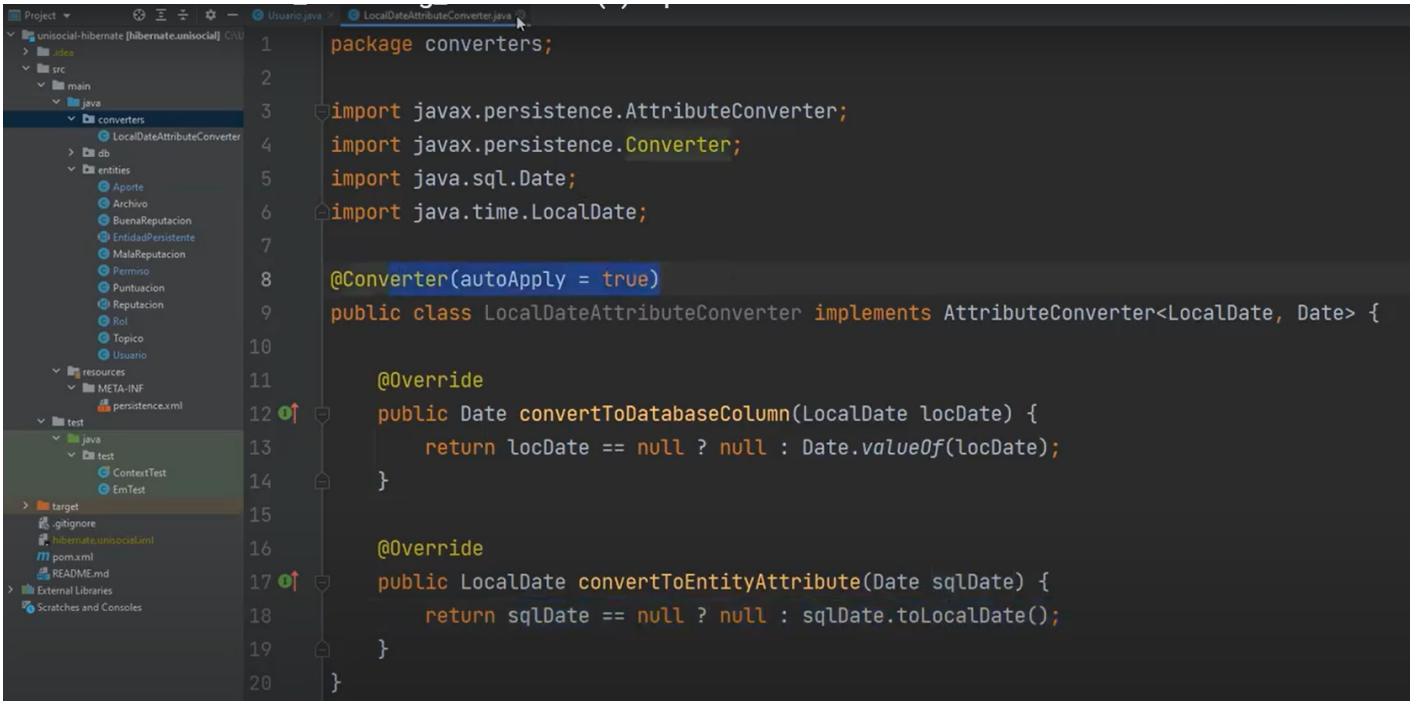
- Tipos de dato
  - Los string, Hibernate los transforma en varchar(255), pero podemos decirle otro largo por ejemplo



  - Hay otros tipos de datos que no sabe Hibernate como “traducirlos”. Una opción para solucionarlo, es la siguiente (con el ejemplo de “fechas”)
    - Anotar así la traducción



    - Crear un Converter, como el siguiente
      - El *autoApply* indica justamente que lo va a aplicar automáticamente en todos los casos que haya que convertir entre esos dos



Esta “traducción” está buena para utilizar también en los **enums**

- Identidad
  - Los objetos no suelen tener Id, pero las tablas si, así que hay dos opciones (recordar que es fuertemente recomendable, que sean *autogenerados* = @GeneratedValue):
    - Agregar el atributo Id con su annotation @Id

```
@Entity
@Table(name = "aporte")
public class Aporte {
    @Id
    @GeneratedValue
    private int id;
```

    - Crear una clase, para que todos los objetos de dominio hereden de ahí, y solo va a tener ese atributo id con la annotation @MappedSuperclass

```
@MappedSuperclass
public abstract class EntidadPersistente {
    @Id
    @GeneratedValue
    private int id;

    public int getId() { return id; }
}
```

- Herencia
  - El detalle teórico de este concepto ya fue explicado. Agregamos que, en Hibernate, la forma de aplicarlo es, en la clase padre, agregar la anotación @Inheritance y elegir alguna de las 3 estrategias mencionadas: TABLE_PER_CLASS, JOINED, y SINGLE_TABLE

```java
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class Reputacion {
```

En ese caso particular, de una única tabla, deberíamos marcar en las clases que hereden el valor discriminante (@DiscriminatorValue) que será el que se inserte en la tabla según cuál clase hija se haya instanciado

```java
@Entity
@DiscriminatorValue("buena")
public class BuenaReputacion extends Reputacion {
```

Cuando se instancie “BuenaReputacion”, en la columna discriminante de la tabla “Reputación” se guardará “buena”, y si la clase fuese “MalaReputacion”, será entonces “mala”

- Recomendamos esta página que explica bastante bien cada estrategia: https://www.baeldung.com/hibernate-inheritance

Para finalizar, los invitamos a asistir a los próximos seminarios sobre el tema, o reveer los de años anteriores. Les dejamos también a mano la Guía de anotaciones de JPA.pdf

Links de interés

Seminario ORM 2021: https://drive.google.com/drive/u/0/folders/1_5fPhdibLPfWjzNQJJjCo_jp1_FkJedW
En este video fue basado el armado del documento

Seminarios varios, de otros años:
https://drive.google.com/drive/u/0/folders/1-6IoMN248EEBUg9E9M-S6YSaNvzaTEZB