

ANNOTATIONS HIBERNATE

REALIZADO POR:

- Ignacio Mateo Villarruel
- Juan Ignacio Borda

AÑO: 2023

VERSIÓN: 1.0

Mapeo ManyToOne OneToMany	2
Mapeo ManyToOne OneToMany	3
Mapeo ManyToMany	4
Mapeo Relación OneToOne	5
MAPEO DE RELACIÓN RECURSIVA	5
MAPEO DE HERENCIA: MAPPED SUPERCLASS:	6
MAPEO DE HERENCIA: SINGLE TABLE.	7
MAPEO DE HERENCIA: JOINED.	8
MAPEO DE HERENCIA: TABLE PER CLASS.	9
RESUMEN MAPEO DE HERENCIA:	9
MAPEO DE INTERFACES.	11
MAPEO DE ENUMERADOS:	12
EMBEBER UNA CLASE:	13
MAPEO DE LISTA DE DATOS PRIMITIVOS	14
MAPEO DE DATOS PRIMITIVOS	14
IMPEDANCE MISMATCH: Data Type : LocalDate	15
NORMALIZAR Y DESNORMALIZAR.	16
Elección de PRIMARY KEY	16
OTRAS CONSIDERACIONES:	17

Mapeo ManyToOne | OneToMany

-Opción 1: BIDIRECCIONAL

class Servicio {

```
@Id
@GeneratedValue
private Integer id;

1 usage
@Column(name = "nombre")
private String nombre;

1 usage
@OneToMany(mappedBy = "servicio")
private List<Tarea> tareas;
```

}

class Tarea {

```
@ManyToOne
@JoinColumn(name = "servicio_id", referencedColumnName = "id")
private Servicio servicio;
```

}

Nota: si la dirección es bidireccional utilizar **MappedBy**

Mapeo ManyToOne | OneToMany

-Opción 2:

Si a nivel objetos va a ser Unidireccional en el OneToMany, Usamos el **JoinColumn**.

```
class Persona {
```

```
    @OneToMany
    @JoinColumn(name = "prestador_id", referencedColumnName = "id")
    private List<Disponibilidad> disponibilidades;
```

```
}
```

Notas:

- Mediante este mapeo le estamos definiendo una columna **"prestador_id"** a la **Disponibilidad**
- **referencedColumnName** es el @id de la disponibilidad.
- Si no ponemos el **@JoinColumn** hibernate va a hacer una relación manyToMany y mapear todo como hibernate quiera.

Mapeo ManyToMany

-Simple: (únicamente las FK)

```
class CommunityMember {
```

```
    @ManyToMany
    @JoinTable(
        name = "TemporalRol_CommunityMember",
        joinColumns = @JoinColumn(name = "CommunityMember_id", referencedColumnName = "id"),
        inverseJoinColumns = @JoinColumn(name = "TemporalRol_id", referencedColumnName = "id")
    )
    private Set<TemporalRolLeToService> temporaryRoles = new HashSet<>();
```

```
}
```

-Compleja:

La compleja es cuando necesitamos crear una clase nosotros para persistir porque necesitamos agregar más atributos.

Ej:

```
Class Alumno {
```

```
    ...
```

```
}
```

```
Class Materia {
```

```
    ...
```

```
}
```

Surge:

```
Class Cursada {
```

```
    ...
```

```
}
```

Mapeo Relación OneToOne

```
class Persona {
```

```
    @OneToOne()
    @JoinColumn(name = "user_id", referencedColumnName = "id")
    User user;
```

```
}
```

```
class User {
```

```
    @Entity
    @Table(name = "user")
    public class User {

        @Id
        @GeneratedValue
        private Long id;

    }
}
```

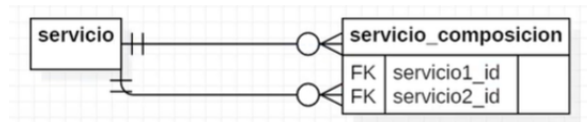
Nota:

- El user no tiene como atributo a la persona.

MAPEO DE RELACIÓN RECURSIVA

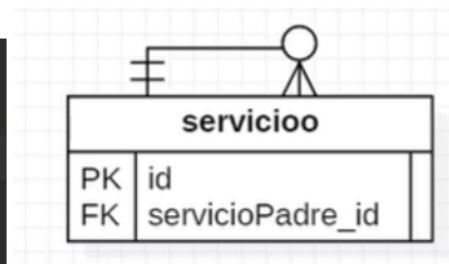
caso 1:

```
@ManyToMany
private List<Servicio> servicios;
```



caso 2:

```
no usages
@OneToMany
@JoinColumn
private List<Servicio> servicios;
```



MAPEO DE HERENCIA: MAPPED SUPERCLASS:

```
@MappedSuperclass
@Getter
public abstract class Persistente {
    no usages
    @Id
    @GeneratedValue
    private Long id;
    no usages
    private Boolean activo;
```

Desde la otra clase:

```
@Entity
@Table(name = "consumidor")
@Setter
@Getter
public class Consumidor extends Persistente {
    no usages
    @Column
    private String nombre;

    no usages
    @Column
    private String apellido;
}
```

Notas:

- Usar cuando buscamos compartir atributos y comportamientos comunes entre entidades relacionadas, pero no queremos persistir la superclase como una entidad por sí misma en la base de datos.

MAPEO DE HERENCIA: SINGLE TABLE.

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@Table(name = "reputacion")
@DiscriminatorColumn(name = "tipo")
public abstract class Reputacion extends Persistente {

    no usages 3 implementations
    public abstract void serContratado(Trabajo trabajo, Prestador prestador);

    no usages 3 implementations
    public abstract void recibirCalificacion(Calificacion calificacion, Prestador prestador);
}
```

En las clases que heredan:

```
no usages
@Entity
@DiscriminatorValue("mala")
public class ReputacionMala extends Reputacion {
    @
}
```

```
no usages
@Entity
@DiscriminatorValue("regular")
public class ReputacionRegular extends Reputacion {
    @
}
```

Notas:

- En este ejemplo estamos Extendiendo de una mappedSuperClass pero sino HAY QUE PONERLE ID a la single table.
- Es muy **performante** porque **no hay JOINS** al hidratar los objetos..
- Si tenemos 200 columnas nulas (por un decir), podríamos optar por otra estrategia.. queda en el criterio de uno.
- Tener 10 nulos no lo veo como un problema frente a la ganancia de performance.

MAPEO DE HERENCIA: JOINED.

- Suponiendo que existe una única superclase y N clases hijas, el resultado de mapear la herencia con la estrategia Joined generará, en la base de datos, una tabla por la superclase y N tablas más, una por cada clase hija.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@Table(name = "reputacion")
@DiscriminatorColumn(name = "tipo")
public abstract class Reputacion extends Persistente {
    no usages
    @Column
    private String nombre;
```

En las clases que heredan:

```
@Entity
@Table(name = "reputacion_regular")
public class ReputacionRegular extends Reputacion {
    no usages
    @Column
    private String nombreDeRegular;
```

Notas:

- El **@DiscriminatorColumn** puede no estar, **no es obligatorio** en **JOINED**
- En joined, al existir el **@Entity**, podremos decirle al ORM que nos traiga todas las Reputacion
- Perdemos performance por el join
- No tendremos nulos en cada clase Hija, buena opción si son muchos nulls.

MAPEO DE HERENCIA: TABLE PER CLASS.

- Suponiendo que existe una única superclase y N clases hijas concretas, el resultado de mapear la herencia con la estrategia Table per Class generará, en la base de datos, N tablas: una por cada clase hija.
- Todos los atributos persistentes de la superclase serán persistidos en cada una de las tablas que mapean contra las clases hijas.
- Para recuperar polimórficamente todos los objetos, el ORM debe realizar *unions* entre todas las tablas que mapean contra las clases hijas.

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Reputacion extends Persistente {
    no usages
    @Column
    private String nombre;
```

Los que heredan:

```
@Entity
@Table(name = "reputacion_mala")
public class ReputacionMala extends Reputacion {
    no usages
    @Column
    private String nombreDeMala;
```

Notas:

- Todos los atributos que estaban definidos en la clase “padre”, “bajan” a las clases hijas.
- Es buena opción cuando los **atributos** de las clases hijas no tienen nada que ver entre y queremos juntar los atributos en una singleTable para **separar “conceptos” (a nivel datos)**

RESUMEN MAPEO DE HERENCIA:

Estrategia	Nº Tablas	Pros ✓	Contras ✗	¿Cuándo usar?/Detalles
Single Table	1	Es sencilla de implementar, Tiene mejor performance por ser única tabla	Pueden existir "varios" campos nulos, a nivel conceptual no es la mejor	Necesita campo discriminador. Si hay menos de 5 campos null aprox es worth
Joined	N + 1 (padre e hijas)	Reduce mucho el número posible de campos nulos	Perjudica la performance al haber varias entidades a las que joinear	Atributos en común en la padre. Campo discriminador opcional. Las PK de las hijas son también FK hacia la padre.
Table Per Class	N (sólo las hijas)	Reduce mucho el número posible de campos nulos. Puede traer todos los elementos polimórficamente (haciendo union de todas las hijas).	Repite campos entre tablas (todos los de la padre)	La superclase no la consideramos tabla (si es abstract) pero si una entidad (@Entity). A nivel de objeto si hay comportamiento parecido, pero no atributos en común
Mapped Superclass	N (sólo las hijas)		No es entidad: El ORM no puede traer todos los elementos de su tipo polimórficamente.	La superclase no es considerada Entidad. Y lo mismo que table per class. Se usa para id.

MAPEO DE INTERFACES.

Si mi interfaz es **STATELESS**, es decir, las clases que implementan la interfaz no tienen atributos, debemos mapear la interfaz con un **CONVERTER**

Siempre y cuando alguien tenga como atributo a la dichosa interface

```
public interface MedioDeNotificacion {  
    no usages 2 implementations  
    void notificar(Prestador prestador, String mensaje);  
}
```

```
public class MedioDeNotificacionWPP implements MedioDeNotificacion {  
    no usages  
    @Override  
    public void notificar(Prestador prestador, String mensaje) {  
        // TODO  
    }  
}
```

converter para el caso de ejemplo:

```
@Converter(autoApply = true)  
public class MedioDeNotificacionConverter implements AttributeConverter<MedioDeNotificacion, String> {  
    no usages  
    @Override  
    public String convertToDatabaseColumn(MedioDeNotificacion medioDeNotificacion) {  
        String medioEnBase = null;  
  
        if(medioDeNotificacion.getClass().getName().equals("MedioDeNotificacionWPP")) {  
            medioEnBase = "wpp";  
        }  
        else if(medioDeNotificacion.getClass().getName().equals("MedioDeNotificacionEmail")) {  
            medioEnBase = "email";  
        }  
        return medioEnBase;  
    }  
}
```

```
no usages  
@Override  
public MedioDeNotificacion convertToEntityAttribute(String s) {  
    MedioDeNotificacion medio = null;  
  
    if(s.equals("wpp")) {  
        medio = new MedioDeNotificacionWPP();  
    }  
    else if(s.equals("email")) {  
        medio = new MedioDeNotificacionEmail();  
    }  
    return medio;  
}
```

El método: **convertToEntityAttribute** es para el proceso de hidratación.
Se puede utilizar un patrón Factory para no caer en tantos IF.

Finalmente Mapeamos la interfaz:

```
@Convert( converter = MedioDeNotificacionConverter.class)
@Column(name = "medioDeNotificacion")
private MedioDeNotificacion medioDeNotificacion;
```

INTERFAZ STATEFUL.

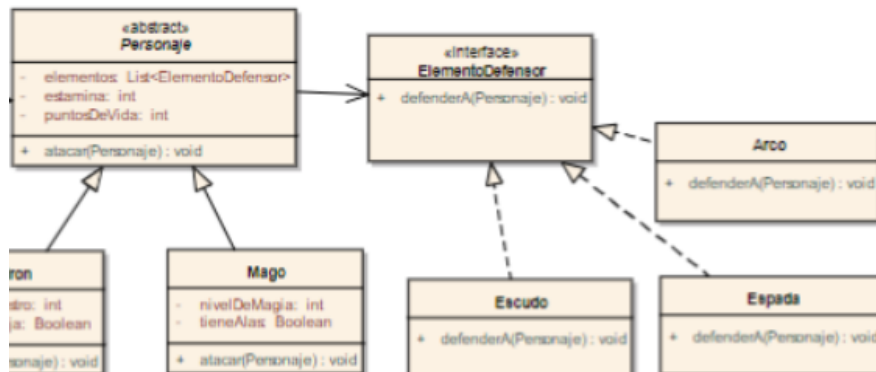
Si mi **interfaz** es **stateful**, es decir, las clases que la implementan **tienen atributos**,

1. primero deberíamos intentar pasarlos por configuración como parámetros (si son pocos) Y luego se mapea como **interfaz stateless**
2. En caso de no poder hacerlo, se convierte la interfaz en **clase Abstracta** y **se mapea** con una de las **4 estrategias de MAPEO DE HERENCIA**

Notas:

- **No hay** que mapear en una tabla una interfaz **STATELESS**, es un **error GRAVE**. (quedaría una tabla con ID e identificador)

MAPEO DE LISTA DE INTERFACES STATELESS:



Vemos que el Personaje tiene una **List<ElementoDefensor>** y elemento defensor es una **interfaz**, por lo tanto tenemos una **lista de interfaces** que **NO** tienen atributos!

Creamos el **ElementoDefensorConverter** y mapeamos de la siguiente manera

```
@ElementCollection
@CollectionTable(name = "ElementoDefensor")
@Convert(converter = ElementoDefensorConverter.class)
@Column(name = "elemento")
private List<ElementoDefensor> elementos;
```

```
@Converter(autoApply = true)
public class ElementoDefensorConverter implements AttributeConverter<ElementoDefensor, String> {

    public String convertToDatabaseColumn(ElementoDefensor elementoDefensor) {
        return elementoDefensor.getClass().getSimpleName();
    }

    public ElementoDefensor convertToEntityAttribute(String dbData) {
        switch (dbData) {
            case "Arco":
                return new Arco();
            case "Escudo":
                return new Escudo();
            case "Espada":
                return new Espada();
            default:
                throw new IllegalArgumentException("Unknown" + dbData);
        }
    }
}
```

MAPEO DE ENUMERADOS:

```
@Enumerated(EnumType.STRING)
@Column(name = "rolInCommunity", nullable = false)
private RolInCommunity rolInCommunity;
```

```
@Enumerated(EnumType.ORDINAL)
@Column(name = "rolInCommunity", nullable = false)
private RolInCommunity rolInCommunity;
```

Notas:

- **EnumType.String** es literalmente el valor del enum.
- Los enumerados **NO** se transforman en tablas

EMBEBER UNA CLASE:

class Persona {

```
@Embedded
private Direccion direccion;
```

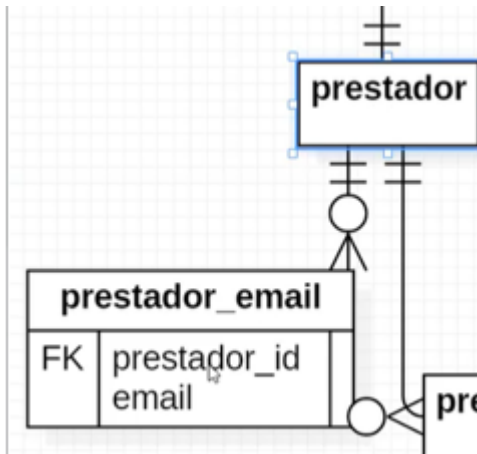
}

```
@Embeddable
@Setter
@Getter
public class Direccion {
    no usages
    @Column
    private String calle;

    no usages
    @Column
    private String altura;
```

MAPEO DE LISTA DE DATOS PRIMITIVOS

```
@ElementCollection
@CollectionTable(name = "prestador_email", joinColumns = @JoinColumn(name = "prestador_id", referencedColumnName = "id"))
@Column(name = "email", nullable = true)
private List<String> emails;
```



MAPEO DE LISTA DE ENUMERADOS

```
@Entity
@Table(name = "persona")
public class Persona {

    @Column @Id @GeneratedValue
    private Long id;

    @Enumerated(EnumType.STRING)
    @ElementCollection()
    @CollectionTable(name = "estado_persona", joinColumns = @JoinColumn(name = "id_persona", referencedColumnName = "id"))
    @Column(name = "estado")
    private List<EstadoPersona> estados = new ArrayList<EstadoPersona>();
```

	id_persona	estado
▶	1	FELIZ
	1	ABURRIDO
	1	DEPRIMIDO

	id
▶	1

```
public enum EstadoPersona {
    FELIZ, ABURRIDO, DEPRIMIDO,
}
```

MAPEO DE DATOS PRIMITIVOS

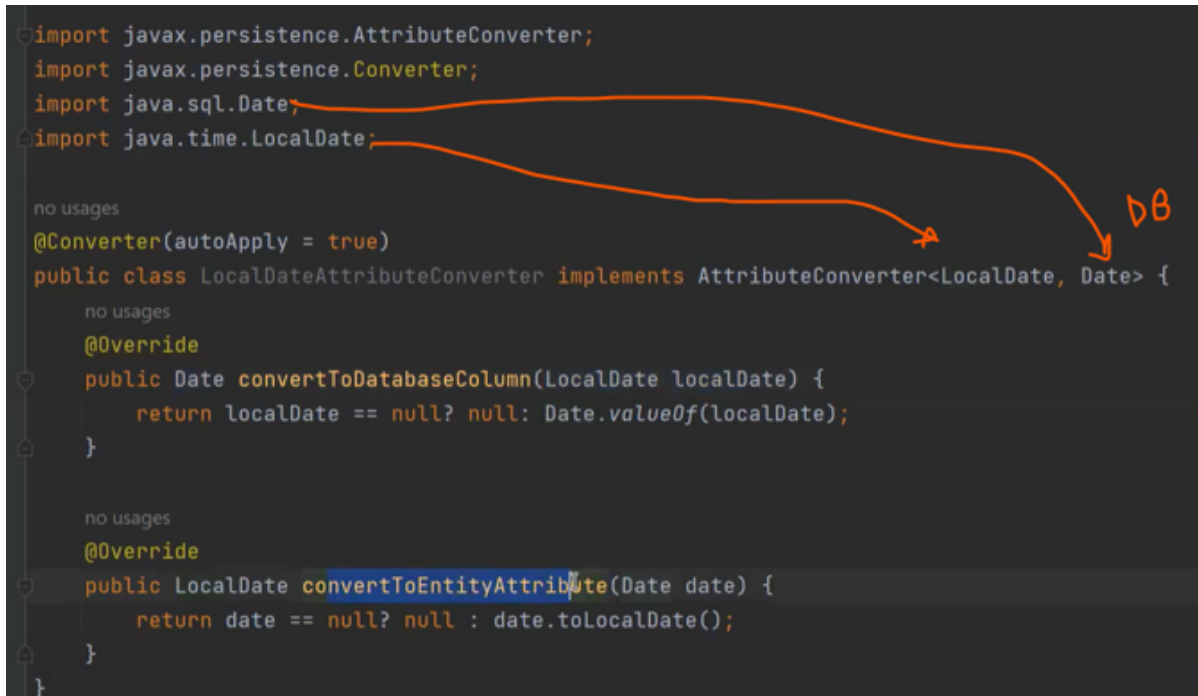
```
@Column
private String calle;
```

IMPEDANCE MISMATCH: Data Type : LocalDate

```
import javax.persistence.AttributeConverter;
import javax.persistence.Converter;
import java.sql.Date;
import java.time.LocalDate;

no usages
@Converter(autoApply = true)
public class LocalDateAttributeConverter implements AttributeConverter<LocalDate, Date> {
    no usages
    @Override
    public Date convertToDatabaseColumn(LocalDate localDate) {
        return localDate == null? null: Date.valueOf(localDate);
    }

    no usages
    @Override
    public LocalDate convertToEntityAttribute(Date date) {
        return date == null? null : date.toLocalDate();
    }
}
```



```
no usages
@Column(name = "horaInicio", columnDefinition = "TIME")
private LocalTime horaInicio;
```

NOTA:

- Existen datos que no existen en el mundo relacional. Por eso se utiliza un **CONVERTER** para persistir y luego hidratar nuestras clases.
- El ejemplo anterior muestra LocalDate y su converter.
- Ese converter se aplicara automaticamente por la propiedad **autoApply=true**
- Existen otros **impedance Mismatch**

OTRAS CONSIDERACIONES:

- Si debemos persistir **imágenes**, debemos guardar el **PATH** de la imagen como un String. Luego el **FileSystem** recupera la imagen utilizando el Path.
- En el **DER** la relación ManyToMany puede no romperse.
- En el **modelo de datos** es **obligatorio** romper la relación many to many.
- **Baja lógica** : Se dice que está dado de baja pero nunca borramos el registro realmente. Es para **Auditoría**, no borrar registro. Con esto tenemos **trazabilidad** de nuestras clases
- Los sistemas tienen **mínimo** 3 capas: presentación, Dominio y persistencia.
- La capa de presentación puede ser una API o Interfaz gráfica
- Hay 2 tipos de **ORM**: Data Mapper (hibernate) y active Record
- Siempre que tengan una relación OneToOne se puede simplificar los datos en una sola entidad. No abusen de estas relaciones, sino que solamente háganlas cuando realmente las amerite.