UA05. ORM

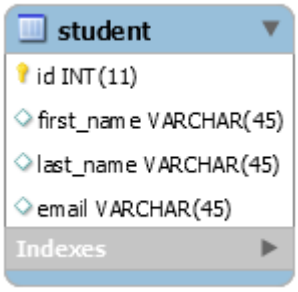
5.2.2. Anotaciones JPA

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
```

```
    @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.

Una **entidad** hará referencia a la clase que se quiere persistir en la base de datos. También la podemos nombrar como **clase entidad**.
Estas etiquetas se usan para para mapear la entidad con la base de datos. Las más importantes son:

@Entity	Se utiliza para marcar la clase como una entidad de Hibernate que se tiene que persistir en la base de datos.
@Table	Se utiliza para definir la equivalencia con la tabla de BBDD.

@Id	Sirve para indicar la clave única de la base de datos, es decir, la primary key.
@GeneratedValue	Esta etiqueta se utiliza para definir que el campo se generará automáticamente.
@Column	Se utiliza para mapear el campo con el atributo de la tabla de la BBDD, se puede definir la longitud del campo o si es obligatorio o no.
@OrderBy	Se usa para indicar que esa columna se ordenará del modo que se indique. El orden puede ser asc o desc.
@Transient	Se utiliza para indicar que ese atributo no se guarda en la base de datos.

@Entity indicará a Hibernate que esa clase se guardará en la base de datos. La etiqueta **@Table** se usará para indicar a qué tabla de la base de datos corresponde. Estas etiquetas se añaden a la clase.

En los atributos también deberemos añadir etiquetas. En primer lugar, debemos asegurarnos de que nuestra clase tendrá un atributo **@Id**, que garantizará que sea un registro único. Tendremos que añadir la etiqueta **@Id** y **@GeneratedValue**, para indicar que se trata de un identificador único autoincrementado, además de la etiqueta **@Column**, ya que esta etiqueta servirá para indicar a qué columna de la tabla corresponde.

Cada atributo de la clase deberá tener una etiqueta **@Column**.

```
@Entity
@Table (name="profesor")
public class Profesor {
@Id @GeneratedValue
@Column (name="id")
private int id;
```

```
    @Column (name="dni")
    @OrderBy("desc")
    private String dni;
    @Column(name = "nombre")
    private String nombre;
    @Column(name = "apellido")
    private String apellido;
    @Column(name = "edad")
    private int edad;
    }
```

Este tipo de anotaciones se usa para establecer **relaciones entre tablas**. En ocasiones, en las entidades se define como atributo un objeto, y este objeto tiene una relación con otra tabla, por lo que debemos definir una **relación**.

Estas anotaciones se añadirán a los atributos. Tenemos otro conjunto de anotaciones que se usan para especificar la relación entre columnas y entre diferentes tablas y entidades.

@OneToOne	Se usa para establecer la relación uno a uno.
@OneToMany	Se usa para establecer la relación de ese atributo con múltiples tablas.
@PrimaryKeyJoinColumn	Esta anotación se utiliza para asociar entidades que comparten

	la misma clave primaria.
@JoinColumn	Se usa para asociaciones uno a uno o muchos a uno cuando una de las entidades posee una clave foránea.
@JoinTable	Se utiliza para entidades vinculadas a través de una tabla de asociación.
@MapsId	Se usa para persistir dos entidades con clave compartida.
@ManyToMany	Se utiliza cuando la relación entre tablas es de muchos a muchos. Por ejemplo, un estudiante puede elegir muchas asignaturas, y las asignaturas pueden tener muchos estudiantes.

```
@Entity
@Table(name = "pregunta")
public class Entrevista {
@Id
@GeneratedValue(strategy = GenerationType.TABLE)
private int id;
private String nombrePregunta;
@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name = "id")
@OrderColumn(name = "type")
private List preguntas;
//constructor,getter, setter
}
```

Se observa que tenemos una lista de preguntas que tiene la etiqueta **@OneToMany.**

Llo que indicará esta anotación es que cada clase "**Entrevista**" tendrá una lista de preguntas y tendrá una relación **@OneToMany**, es decir, que la clase podrá tener más de una pregunta.

En Java, la herencia es una de los conceptos principales. El hecho es que transformar esta herencia en mapeo de Hibernate puede resultar un problema porque las bases de datos relacionales no cuentan con un proceso para realizar la operación, es decir, SQL, Hibernate o cualquier otro ORM no soporta este tipo de mapeo.

Para poder solucionar este problema, JPA (Java Persistance API) tiene ciertas estrategias para abordar esta problemática.

@Inheritance	Se usa para indicar qué estrategia de herencia se utilizará.
@DiscriminatorColumn	Como su nombre indica, esta columna es el discriminador, y esta anotación especifica la columna discriminada para las estrategias de mapeo de herencia SINGLE_TABLE y JOINED.
@DiscriminatorValue	Se utiliza para indicar que esa clase será filtrada con el valor que se asigne.
@PrimaryKeyJoinColumn	Se utiliza para indicar que esa columna se utiliza para unir con

	otra a través de un identificador.
@MappedSuperclass	Se utiliza para indicar que esa clase es una clase padre.

Para poder resolver la problemática, JPA abordó el problema con varias soluciones:

Mapeo superclase

Esta estrategia de mapeo es el enfoque más simple para asignar una estructura de herencia a las tablas de la base de datos. Se establece que la clase padre no puede ser una entidad. Por ejemplo, si tenemos una superclase "Persona":

```
@MappedSuperclass
public class Persona {
@Id
private long idPersona;
private String nombre;
//constructor, getters, setters
}
```

```
@Entity
public class Alumno extends Persona {
private String dni;
//constructor, getters, setters
}
```

Tendremos que añadir la anotación **@MappedSuperclass**, que indicará que esa clase no se mapea porque es una superclase. En cambio, la clase "Alumno" es una entidad porque extiende de "Persona". Por eso, a esa subclase deberemos añadirle la etiqueta **@Entity**.

En la base de datos, la clase "Alumno" equivaldrá a la tabla "Alumno" y tendrá tres columnas: las dos primeras de la superclase y la de la subclase. Por tanto, la representación en la base de datos será con los datos de la superclase más los de la entidad.

Asigna cada clase concreta a su propia tabla. Eso permite compartir la definición de atributo entre varias entidades, pero también supone un gran inconveniente: el mapeo de una superclase no es una entidad, y no hay una tabla para ello.

Tabla única

Esta estrategia es muy similar a la anterior, la principal diferencia es que ahora la superclase también es una entidad. La estrategia de tabla única crea una tabla para cada jerarquía de clase, la superclase y las subclases estarán dentro de la misma tabla. Esta es también la estrategia predeterminada elegida por JPA si no especificamos una explícitamente. Eso hace que la consulta para una clase específica sea fácil y eficiente.

Esta estrategia se centrará en agrupar las clases padre e hijo en una misma tabla. Dado que los registros de esta jerarquía de clases estarán en la misma tabla, Hibernate necesita una forma de identificar a qué entidad pertenece cada registro.

```
@Entity(name="articulos")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="tipo_articulo",
discriminatorType = DiscriminatorType.INTEGER)
public class Articulo{
//atributos,constructor, getters, setters
}
```

Como podemos apreciar por el ejemplo, la clase padre se mapeará como una entidad y se añadirá la etiqueta **@Inheritance**, definiendo la estrategia como **InheritanceType.SINGLE_TABLE**. Esta definición indicará que la estrategia de

herencia será de tabla única y es necesario añadirla a toda clase que sea superclase. Ahora deberemos definir qué columna se utilizará para distinguir a qué clase pertenece cada registro. Para ello utilizamos la etiqueta **@DiscriminatorColumn** y le asignamos un nombre "tipo_articulo". Aquí se añadirá el nombre de la clase y se podrá filtrar cuando se haga una consulta. Si no se especifica el nombre, Hibernate establecerá esa columna como **DTYPE**.

Para las subclases, se añadirá solo la anotación **@Entity** y la anotación **@DiscriminatorValue**.

```
@Entity
@DiscriminatorValue("1")
public class Lapiz extends Articulo{
//atributos,constructor, getters, setters
}
```

```
@Entity
@DiscriminatorValue("2")
public class Libreta extends Articulo{
//atributos,constructor, getters, setters
}
```

En la definición de las subclases, se debe añadir la anotación **@DiscriminatorValue**, que especificará el valor que discriminar para esa entidad. Es decir, el valor 1 indicará que la clase a la que pertenece ese registro es 1, y el 2 hará referencia a la clase "Libreta". Es una manera simple de identificar la clase a la que pertenecen, pero también se puede indicar que la columna sea un string y asignar un valor de texto. En lugar de poner **discriminatorType=DiscriminatorType.INTEGER**, se pondría **discriminatorType=DiscriminatorType.STRING**.

Este tipo de estrategia es una manera eficiente y fácil de acceder a los datos de la base de datos. Todos los atributos de cada entidad se guardan en una misma tabla, y las consultas no necesitan de joins para ejecutarse. La única

particularidad es que Hibernate necesita añadir a la consulta SQL una comparación del valor discriminador para obtener la entidad.

Joined table

Esta estrategia, a diferencia de la anterior, se encarga de mapear cada clase en una tabla propia. La única columna que se repite es el identificador, que se utilizará para enlazar las tablas.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Animal {
@Id
private long idAnimal;
private String especie;
// constructor, getters, setters
}
```

En la clase padre, deberemos definir la clase con la etiqueta **@Entity** y en strategy de la etiqueta **@Inheritance** deberemos añadir **InheritanceType.JOINED**.

Para la subclase se definirá así:

```
@Entity
@PrimaryKeyJoinColum("idGato")
public class Gato extends Animal {
private String nombre;
// constructor, getters, setters
}
```

Las dos clases tendrán un identificador "idAnimal". La clave primaria de la entidad "Gato" también tiene una restricción de clave externa para la clave primaria de su clase padre. Para personalizar esta columna, podemos agregar la anotación **@PrimaryKeyJoinColumn** y añadir el nombre de la columna.

La desventaja de este tipo de estrategia es que la recuperación de entidades requiere uniones entre tablas, lo que puede resultar en un menor rendimiento para grandes cantidades de registros. El número de combinaciones es mayor cuando se consulta la clase principal, ya que se unirá con cada elemento secundario relacionado, por lo que es más probable que el rendimiento se vea afectado en la superclase de la que queremos recuperar registros.

Ejemplo

```
import org.hibernate.mapping.List;
import javax.persistence.*;
import java.io.Serializable;

@Entity
@Table(name = "Escritores_personas")
public class PersonasEntity implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "idPersona")
    private int idPersona;
    @Column(name = "Nombre")
    private String Nombre;
    @Column(name = "Tipo")
    private String Tipo;

    PersonasEntity(String Nombre, String Tipo) {
        this.Nombre = Nombre;
```

```
        this.Tipo = Tipo;
    }

    public PersonasEntity() {
    }

    public int getIdPersona() {
        return idPersona;
    }

    public void setIdPersona(int idPersona) {
        this.idPersona = idPersona;
    }

    public String getNombre() {
        return Nombre;
    }

    public void setNombre(String nombre) {
        Nombre = nombre;
    }

    public String getTipo() {
        return Tipo;
    }

    public void setTipo(String tipo) {
        Tipo = tipo;
    }
}
```