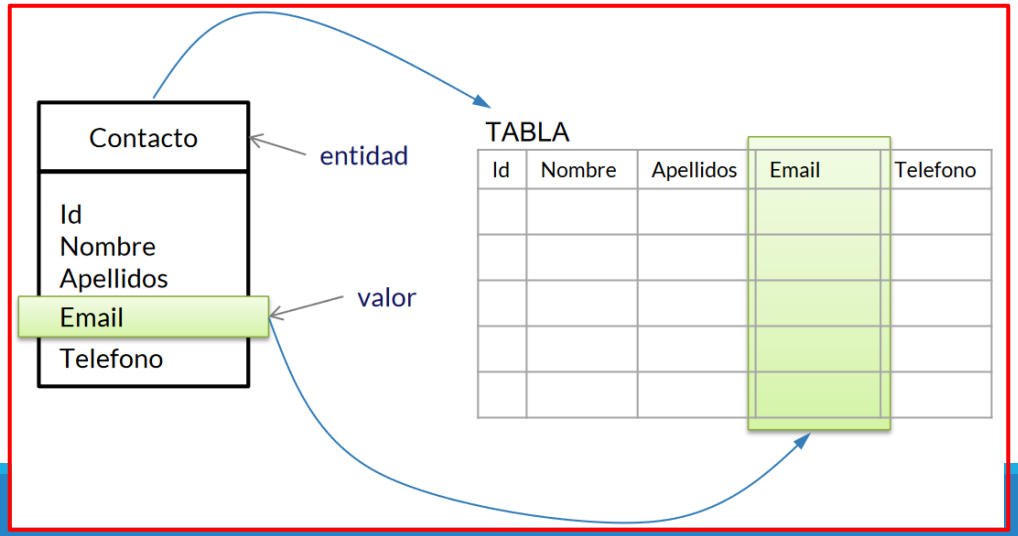
Acceso a Datos

UT4. VALORES Y ENTIDADES EN HIBERNATE

1. Introducción

- En Hibernate, una **entidad** será la representación de una **clase del modelo**. En cambio, un **valor** será la representación de un atributo de una clase.

- Por ejemplo, si estuviéramos implementando una lista de contactos, el *Contacto* sería, con toda probabilidad, una **entidad**; y el email sería un **valor** de la misma.



1. Introducción

Los **valores**, a diferencia de las entidades, no tienen un ciclo de vida propio, y existen tres categorías:

- Tipos básicos
- Tipos *embeddable*
- Tipos colección

Las **entidades** sí tienen un ciclo de vida; además, por el hecho de tener un identificador único (en el campo de las RDBMS se llama clave primaria), existen independientemente de otros objetos. Esto no sucede con los valores.

1. Introducción

En Java, hay diferencia entre **identidad e igualdad**. Dos objetos tienen la misma identidad si la comparación con el operador == devuelve true; es decir, son dos referencias apuntando al mismo objeto en memoria.

Por otro lado, la igualdad, también llamada a veces *equivalencia*, hace referencia al método *equals*. Dos objetos son iguales si su estado (el valor de sus propiedades) es el mismo. Las bases de datos relacionales complican este panorama. Dos objetos almacenados en una base de datos relacional son idénticos si están en la misma tabla y tienen la misma clave primaria (a esto se le llama identidad de base de datos).

2. Mapeo de entidades con anotaciones

Para convertir una clase Java en una **entidad**, tan solo tenemos que hacer dos cosas:

- Anotarla con @Entity.
- Incluir un atributo que esté anotado con @Id.

Por defecto, esta entidad se mapea en la base de datos con una tabla llamada MyEntity.

Como la anotación @Id es sobre un campo, por defecto, Hibernate habilitará por defecto todos los atributos de la clase como propiedades persistentes.

```
@Entity
public class MyEntity {

	@Id
	private long id;

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

2.1. Eligiendo la clave primaria

Si bien en el mundo de los diseñadores de bases de datos relacionales ha existido siempre una serie de procesos formales para la determinación de dependencias entre atributos y el cálculo de las claves primarias, es una práctica habitual entre los desarrolladores el utilizar un valor numérico (entero y potencialmente grande), como en el ejemplo anterior (*long id*).

Además, para tranquilidad del programador, este valor puede ser **autogenerado** (por ejemplo, puede comenzar en 1, y asignar el siguiente valor a cada fila insertada).

2.1. Eligiendo la clave primaria

Añadiendo la anotación *@GeneratedValue* a continuación de *@Id*, JPA asume que se va a generar un valor y va a asignar el mismo antes de almacenar la instancia de la entidad. Existen diferentes estrategias de asignación:

- GenerationType.AUTO: Hibernate escoge la mejor estrategia en función del dialecto SQL configurado (es decir, dependiendo del RDBMS).
- GenerationType.SEQUENCE: Espera usar una secuencia SQL para generar los valores.
- GenerationType.IDENTITY: Hibernate utiliza una columna especial, autonumérica. (Recomendada para MySQL)
- GenerationType.TABLE: Hibernate usa una tabla extra en nuestra base de datos. Tiene una fila por cada tipo de entidad diferente, y almacena el siguiente valor a utilizar.

En el resto de opciones para MySQL Hibernate generará en la BD una tabla denominada **hibernate_sequence** con una única columna **next_val** de tipo entero donde se guardará el siguiente valor a asignar en la secuencia.

2.1. Eligiendo la clave primaria

Generación de los ID antes o después de la inserción

Normalmente, un ORM optimizará las inserciones en una base de datos, agrupándolas por lotes. Esto supone que, en la realidad, es posible que no se realice la inserción al llamar al método *entityManager.persist(someEntity).*

Si esperamos a generar el ID a insertar los datos tiene como inconveniente que si llamamos a *someEntity.getId(),* obtendremos nulo. Por tanto, suele ser buena estrategia generar el ID antes de insertar, por ejemplo a través de una secuencia. Las columnas autoincrementales, los valores por defecto o las claves generadas por un trigger solo están disponibles después de la inserción.

2.2. Control de nombres

Por defecto, Hibernate toma una estrategia de generación de nombres para transformar el nombre de una clase (que en Java normalmente estará escrita en notación *UpperCamelCase*) al nombre de una tabla (por defecto, suele usar el mismo nombre en minúsculas).

Se puede controlar el nombre de la tabla a través de la anotación @Table(name= "NEWNAME").

```
@Entity
@Table(name= "MY_ENT")
public class MyEntity {

	@Id
	private long id;

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

3. Mapeo de valores

Cuando mapeamos una entidad, todos sus atributos son considerados persistentes por defecto. Las reglas por defecto son:

- Si la propiedad es un tipo primitivo, un envoltorio de un tipo primitivo *(Integer, Double, …), String, BigInteger, BigDecimal, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp, byte[], Byte[], char[], o Character[]* se persiste automáticamente con el tipo de dato SQL adecuado.
- Si usamos @Embeddable (lo estudiaremos después), también lo persiste.

En otro caso, **lanzará un error en la inicialización**. Hibernate escoge, dependiendo del dialecto configurado, la mejor correspondencia de tipos de dato en el RDBMS para los tipos Java que hayamos usado.

3.1. Anotación @Column

Esta anotación, sobre una propiedad, nos permitirá indicar algunas propiedades, entre las que se encuentran:

- nullable: nos permite indicar si la columna mapeada puede o no almacenar valores nulos. En la práctica, es como marcar el campo como requerido.
- name: permite modificar el nombre por defecto que tendrá la columna mapeada.
- insertable, updatable: podemos modificar si la entidad puede ser insertada, modificada, …
- length: nos permite definir el número de caracteres de la columna.

¿Dónde anotar? ¿En las propiedades o en los getter? Hibernate nos permite definir las anotaciones (@Id, @Column, …) tanto sobre las propiedades como sobre los métodos getter (**nunca sobre los setter**). La pauta la marca la anotación @Id. Allá donde usemos esta anotación, marcaremos la estrategia a seguir.

3.2. Tipos temporales

Los tipos de datos temporales, de fecha y hora, tienen un tratamiento algo especial en Hibernate. Para un campo que contiene este tipo de información, se añade la propiedad @Temporal.

Esta anotación la podemos usar con los tipos java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.util.Timestamp. Hibernate también soporta los nuevos tipo de java.time disponibles en el JDK 8.

Como propiedad, podemos indicar qué tipo de dato temporal vamos a querer usar a través del Atributo TemporalType, teniendo disponibles DATE, TIME, TIMESTAMP.

Hibernate utiliza por defecto TIMESTAMP si no encuentra una anotación @Temporal sobre alguno de los tipos de datos definidos más arriba.

```
@Entity
@Table(name= "MY_ENT")
public class MyEntity {

      @Id
      private long id;

      @Temporal(TemporalType.TIMESTAMP)
      private Date createdOn;
      //...
}
```

3.3. Tipos embebidos

En ocasiones, nos puede interesar tratar un grupo de atributos como si fueran uno solo. Un ejemplo clásico suele ser la dirección (nombre de la vía, número, código postal, …). Para este tipo de situaciones tenemos la posibilidad de convertir una clase en Embeddable. Veamos un ejemplo:

```java
import javax.persistence.Column;
import javax.persistence.Embeddable;

@Embeddable
public class Direccion {
    @Column(nullable = false)
    private String via;
    @Column(nullable = false, length = 5)
    private String codigoPostal;
    @Column(nullable = false)
    private String poblacion;
    @Column(nullable = false)
    private String provincia;
    //...
}
```

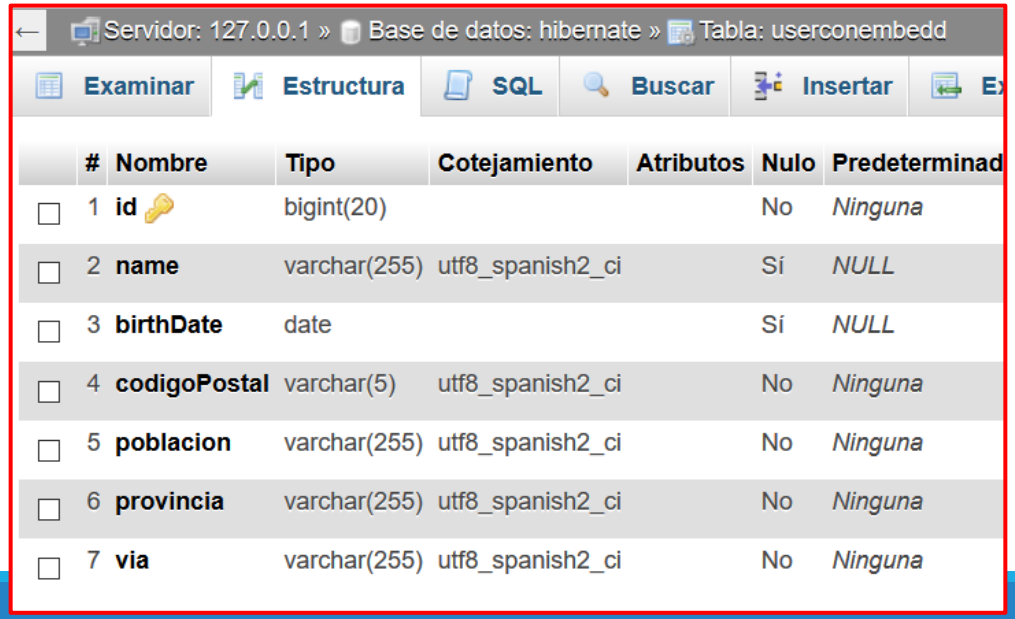
```java
import java.sql.Date;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
@Table(name = "USERCONEMBEDD")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String name;
    @Temporal(TemporalType.DATE)
    private Date birthDate;
    private Direccion address;
    //...
}
```

3.3. Tipos embebidos

De esta forma, Hibernate detecta que el campo Direccion es una clase Embeddable, y mapea las columnas a la tabla *USERCONEMBEDD*. La tabla *USERCONEMBEDD* tendría la siguiente estructura:



#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminad
1	id	bigint(20)			No	*Ninguna*
2	name	varchar(255)	utf8_spanish2_ci		Sí	*NULL*
3	birthDate	date			Sí	*NULL*
4	codigoPostal	varchar(5)	utf8_spanish2_ci		No	*Ninguna*
5	poblacion	varchar(255)	utf8_spanish2_ci		No	*Ninguna*
6	provincia	varchar(255)	utf8_spanish2_ci		No	*Ninguna*
7	via	varchar(255)	utf8_spanish2_ci		No	*Ninguna*

3.4. Sobrescritura con @Embedded

¿Qué pasaría si quisiéramos añadir dos direcciones a un usuario? Hibernate nos lanzará un error, indicando que no se soportan columnas con nombre repetido.

La solución la podemos aportar **sobrescribiendo** los atributos de la clase embebida, para que tengan otro nombre (o incluso otras propiedades) –ver siguiente diapositiva-.

La anotación @Embedded es útil cuando queremos mapear otras clases.
El atributo @AttributeOverrides selecciona las propiedades que serán sobrescritas.
El atributo @AttributeOverride indica el cambio que va a haber en un determinado atributo.

3.4. Sobrescritura con @Embedded

```
@Entity
@Table(name = "USERCONEMBEDD")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
    //Otros atributos
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name = "via",
            column = @Column(name = "VIA_FACTURACION")),
        @AttributeOverride(name = "codigoPostal",
            column = @Column(name = "CODIGOPOSTAL_FACTURACION", length = 5)),
        @AttributeOverride(name = "poblacion",
            column = @Column(name = "POBLACION_FACTURACION")),
        @AttributeOverride(name = "provincia",
            column = @Column(name = "PROVINCIA_FACTURACION"))
            })
    private Direccion billingAddress;
    //...
}
```

4. Ciclo de vida de una entidad

Si bien profundizaremos sobre este apartado en lecciones posteriores, no está de más que vayamos conociendo los diferentes estados en los que se puede encontrar una entidad JPA.

- transient (nueva): la entidad acaba de ser creada (posiblemente con el operador new) y aún no está asociada al contexto de persistencia. No tiene representación en la base de datos.
- managed, persistent: la entidad tiene un identificador y está asociada al contexto de persistencia. Puede estar almacenada en la base de datos, o aún no.
- detached: la entidad tiene un identificador, pero no está asociada al contexto de persistencia (normalmente), porque hemos cerrado el contexto de persistencia.
- removed: la entidad tiene un identificador y está asociada al contexto de persistencia, pero este tiene programada su eliminación.

4. Ciclo de vida de una entidad

Dudas y preguntas

Práctica

Utilizando un proyecto de JPA, créate una entidad Estudiante cuyos atributos sean: ID (autonumérico), DNI, nombre, apellidos, fecha_nacimiento, teléfono, e-mail y dirección (vía, código postal, población y provincia).

Crea objetos de tipo estudiante y guárdalos en la base de datos.