# ORM Básico

El estado de una entidad debe poder ser accedido por el provider en tiempo de ejecución, en el momento que sea necesario escribir o cargar datos hacia o desde la base de datos. Usando anotaciones existen dos posibles opciones para hacerlo; anotando el atributo o campo de la clase o anotando el metodo get correspondientes a dicho atributo. Si anotamos los atributos, el provider obtendrá y establecerá el valor para dicho campo usando reflection. Si anotamos el método get los métodos accesores del atributo serán usados para obtener y establecer el valor.

## Acceso de campos

Anotando los campos ocasionara que el provider use el campo para obtener y establecer el estado de la entidad. Cuando se anotan los campos los métodos accesores pueden o no estar presentes, en caso de estar presentes serán obviados por el provider. Los campos deben ser declarados protected, package o private; no deben ser declarados public bajo ninguna circunstancia.

Ejemplo:

package co.com.test;

import javax.persistence.Entity;

import javax.persistence.Id;

@Entity

public class Empleado {

@Id

private String id;

private String nombre;

public Empleado() {

}

public Empleado(String id, String nombre) {

this.id = id;

this.nombre = nombre;

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getNombre() {

return nombre;

}

public void setNombre(String nombre) {

this.nombre = nombre;

}

}

## Acceso a través de métodos accesores

Cuando se anotan los métodos get, deben declararse los métodos accesores para las propiedades persistentes. El tipo de la propiedad persistente es determinado por el valor de retorno definido en el método get y debe ser el mismo tipo del parámetro definido en el método set, ambos métodos deben ser definidos public o protected.

Ejemplo:

package co.com.test;

import javax.persistence.Entity;

import javax.persistence.Id;

@Entity

public class Empleado {

private String id;

private String nombre;

public Empleado() {

}

public Empleado(String id, String nombre) {

this.id = id;

this.nombre = nombre;

}

@Id public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getNombre() {

return nombre;

}

public void setNombre(String nombre) {

this.nombre = nombre;

}

}

## La anotación @Access

Sirve para modificar el comportamiento provisto por JPA permitiendo anotar bien sea el campo o getter y determinando cual de las políticas aplicar.

Ejemplo:

@Id

@Access(AccessType.FIELD)

public String getId() {

return id;

}

@Id

@Access(AccessType.PROPERTY)

private String id;

También es posible anotar la clase para especificar la política de acceso para la clase completa.

## La anotación @Transient

Permite especificar que un campo de una clase no será persistido en la base de datos.

Ejemplo:

@Transient

private String nombre;

## Mapeando tablas

Para mapear una entidad a una tabla solo es requerido que la entidad posea las anotaciones @Entity y @Id, pero hacerlo de esta manera implica que el nombre de la tabla sea igual al de la entidad; lo que no se ajusta a la mayoría de los casos donde ya existen la tablas de la base de datos. Para dichos casos podemos usar la anotación @Table que posee el atributo name que permite especificar el nombre de la tabla a la que vamos a mapear la entidad.

Ejemplo:

@Entity

@Table(name=”TBP\_EMPLEADO”)

public class Empleado

La anotación @Table también nos permite hacer uso del atributo catalog que nos permite especificar el catalogo al que pertenece la tabla.

Ejemplo:

@Entity

@Table(name=”TBP\_EMPLEADO”, catalog=”COREB”)

public class Empleado

## Mapeando tipos simples

Los objetos simples de Java son mapeados como parte del estado de una entidad en sus campos o propiedades. El listado de tipos que pueden ser persistidos es larga e incluye casi todos los tipos que se quiera persistir. Incluye los siguientes tipos:

* Primitivas
* Clases wrapper
* Arreglos de char o byte incluyendo arreglos de sus wrappers
* Números grandes java.math.BigInteger y java.math.BigDecimal
* String
* Datos temporales java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time y java.sql.Timestamp
* Enumeraciones tanto definidas por Java como por el usuario
* Cualquier tipo de dato serializable

## Mapeando columnas

Al igual que cuando se mapean tablas, en la mayor parte de los casos los estándares de nombrado de los objetos de base de datos definido no encaja con el estándar de nombrado de Java (Camel Case), por tal motivo existe la anotación @Column que nos permite especificar el nombre de la columna en la base de datos a la que vamos a mapear el campo o propiedad de la entidad, usando el atributo name de la entidad.

Ejemplo:

package co.com.test;

import javax.persistence.Entity;

import javax.persistence.Id;

@Entity

@Table(name=”TBP\_EMPLEADO”, catalog=”COREB”)

public class Empleado {

@Id

@Column(name=”DNI”)

private String id;

@Column(name=”DSNOMBRE”)

private String nombre;

public Empleado() {

}

public Empleado(String id, String nombre) {

this.id = id;

this.nombre = nombre;

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getNombre() {

return nombre;

}

public void setNombre(String nombre) {

this.nombre = nombre;

}

}

## Carga perezosa (Lazy loading)

En general las aplicaciones son muy sensibles en cuanto a rendimiento se refiere; por esto JPA nos brinda la posibilidad de mejorar dicho aspecto usando la carga perezosa (lazy loading, deferred loading, lazy fetching, on-demand fetching, just-in-time Reading), que nos permite especificar que atributos de la entidad no deben ser cargados cuando el objeto es inicialmente leído de la base de datos, pero será extraído solo cuando sea accedido a través de sus métodos accesores.

La anotación que permite realizar la configuración de la carga de los atributos de las entidades es @Basic a través de su atributo fetch; el cual es un tipo enumerado (FetchType) que puede tomar uno de los siguientes valores EAGER o LAZY. Asignando LAZY como valor para el atributo indica que el provider pospondrá la carga de dicho atributo hasta que sea accedido. El valor por defecto es EAGER e indica que el atributo será cargado una vez el objeto sea consultado desde la base de datos.

Ejemplo:

@Basic(fecth=FetchType.LAZY)

@Column(name=”TXT\_AYUDA”)

private String help;

Antes de usar esta característica existen algunos puntos que deben ser tenidos en cuenta acerca de la carga perezosa. El primero y mas importante es que la directiva esta pensada solo como una indicación para el persistence provider para ayudar a mejorar el rendimiento. No es mandatorio que el provider respete la petición dado que no se compromete el comportamiento de la entidad si el atributo es cargado. Cuando el valor para fetch es EAGER el provider debe cargar el atributo, dado que es crítico poder acceder al atributo una vez la entidad ha sido separada del persistence context. En general nunca es una buena idea usar carga perezosa sobre atributos simples; solo existe una pequeña mejora en el rendimiento cuando estamos seguros que el atributo no va a ser accedido posteriormente.

## Objetos Largos (LOB)

Comúnmente en las bases de datos existen tipos especiales de columnas que permiten almacenar datos en forma de bytes o caracteres con capacidades del orden de gigabytes. En JDBC este tipo de columnas requieren llamados especiales de JDBC para ser accedidas desde Java. Para indicarle al provider que debe usar métodos LOB cuando esta leyendo y almacenando a través del driver JDBC, dede hacerse uso de la anotación @Lob, esta anotación puede aparecer con o sin la compañía de las anotaciones @Basic y @Column.

En las bases de datos los LOBs aparecen de dos formas CLOBS (characters long objects) y BLOBS (binary long objects). Los tipos de datos de Java que pueden ser usados para mapear columnas de tipo BLOB son byte[], Byte[] o tipos de datos serializables, mientras char[], Character[] y String son usados para mapear las columnas de tipo CLOB. Dado que no existe una anotación diferente para cada tipo de columna LOB, el provider es responsable de realizar la diferenciación.

## Tipos Enumerados

Un tipo de datos simple que debe recibir un tratamiento especial son los datos enumerados. Los valores de una enumeración son constantes que pueden ser controlados de manera diferente dependiendo de las necesidades de la aplicación.

Como en cualquier otro, los valores de las enumeraciones en Java tienen un valor ordinal que es determinado por el orden en que fueron declarados. Dicho valor ordinal no puede ser cambiado en tiempo de ejecución y puede ser usado para representar y almacenar los valores de la enumeración en la base de datos. Interpretando los valores como ordinales es la manera por defecto en que el provider mapeara la enumeración a la base de datos, y el provider asume que la columna de la base de datos es un entero.

Ejemplo:

public enum TipoDocumento{

CEDULA,

NIT,

CEDULA\_EXTRANJERIA;

}

private TipoDocumento tipoDocumento;

Es posible modificar el comportamiento por defecto provisto por el provider usando la anotación @Enumerated y estableciendo su valor a EnumType.STRING; lo que ocasiona que el provider almacene en la base de datos el String declarado como el nombre del valor.

## Tipos Temporales

Los tipos temporales son el conjunto de tipos que podemos usar para representar el tiempo en estado persistente. El listado de tipos temporales soportados por JPA incluye java.sql.Date, java.sql.Time, java.sql.Timestamp, y adicionalmente java.util.Date y java.util.Calendar.

Los tipos temporales pertenecientes al paquete java.sql funcionan como cualquier otro tipo simple y no requieren ninguna consideración especial. Los tipos pertenecientes al paquete java.util requieren metadata adicional para especificar el tipo del paquete java.sql que debe ser usado para comunicarse con el driver JDBC. Para adicionar esta metadata es necesario anotar el campo o la propiedad con la anotación @Temporal y especificar el valor para la enumeración TemporalType; en dicha enumeración existe un valor para cada tipo temporal del paquete java.sql (DATE, TIME, TIMESTAMP).

Ejemplo:

@Entity

public class Empleado {

@Id

private String cedula;

@Temporal(TemporalType.Date)

private Date fechaIngreso;

@Temporal(TemporalType.Timestamp)

@Column(name=”FE\_MODIFICACION”)

private Date ultimaModificacion;

}

Al igual que con los demás mapeos básicos, la anotación @Column puede ser usada para sobrescribir el nombre por defecto de la columna.

## Estado transitorio

En algunas ocasiones ocurre que la entidad posee un campo que no existe en la tabla a la cual se encuentra mapeada, normalmente ocurre con campos calculados. Estos campos pueden ser definidos usando el modificador transient o anotando el atributo o la propiedad con la anotación @Transient.

Ejemplo:

@Entity

public class Empleado {

@Id

private String cedula;

@Transient

private int edad;

transient private String horasExtras;

}

## Claves Primarias

En general los campos que identifican las entidades cumplen prácticamente las mismas reglas que el mapeo de los tipos básicos, con algunas restricciones en los tipos de datos, a continuación se detallan los tipos que pueden ser usados:

* **Primitivas:** byte, int, short, long, char
* **Clases envolventes de primitivas:** Byte, Integer, Short, Long, Character
* **String:** java.lang.String
* **Numeros largos:** java.math.BigInteger
* **Temporales:** java.util.Date, java.sql.Date

Las primitivas de punto flotante y sus envolventes también pueden ser usados, pero se recomienda no hacerlo debido al error de redondeo y que el metodo .equals() no es confiable cuando es aplicado a ellos.

### Generación del Identificador

Algunas ocasiones las entidades pertenecientes al modelos del dominio no posee identificadores en la vida real o se decide no usar el identificador real que posee y se genera un atributo artificial que la identifica usando un consecutivo; esto es conocido como generación del identificador y en JPA indicamos su uso con la anotación @GeneratedValue.

Cuando la generación del identificador esta activa el provider se encargara de generar el identificador para cada instancia de la entidad. Una vez el identificador es obtenido, el provider lo insertara en la nueva entidad persistida. Dependiendo de la forma en que se genera el identificador este puede o no estar presente antes o después de que la entidad haya sido persistida. Por tal motivo el aplicativo debería no tratar de acceder a este hasta que la transacción que inicia su generación no haya sido completada.

Existen 4 posibles estrategias de generación del identificador enumeradas en GenerationType y son AUTO, TABLE, SEQUENCE Y IDENTITY.

Los generadores para las estrategias de tabla y secuencia son nombrados y accesibles para todas las entidades en la unidad de persistencia; lo que los hace reutilizables por múltiples entidades.

#### Generación Automática del Identificador

Si el tipo de generador usado por el provider no es importante para el modelo del dominio, se puede usar la estrategia AUTO. Lo que significa que el provider usara la estrategia que quiera para generar los identificadores.

Ejemplo:

@Entity

public class Pedido {

@Id

@GeneratedValue(strategy=GenerationType.AUTO)

private int consecutivo;

}

Existe una consideración que debe ser tenida en cuenta cuando se usa la estrategia de generación AUTO. El provider selecciona su propia estrategia para generar los identificadores, pero requiere un recurso persistente para hacerlo. Si escoge TABLE requiere privilegios para poder crear la tabla en donde almacenara los identificadores y si escoge SEQUENCE requiere privilegios para poder crear la respectiva secuencia; lo que supone una dificultad en ambientes de producción donde ese tipo de privilegios en general solo los posee el DBA.

#### Generación de Identificadores Usando Tabla

La forma más flexible y portable de generar identificadores es usar una tabla en la base de datos. Esta estrategia nos permite almacenar múltiples secuencias de identificadores en la misma tabla y también nos permite cambiar de motor de base de datos sin mayores complicaciones.

La tabla usada para la generación de identificadores debe tener dos columnas la primera de tipo texto como clave primaria, que será usada para identificar todas las diferentes secuencias y la segunda de tipo entero que servirá para almacenar el ultimo identificador generado para cada secuencia. Para especificar la tabla que será usada para generar los identificadores se usa la anotación @TableGenerator usando el atributo name para especificar el nombre de la tabla que se debe usar; en caso de no usar la anotación el provider usara la tabla por defecto; si la generación del esquema esta activada el provider creara la tabla de lo contrario la tabla debe existir en la base de datos.

Ejemplo:

@Entity

public class Factura {

@TableGenerator(name="FacGen",

table="TBL\_SQ",

pkColumnName="GEN\_KEY",

valueColumnName="GEN\_VALUE",

pkColumnValue="FAC\_CON",

initialValue= 0,

allocationSize=1

)

@Id

@GeneratedValue(strategy=GenerationType.TABLE, generator=”FacGen”)

private int consecutivo;

}

La anotación @TableGenerator nos permite especificar el nombre del generador en el atributo name, el nombre de la tabla de la base de datos con el atributo table, el nombre de la columna que lleva el nombre de la secuencia con el atributo pkColumnName, el nombre de la columna que lleva el ultimo valor generado con el atributo valueColumnName, el nombre de la secuencia con el atributo pkColumnValue, el valor inicial para la secuencia con el atributo initialValue y el incremento de cada valor con el atributo allocationSize.

#### Generación de identificadores usando Secuencias de Base de Datos

Si especificamos que la estrategia a usarse es SEQUENCE, pero no especificamos el generador el provider usara una secuencia por defecto, si múltiples entidades usan dicha estrategia y no definen el generador; dependiendo del provider podrían estar usando la misma secuencia por defecto, lo que podría ocasionar un comportamiento no desea en la generación de los identificadores de las diferentes entidades.

Para especificar el nombre de la secuencia que será usada debemos especificar el generador usando la anotación @SequenceGenerator y especificar el nombre del generador en el atributo name, el nombre de la secuencia en el atributo sequenceName, el valor inicial en el atributo initialSize y el incremento entre valores en el atributo allocationSize.

Ejemplo:

@Entity

public class Pedido {

@SequenceGenerator(name="PedGen",

sequenceName="SQ\_PED",

initialValue= 0,

allocationSize=1

)

@Id

@GeneratedValue(strategy=GenerationType.SEQUENCE, generator=”PedGen”)

private int consecutivo;

}

#### Generación de Identificadores usando Auto numéricos

Algunas bases de datos soportan columnas identidad para las claves primarias, algunas veces son llamadas columnas auto numéricas. Cada vez que un registro es insertado en la base de datos, un identificador único será asignado a la columna identidad.

Ejemplo:

@Entity

public class Pedido {

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

private int consecutivo;

}

Al usar la estrategia IDENTITY se debe estar seguro que la base de datos con que trabajara la aplicación lo soporta.

## Relaciones

En general las entidades pertenecientes al modelo de dominio se encuentran interrelacionadas y JPA provee los mecanismos necesarios para expresar dichas relaciones.

### Roles

Las relaciones existentes entre las entidades ocurren de una entidad a otra, lo que implica que cada relación posee dos lados, a esos lados se les conoce como roles.

Los roles que una entidad dada juega en el modelo difieren dependiendo de la relación, y una entidad puede participar en muchas relaciones diferentes. De lo anterior podemos concluir que cualquier entidad puede jugar numerosos y diferentes roles en un modelo dado.

### Direccionalidad

JPA provee múltiples mecanismos para especificar las relaciones existentes entre las entidades y la dirección en que aplica la relación. JPA permite que una entidad referencie la entidad con que se relaciona. Si las entidades que participan de una relación se referencian una a la otra se dice que la relación es bidireccional; si solo una de ellas referencia la otra la relación es considerada unidireccional.

### Cardinalidad

JPA aprovecha los mecanismos provistos por Java para poder representar el número de instancias de una entidad que se relacionaran con las instancias de otra, haciendo uso de referencias únicas, o listas y colecciones.

### Opcionalidad

JPA permite expresar cuando una relación es requerida o no, es decir, expresar si es necesario que una instancia de una entidad exista al crear una instancia de una entidad con que se relaciona.

### Mapeando Relaciones

JPA provee una anotación para cada posible caso de relación basada en la fuente de la relación y su objetivo:

1. Many-to-one
2. One-to-one
3. One-to-many
4. Many-to-many

#### Relaciones de un solo valor

Una relación desde una instancia de una entidad a otra instancia de otra entidad, es llamada relación de un solo valor. Las relaciones que pueden ser clasificadas dentro de esta categoría son many-to-one y one-to-one debido a que la fuente de la relación referencia como mucho a una instancia de la entidad objetivo.

##### Many-to-one

El mapeo muchos a uno es definido al anotar el atributo de la entidad fuente (el atributo que referencia la entidad objetivo) con la anotación @ManyToOne.

Ejemplo:

@Entity

public class Pedido {

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

private int consecutivo;

@ManyToOne

private Sucursal sucursal;

}

##### Columnas unión

En la base de datos, una relación significa que una tabla posee una referencia a otra. El termino usado para referirse a la columna que hace referencia a la llave en otra tabla y clave foránea. En JPA, son llamadas columnas unión, y la anotación usada para configurar dichas columnas es @JoinColumn.

En casi todas las relaciones uno de los dos lados de la relación tendrá la columna unión en la tabla que lo mapea. Ese lado de la relación es llamado el propietario de la relación. El lado que no posee la columna unión es llamado el lado inverso.

El propietario de la relación es importante debido a que la anotación debe ser definida en el. El mapeo many-to-one siempre debe ser definido en el propietario de la relación por tal motivo si existe una anotación @JoinColumn en la relación esta acompañando a la anotación @ManyToOne. El nombre de la columna que referencia la llave de la otra tabla se configura usando el atributo name de la anotación @JoinColumn.

Ejemplo:

@Entity

public class Pedido {

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

private int consecutivo;

@ManyToOne

@JoinColumn(name=”CDSUCURSAL”)

private Sucursal sucursal;

}

##### One-to-One

El mapeo para las relaciones uno a uno es definido de manera similar a las de muchos a uno excepto por la anotación que en el caso de las relaciones uno a uno es @OneToOne en lugar de @ManyToOne. Al igual que en las relaciones muchos a uno las relaciones uno a uno tienen una columna de unión en la base de datos y necesita sobrescribirse el nombre de la columna usando la anotación @JoinColumn.

Ejemplo:

@Entity

public class Factura {

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

private int consecutivo;

@OneToOne

@JoinColumn(name=”CDPEDIDO”)

private Pedido pedido;

}

Frecuentemente la entidad objetivo de las relaciones uno a uno posee una relación de regreso hacia la entidad fuente; cuando esto ocurre esta relación es llamada relación bidireccional uno a uno. Para poder agregar la relación de regreso es necesario añadir la anotación @OneToOne en el atributo que representa la entidad fuente de la relación en la entidad objetivo y asignando al atributo mappedBy el nombre del campo en la entidad fuente que representa a la entidad objetivo.

Ejemplo:

@Entity

public class Pedido {

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

private int consecutivo;

@OneToOne(mappedBy=”pedido”)

private Factura factura;

}

##### One-To-Many

Cuando una instancia de una entidad se encuentra asociada con varias instancias de otra entidad, ocurre más frecuentemente en la forma una a muchas. Una relación bidireccional uno a muchos siempre implica una relación muchos a uno de regreso a la entidad fuente. Para poder representar este tipo de relaciones es necesario que la entidad objetivo defina un atributo de tipo Collection o List y como parámetro genérico para dicho atributo debe ser definido el tipo de la entidad fuente; dicho atributo debe poseer la anotación @OneToMany especificando en el atributo mappedBy el nombre del atributo correspondiente a la entidad objetivo.

Ejemplo:

@Entity

public class Sucursal {

@Id

private String codigo;

@OneToMany(mappedBy=”sucursal”)

private List<Pedido> pedidos;

}

**Nota:** para observar la entidad fuente de la relación véase el ejemplo del apartado Columnas de unión.

Es importante tener en cuenta que el lado de la relación que posee la anotación @JoinColumn es el propietario de la relación y el restante es el inverso y debe definir el elemento mappedBy.

##### Many-To-Many

En este tipo de relación cada entidad poseerá un atributo de tipo Collection o List con parámetro genérico del tipo de la entidad objetivo. Una relación muchos a muchos es definida en ambos lados de la relación la fuente y el objetivo con una anotación @ManyToMany en el atributo de tipo Collection o List. El atributo mappedBy de la anotación @ManyToMany debe encontrarse en el lado inverso de la relación, en el caso de este tipo de relaciones es arbitrario cual de los lados es el propietario y cual el inverso dada la multiplicidad de la relación. En el modelo de base de datos para poder representar este tipo de relaciones es necesario contar con una tercera tabla a la cual se llama tabla de intersección. Para representar la tabla de intersección en JPA usamos la anotación @JoinTable asignando en el atributo name de la anotación el nombre de la tabla de intersección, en el atributo joinColumns el listado de @JoinColumn que representan la clave foránea a la entidad propietario de la relación y en el atributo inverseJoinColumns el listado de @JoinColumn que representan la clave foránea a la entidad objetivo de la relación.

Ejemplo:

@Entity

public class Pedido {

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

private int consecutivo;

@ManyToMany

@JoinTable(name=”TBLPRODUCTOS\_PEDIDOS”,

joinColumns=@JoinColumn(name=”CDPEDIDO”),

inverseJoinColumns=@JoinColumn(name=”CDPRODUCTO”))

private List<Producto> productos;

}

@Entity

public class Producto {

@Id

private String codigo;

@ManyToMany(mappedBy=”productos”)

private List<Pedido> pedidos;

}

##### Carga perezosa

JPA provee en todas las anotaciones para definir relaciones (@OneToOne, @OneToMany, @ManyToOne, @ManyToMany) el atributo fetch que nos permite recomendar al provider la forma en que debe realizarse la carga de las relaciones de la misma manera que se hace con los atributos básicos.