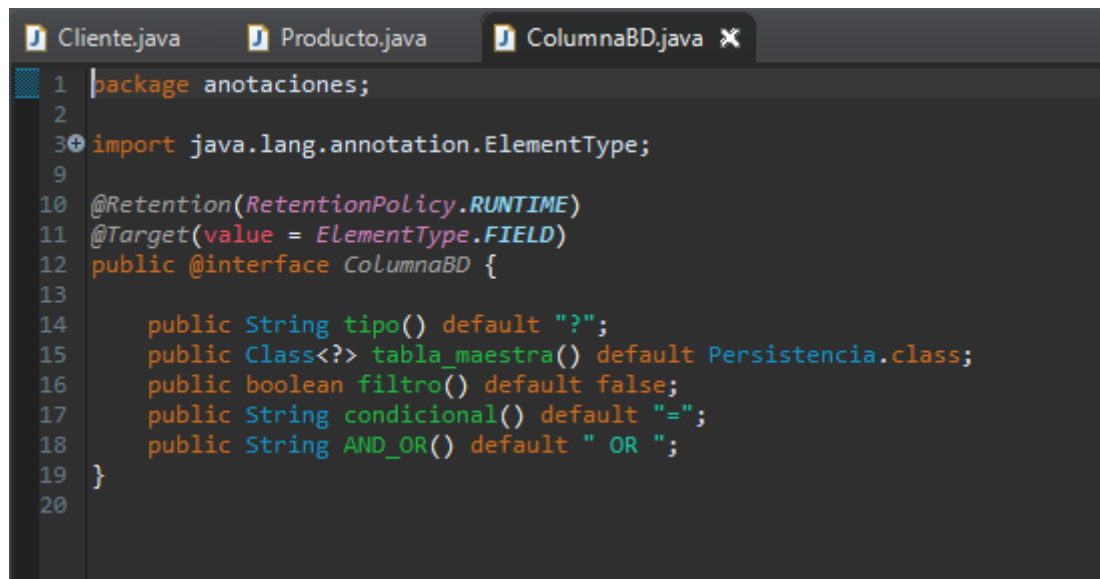


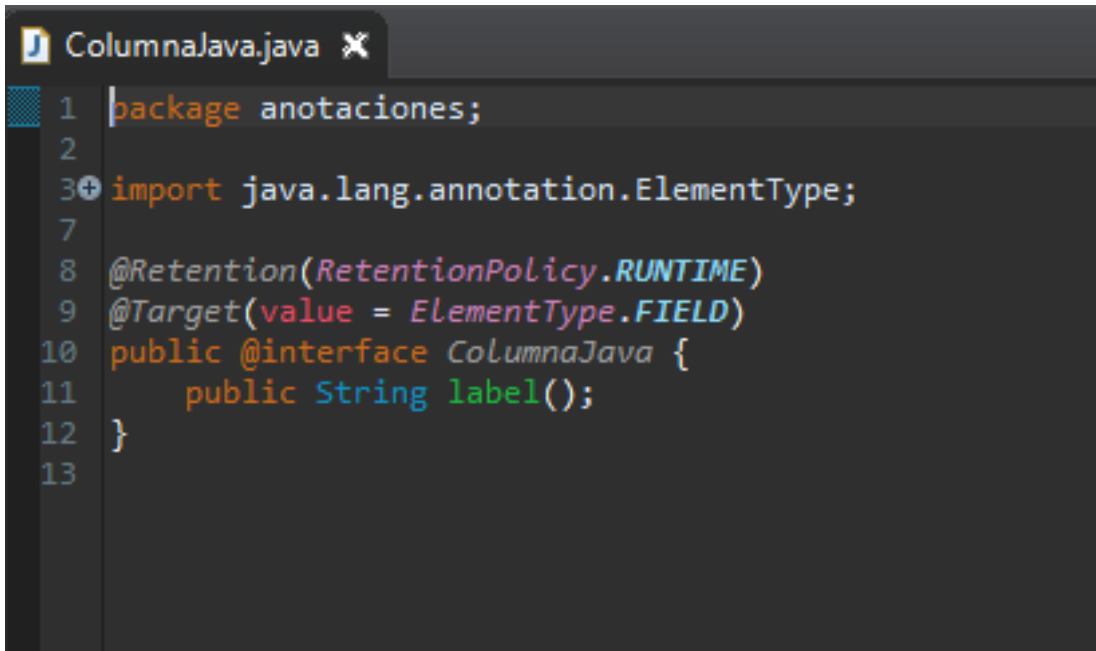
DOCUMENTO DE GUÍA PARA LA IMPLEMENTACIÓN DE LA API



```
1 package anotaciones;
2
3 import java.lang.annotation.ElementType;
4
5
6
7
8
9
10 @Retention(RetentionPolicy.RUNTIME)
11 @Target(value = ElementType.FIELD)
12 public @interface ColumnaBD {
13
14     public String tipo() default "?";
15     public Class<?> tabla_maestra() default Persistencia.class;
16     public boolean filtro() default false;
17     public String condicional() default "=";
18     public String AND_OR() default " OR ";
19 }
20
```

Figura 1. Clase anotación *ColumnaBD*

- a) **Tipo:** El metadato `tipo()` es para indicar el tipo de dato que tendrá la columna en la tabla de la base de datos, en caso de no especificar, la API está preparada para establecer de acuerdo a los tipos de datos correspondiente a cada uno.
- b) **Tabla maestra:** El metadato `tabla_maestra()` sirve para indicar a la API, a qué entidad pertenece el atributo que ha sido declarado como composición, el cual por defecto tendrá como valor a la clase `Persistencia` únicamente para que el valor del mismo no sea nulo porque de ser así el programa se detiene.
- c) **Filtro:** El metadato `filtro()` corresponde a la condición de búsqueda que tendrá la sentencia SQL en caso que se desee hacer una filtración específica. Para la implementación del mismo se deberá invocar a la anotación de esta manera `@ColumnaBD(filtro=true)` sobre el atributo que será utilizado como criterio de filtro, el valor predeterminado de este metadato es *false* (falso).
- d) **Condicional:** Este metadato es para condicionar el filtro de búsqueda de la sentencia SQL, en caso que se desee comparar las similitudes de cada letra ingresada como dato, se deberá usar la palabra reservada “ILIKE”, se implementará de esta forma `@ColumnaBD(filtro=true, condicional=“ILIKE”)`, pero si se quiere filtrar por un dato exacto al cien por cien, se debe usar el signo igual “=”, el valor predeterminado de este metadato es “=”.
- e) **AND_OR:** Está ideado para mejorar el criterio de filtro con una condicional extra de inclusión o exclusión. **Ejemplo:** Filtrar por nombre y apellido, en este caso la condición deberá ser “AND”, en caso se quiera filtrar por cédula o nombre, la condición deberá ser “OR”. El valor predeterminado de este metadato es “OR”.



```
1 package anotaciones;
2
3 import java.lang.annotation.ElementType;
4
5
6
7
8 @Retention(RetentionPolicy.RUNTIME)
9 @Target(value = ElementType.FIELD)
10 public @interface ColumnaJava {
11     public String label();
12 }
13
```

Figura 2. Clase anotación *ColumnaJava*

- a) **Label:** El metadato label() sirve para asignarle el nombre a las columnas de las tablas de java. En el caso de que esta anotación no sea invocada, el nombre de las columnas será tomado del nombre de los atributos de la clase modelo.

```

1 package modelo;
2
3 import anotaciones.ColumnaBD;
4
5 public class Producto {
6     public int id;
7     @ColumnaBD(filtro = true, condicional = "ILIKE")
8     public String descripcion;
9     public double precio;
10    @ColumnaBD(tabla_maestra = Marcas.class)
11    public Marcas marca;
12    public boolean estado;
13
14    public Producto() {
15        this.id = 0;
16        this.descripcion = "";
17        this.precio = 0d;
18        this.marca = new Marcas();
19        this.estado = true;
20    }
21
22
23 }
24

```

Figura 3. Ejemplo de la *clase* modelo Producto

En la figura 3, se puede observar como ejemplo la estructura de una clase Producto con sus respectivos atributos, por defecto, para el número identificador (**id**) de las entidades, obligatoriamente deberá tener ese nombre, el motivo es que los algoritmos de la API lo exige porque dentro de sus funciones siempre buscará un atributo con el nombre id y con el objetivo de evitar crear un metadato o anotación para ello, sencillamente con identificar a este atributo lo considerará como la clave primaria (Primary Key). Siguiendo con el ejemplo, a continuación, el atributo descripción tiene la anotación ColumnaBD, empleando los metadatos **filtro** y **condicional** (véase Figura 1, incisos c & d). Los atributos que corresponden a una composición, requieren de la anotación ColumnaBD y el metadato tabla_maestra (véase Figura 1, inciso b), en donde se indique con qué tabla se relaciona el atributo declarado, para este caso, Marca compone a Producto por lo que para la sentencia SQL, la API utilizará la función `JOIN productos ON pro_marca=mar_id`.

Una importante observación que se puede reflejar en este ejemplo, es que no se requieren los métodos getters ni setters, la API tiene su propia dinámica interna de capturar o cargar valores a los atributos, por tanto, no se crean estos métodos que ya resultan innecesarios. Por último, un dato no menos importante, todos los atributos deberán ser públicos, esto es porque la API requiere acceso total a cada uno de ellos y la palabra reservada public lo hace posible.

```

1 package lib;
2
3 import java.util.ArrayList;
4
5
6
7
8 public class Inicializar {
9     public static void main(String[] args) {
10         ArrayList<Class<?>> lista = new ArrayList<>();
11         lista.add(Marcas.class);
12         lista.add(Producto.class);
13         Persistencia.crearTablas(lista);
14     }
15 }

```

Figura 4. Ejemplo de la *entidad* inicializar

Para que la herramienta pueda hacer el procedimiento creacional de las tablas, es necesario hacer una implementación similar a la que se muestra en la figura 4, donde se instancia una lista de tipo *Class* y posterior a eso se le agrega las clases que serán las futuras tablas de la base de datos. Finalmente, se invoca al método *crearTablas*, el cual pide como parámetro la lista. Este procedimiento se hace por una única vez, una vez creada las tablas, se procede a la implementación de los demás métodos.

La implementación de la API en un formulario ABM requiere la instancia de la *clase modelo* de la entidad, para este ejemplo, se va a utilizar a la entidad *Marca*, la cual estará instanciada de la siguiente manera: `Marcas m = new Marcas();`

```

194 private void guardar() {
195     m.descripcion = tfDescripcion.getText();
196     m.estado = rdbtnEstado.isSelected();
197     Persistencia.guardar(m);
198 }

```

Figura 5. Ejemplo del método *guardar* de la entidad *Marca*.

En la presente figura, se puede ilustrar la carga de los atributos con los datos provenientes de los componentes. El siguiente paso es invocar al método *guardar* de la clase *Persistencia* (línea 197), el cual pide un parámetro de tipo *Object*, en el cual se le indica la instancia de la clase *Marca* (*m*). La API tiene la capacidad de interpretar si es un nuevo registro o un registro para actualizar, de ahí el nombre del método se engloba en '*guardar*'.

```

199 private void eliminar() {
200     Persistencia.eliminar(m);
201 }

```

Figura 6. Ejemplo del método *eliminar* de la *entidad* *Marca*.

En la figura ilustrada más arriba, se puede observar la implementación del método eliminar, el cual tiene únicamente una línea de código, se invoca al método *eliminar* de la clase *Persistencia*, el cual también pide un parámetro de tipo *Object*, en el cual se le indica la instancia de la clase *Marca* (*m*)

```
202 private void buscar() {  
203     Persistencia.cargarTabla(Marcas.class, table, listaMarca, tfBuscar.getText());  
204 }  
205 }
```

Figura 7. Ejemplo del método *cargarTabla* de la entidad *Marca*.

En esta figura se muestra la implementación del método de consulta, al cual se le puso como nombre *buscar*, la razón del nombre es que este método implica la carga del *JTable* y la lista, utilizando como criterio de filtro el dato obtenido en la caja de texto que corresponde a la función *buscar*.

```
229 private void cargarComboBox() {  
230     Persistencia.cargarComboBox(Marcas.class, comboBox, listaMarcas, "");  
231 }
```

Figura 8. Ejemplo del método de carga del componente *JComboBox*.

En la figura 7 se muestra la implementación de carga de lista de marcas, esta información es necesaria ya que la entidad *Producto* está asociada a *Marca* y para que el programador pueda identificar de qué marca es un producto, necesita hacer la implementación a partir de la lista de esa entidad. El método *cargarComboBox*, pide 4 parámetros, el primero corresponde a la entidad, el segundo al componente, el tercero la lista y por último una cadena de caracteres que corresponde al filtro (por si se necesita un filtro en especial), que para este caso estará vacío.

```
216 private void guardar() {  
217     p.descripcion = tfDescripcion.getText();  
218     p.precio = Double.parseDouble(tfPrecio.getText());  
219     p.marca = (Marcas) listaMarcas.get(comboBox.getSelectedIndex());  
220     p.estado = rdbtnEstado.isSelected();  
221     Persistencia.guardar(p);  
222 }
```

Figura 9. Ejemplo del método *guardar* de la entidad *Producto*.

En la figura 8 se ilustra la implementación del método guardar para la entidad Producto (véase figura 4), con una línea de código con implementación diferente a las demás (línea 219). El paso final para insertar o modificar el registro se muestra en la línea 221 de la figura.

```
1 servidor=?  
2 puerto=?  
3 usuario=?  
4 nombre_bd=?  
5 password=?|
```

Figura 10. Ejemplo del Archivo credencial.properties

En la figura 10 se muestra el archivo credencial.properties que sirve para ingresar los datos necesarios para hacer la conexión con la base de datos. Este archivo tiene que estar ubicado en la carpeta raíz del proyecto.