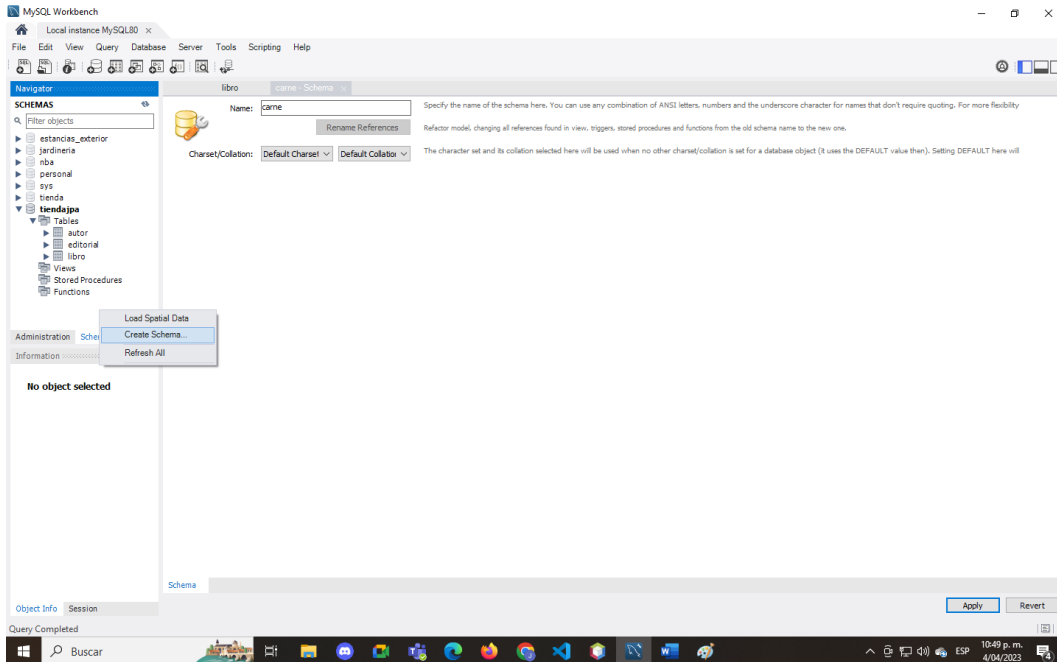


INSTRUCTIVO JPA

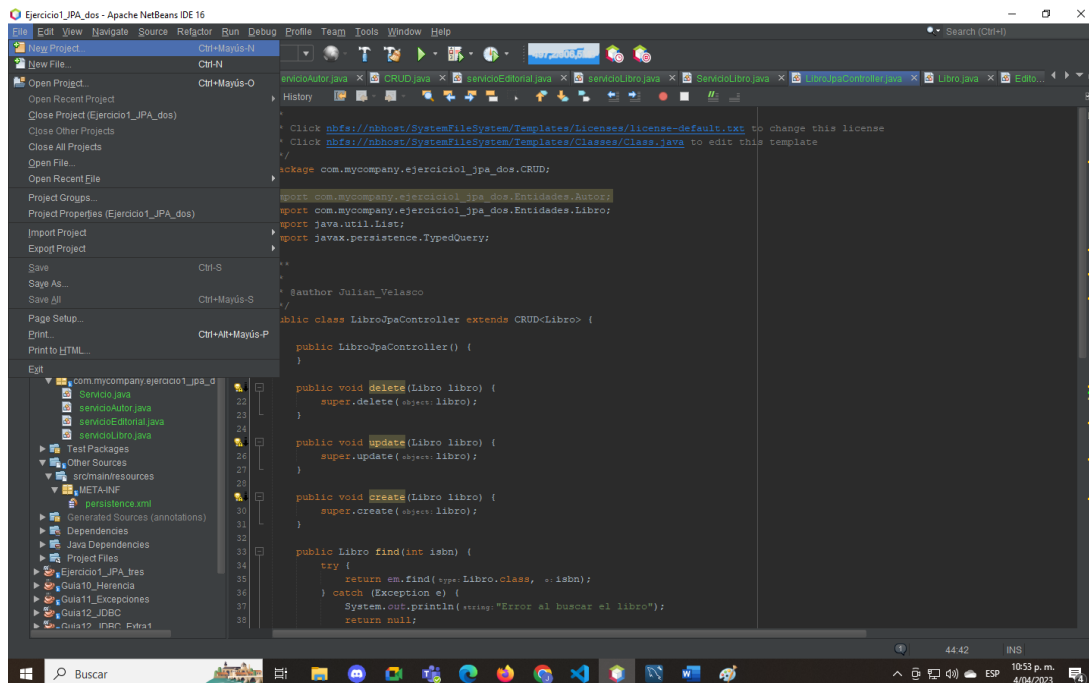
1. Crear base datos vacía en Workbench

El primer paso es la creación de la base de datos. Existen diversas maneras de hacerlo, una de ellas es dar click derecho en el espacio donde están reportadas las bases de datos, luego dar click en **create schema**, elegimos un nombre en letras minúsculas y culminamos con click en **Apply**.

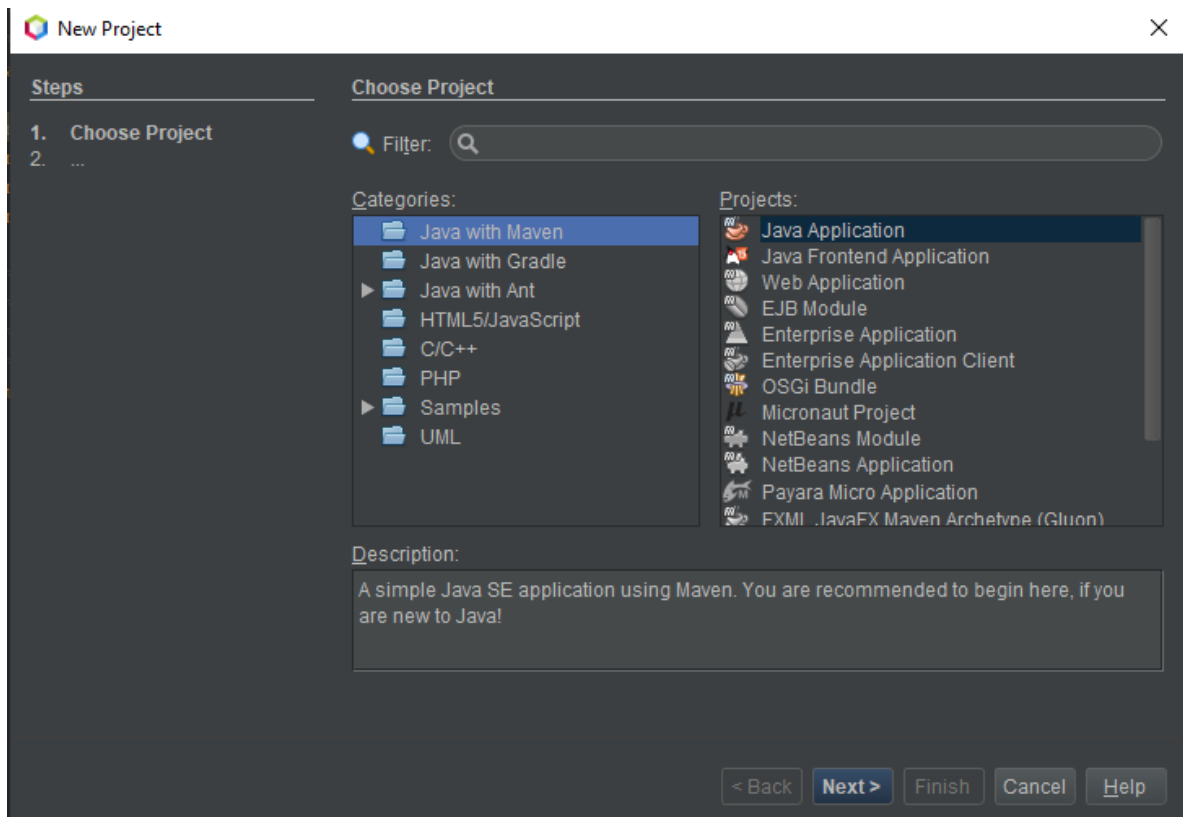


2. Creación de un nuevo proyecto

Creamos un nuevo proyecto en nuestro programa, en el caso de NetBeans, damos click en **File** y luego elegimos la opción **New Project**.



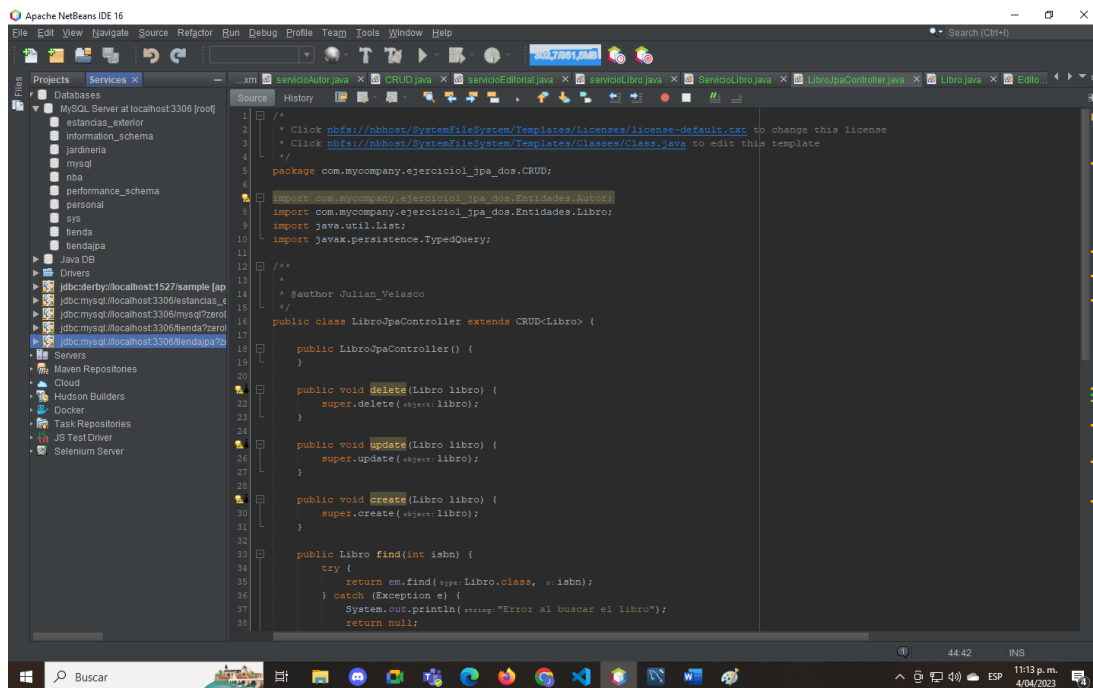
3. Seleccionamos en **Categories** la opción **Java with Maven** y en **Projects**, elegimos la opción **Java Application**. Damos click en **Next**, lo cual despliega otra ventana, en donde podemos dar el nombre a nuestro proyecto.



4. Establecemos conexión con la base de datos

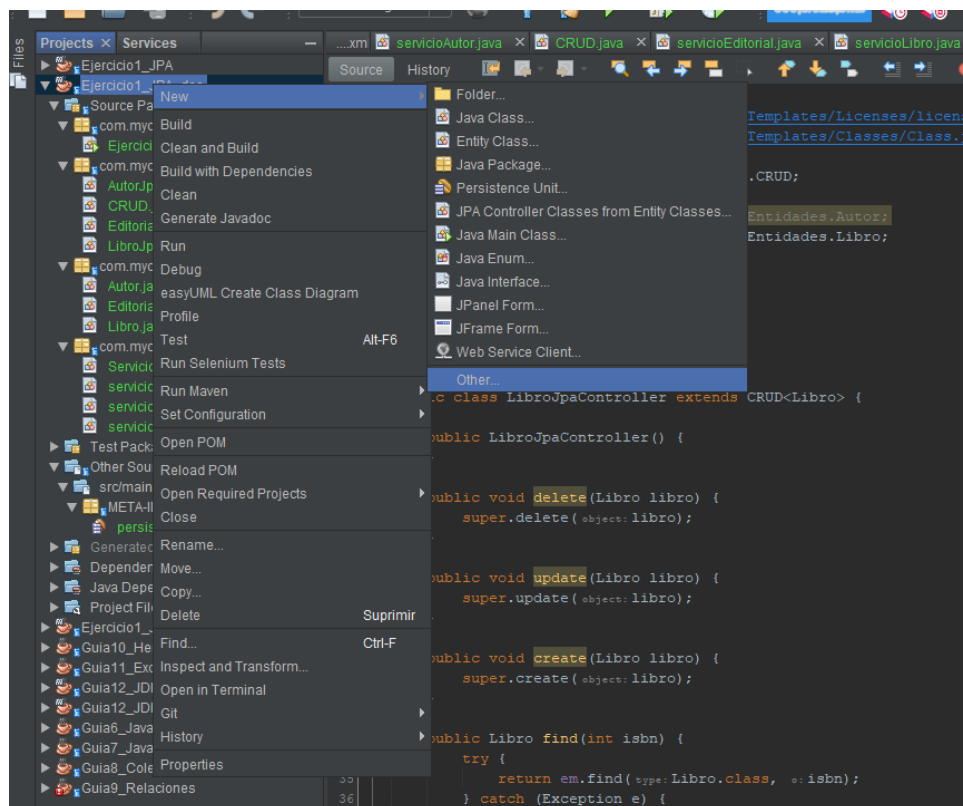
Nos dirigimos ahora a la pestaña **Services**, para luego desplegar la opción **Databases**, que inmediatamente traerá la opción **MYSQL Server at localhost:3306[root]** para quienes estén conectados, si no está conectada entregará un mensaje informando que está desconectada, así que debemos dar click derecho sobre la opción **MYSQL Server at localhost:3306[root]**, luego se selecciona **connect** y se establece la conexión.

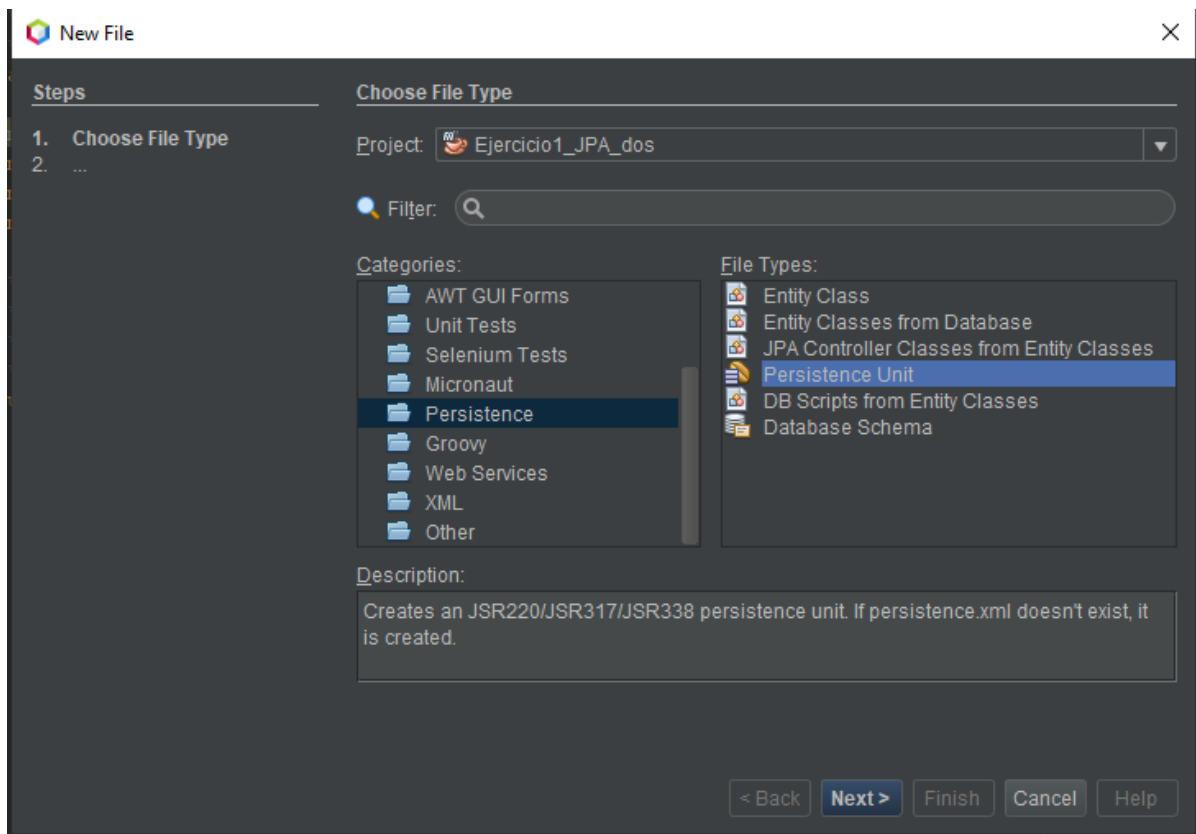
Luego se despliegan las opciones que aparecen en la opción **MYSQL Server at localhost:3306[root]**, buscando la base de datos que acabamos de crear en **Workbench**. De igual manera, click derecho sobre la base de datos y luego se selecciona la opción **connect**. Acto seguido, debería desplegarla en la parte inferior.



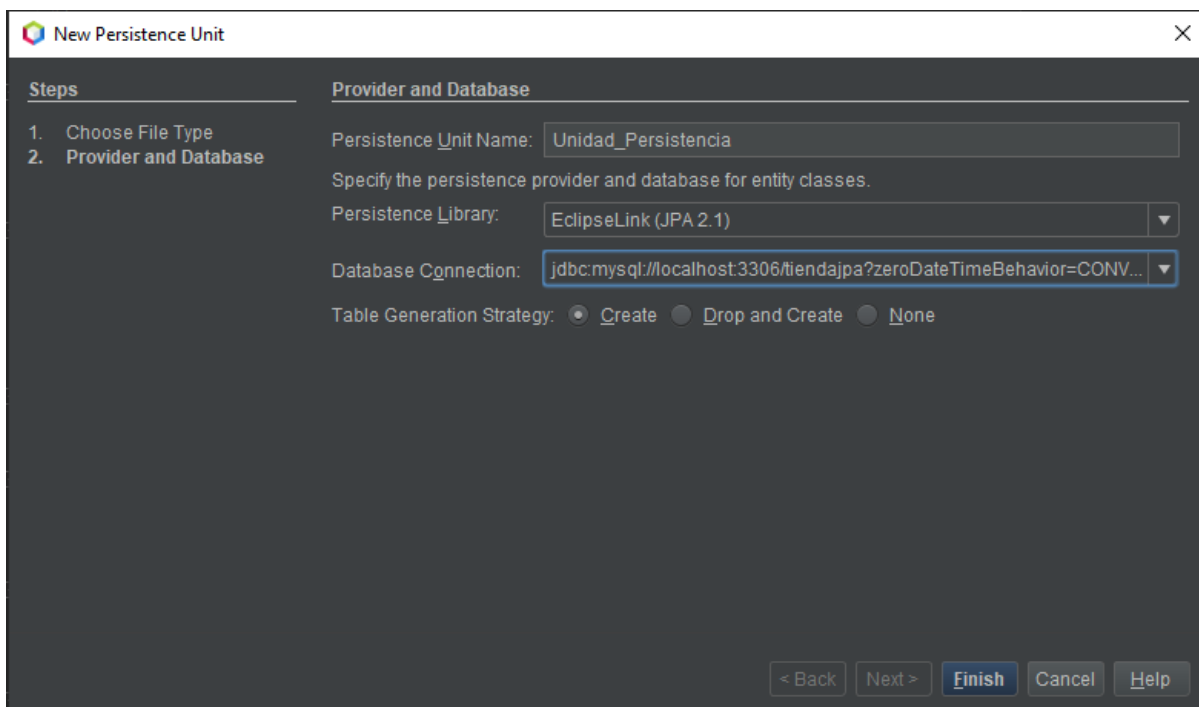
5. Creamos la unidad de persistencia.

Volvemos a la pestaña **Projects**, damos click derecho sobre nuestro **proyecto**, luego ingresamos a la opción **new** y debemos buscar la opción **Persistence Unit**. Si es la primera vez que se creará, debemos dirigirnos a la opción **others** y en esa nueva ventana, en la opción **Categories**, seleccionamos **Persistence**. Finalmente, en la opción **File Types**, seleccionamos **Persistence Unit** y finalizamos con click en **Next**.



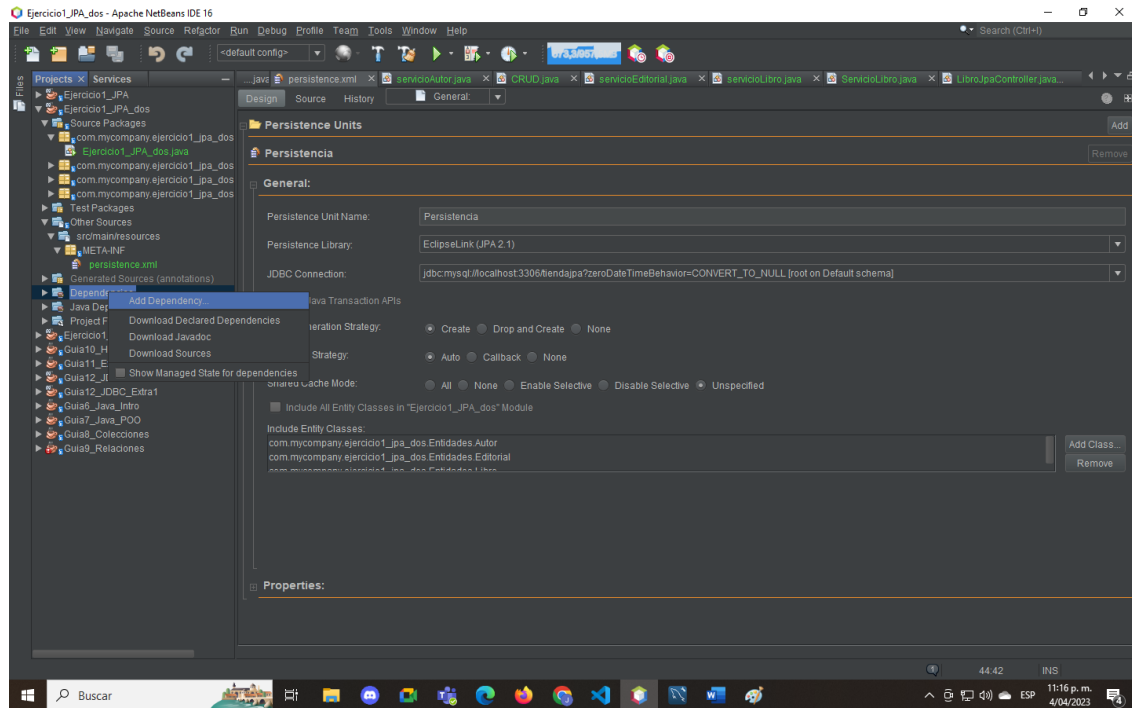


Luego, se abrirá una ventana en la que se ingresa el nombre que se le dará a la persistencia, la librería debe ser la que aparece de fábrica, **EclipseLink (JPA 2.1)**. Finalmente, en **Database Connection**, seleccionamos la librería que creamos en Workbench y damos click en **Finish**.

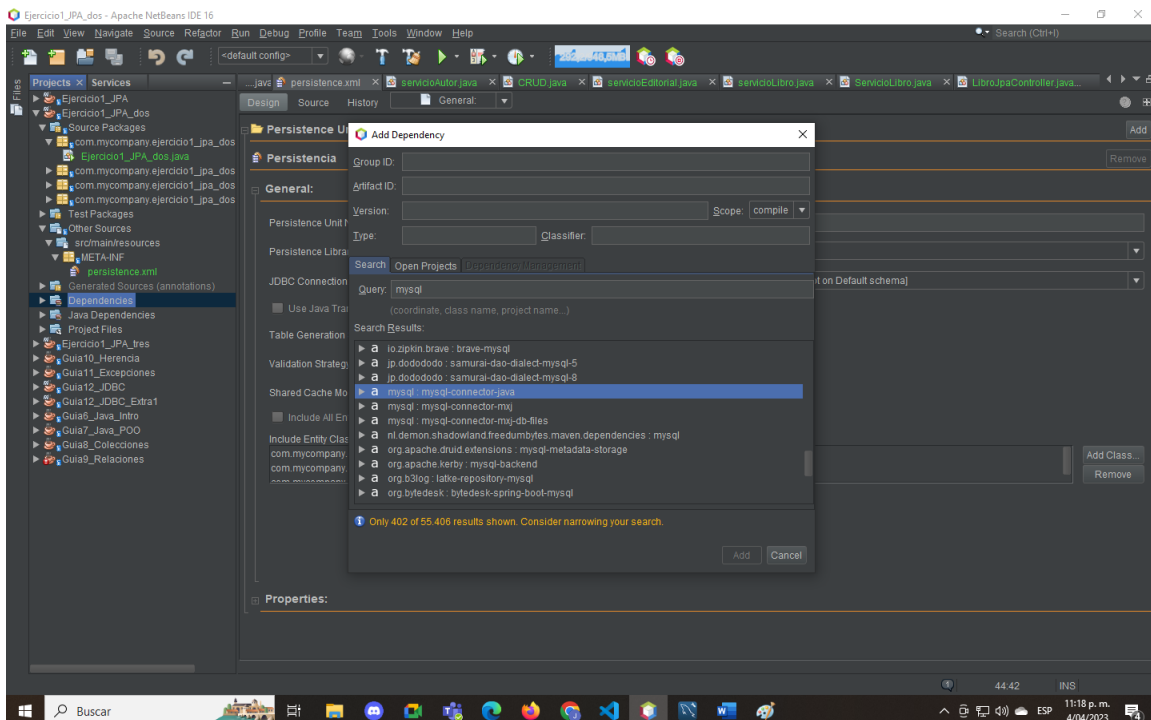


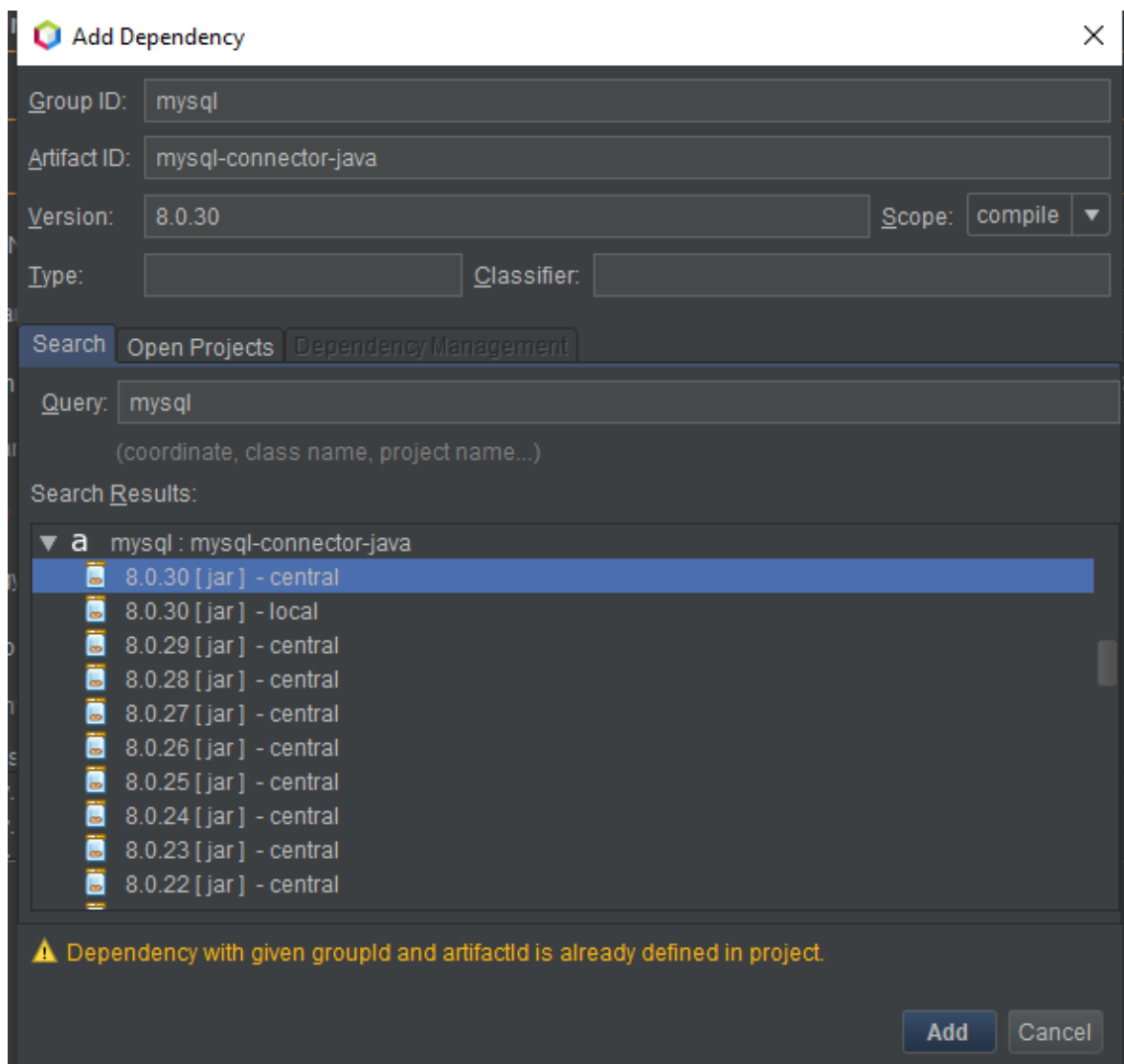
6. Asignación de dependencias

Tenemos una carpeta llamada **Dependencias**, al dar click derecho sobre ella, debemos elegir la opción **Add Dependency**.

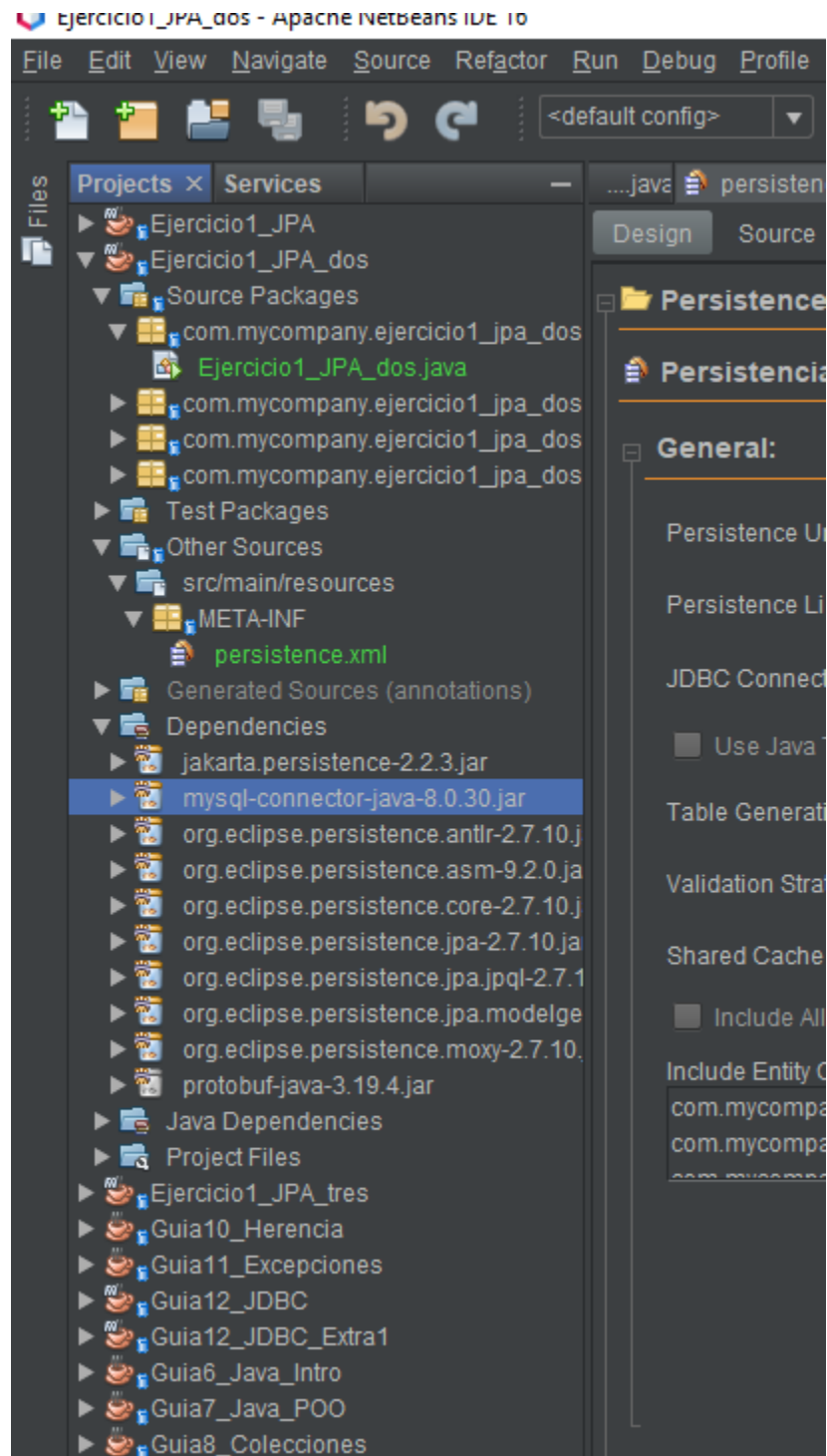


Luego, se genera otra ventana en donde debemos traer los conectores. Este paso se debe repetir cada que se cree un nuevo proyecto. Allí, en la parte central de la ventana aparece la opción **Query**; escribimos la palabra **mysql** en minúscula y bajamos la barra hasta encontrar el conector **mysql:mysql-connector-java**. Desplegamos las opciones que se desprenden del conector, seleccionamos el conector **8.0.30 [jar] central** y finalizamos con click en la opción **Add**.



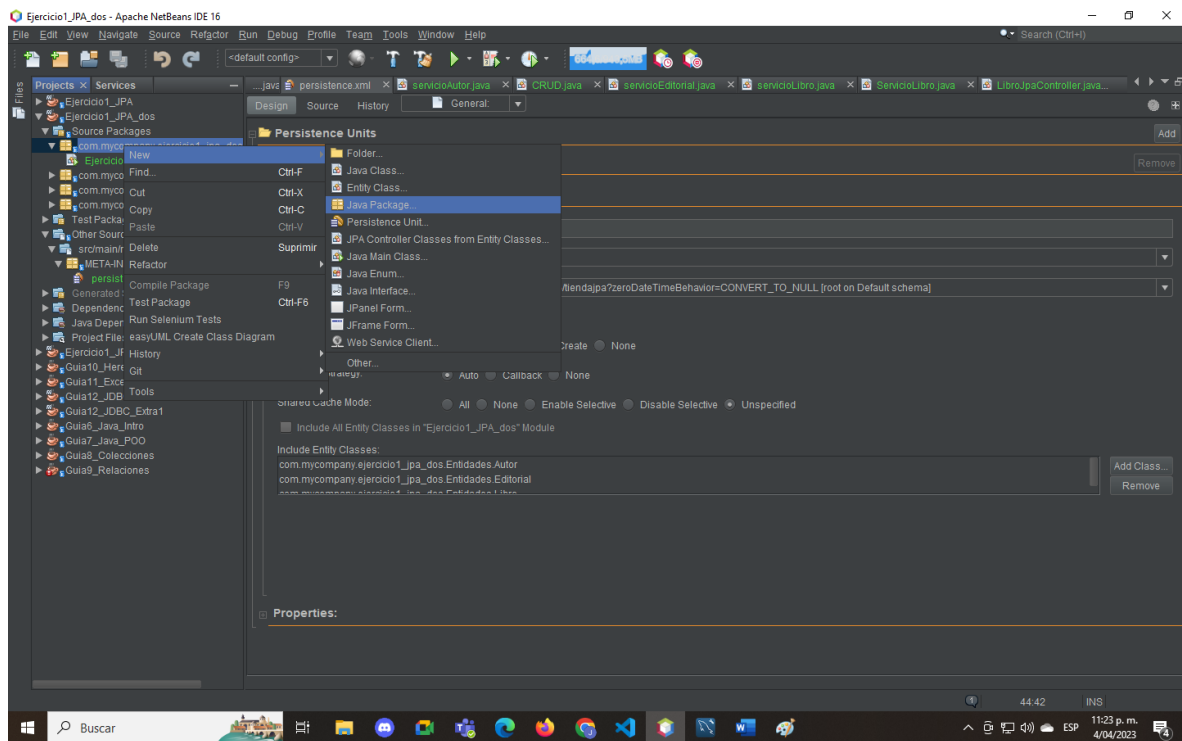


Luego, deberíamos observar el conector en la lista de dependencias.

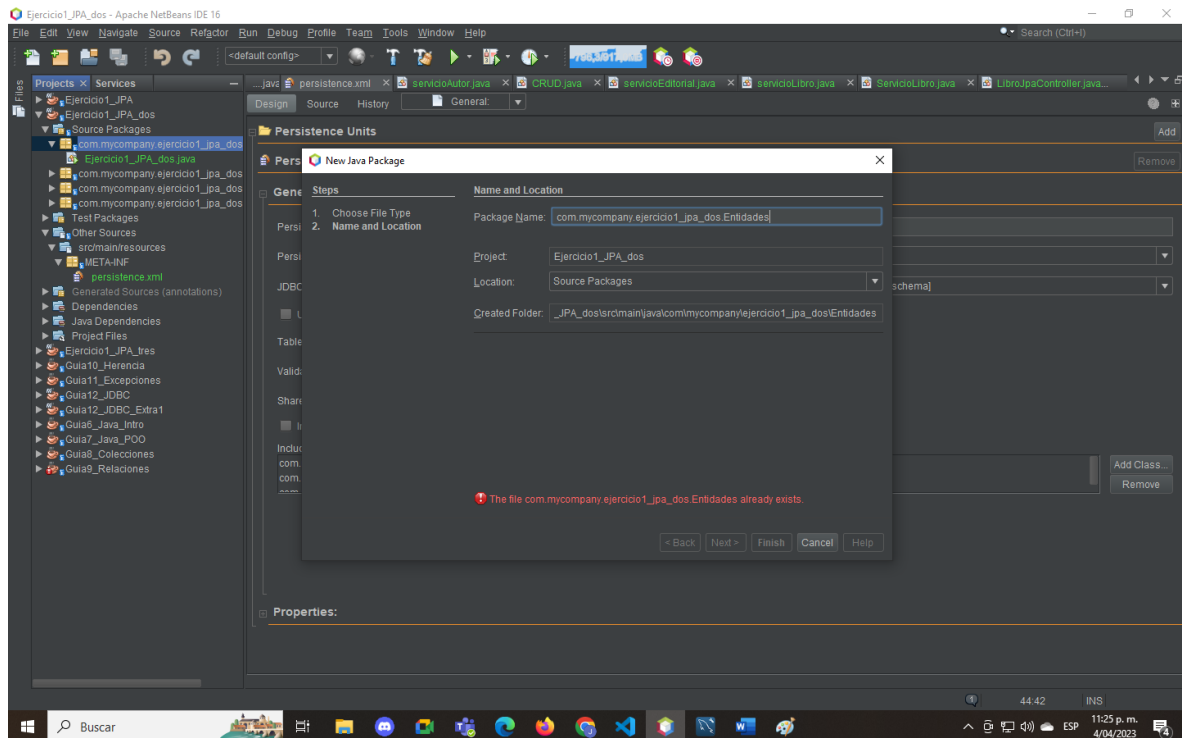


7. Creación de Paquete Entidades

Sobre nuestra **carpeta main**, damos click derecho para luego seleccionar la opción **new** y posteriormente elegir **Java Package**.

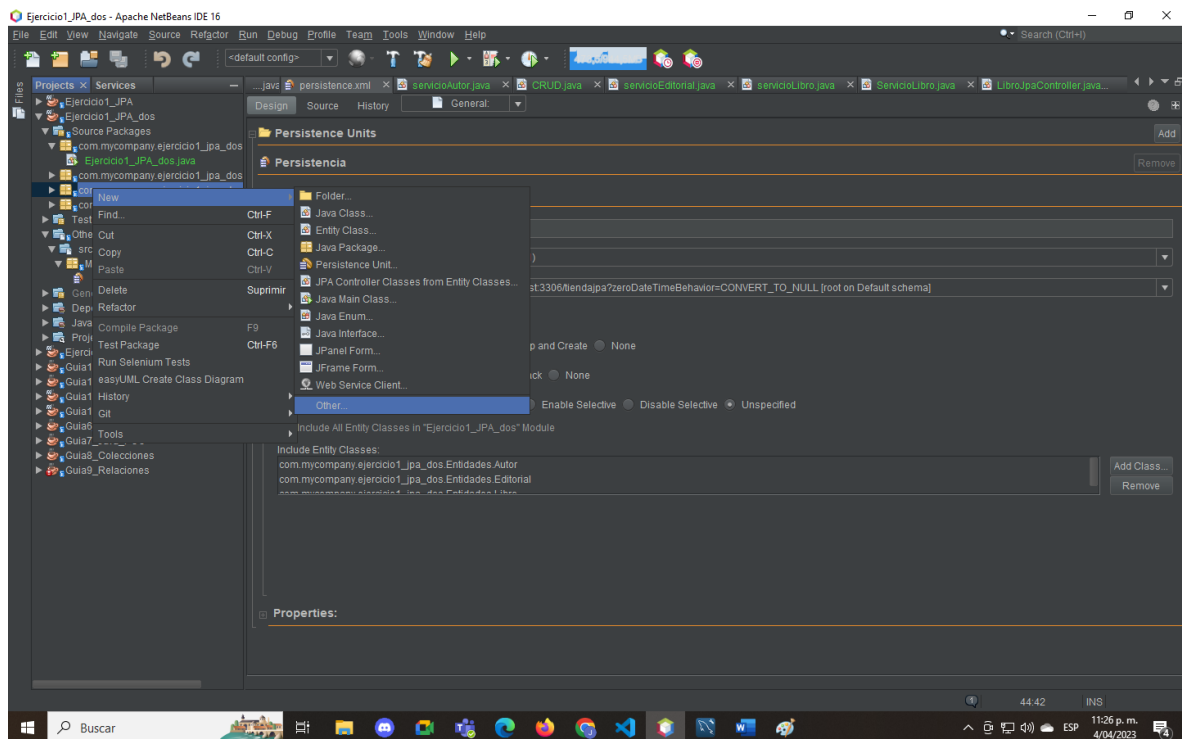


Luego, se despliega otra ventana en la que asignan un nombre, en mi caso, algo que resulte familiar, así que le digo que será mi paquete Entidades.

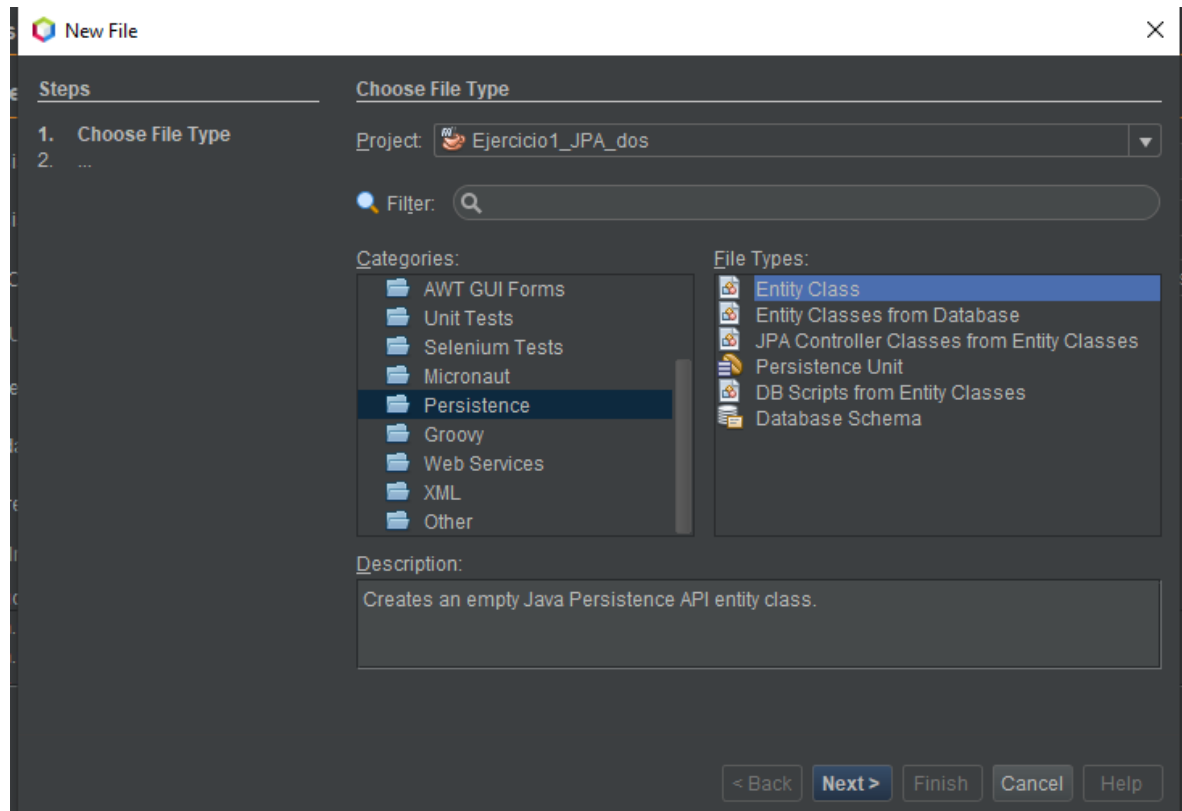


8. Creación de Entity Class

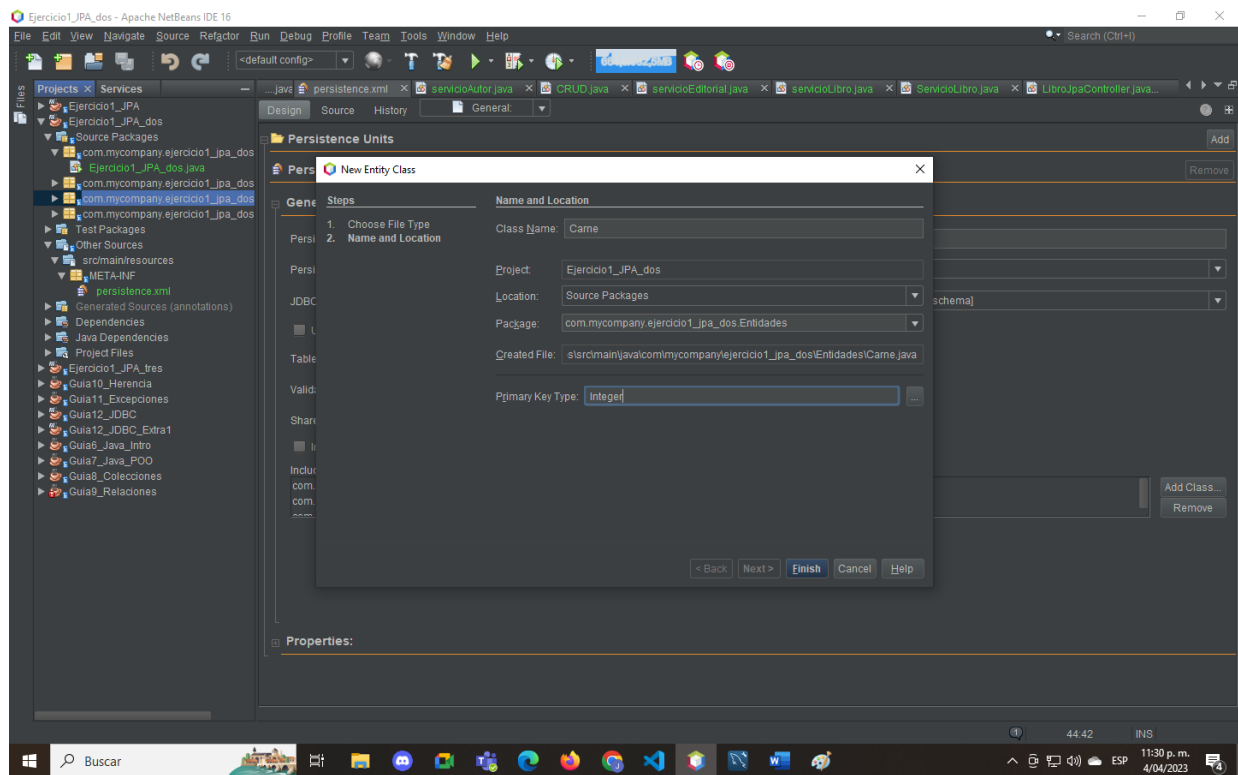
Sobre el paquete de entidades que acabamos de crear, damos click derecho y seleccionamos la opción **new**. Debemos crear **Entity Class**, así que si es la primera vez que las creamos, debemos ir a la opción **others**.



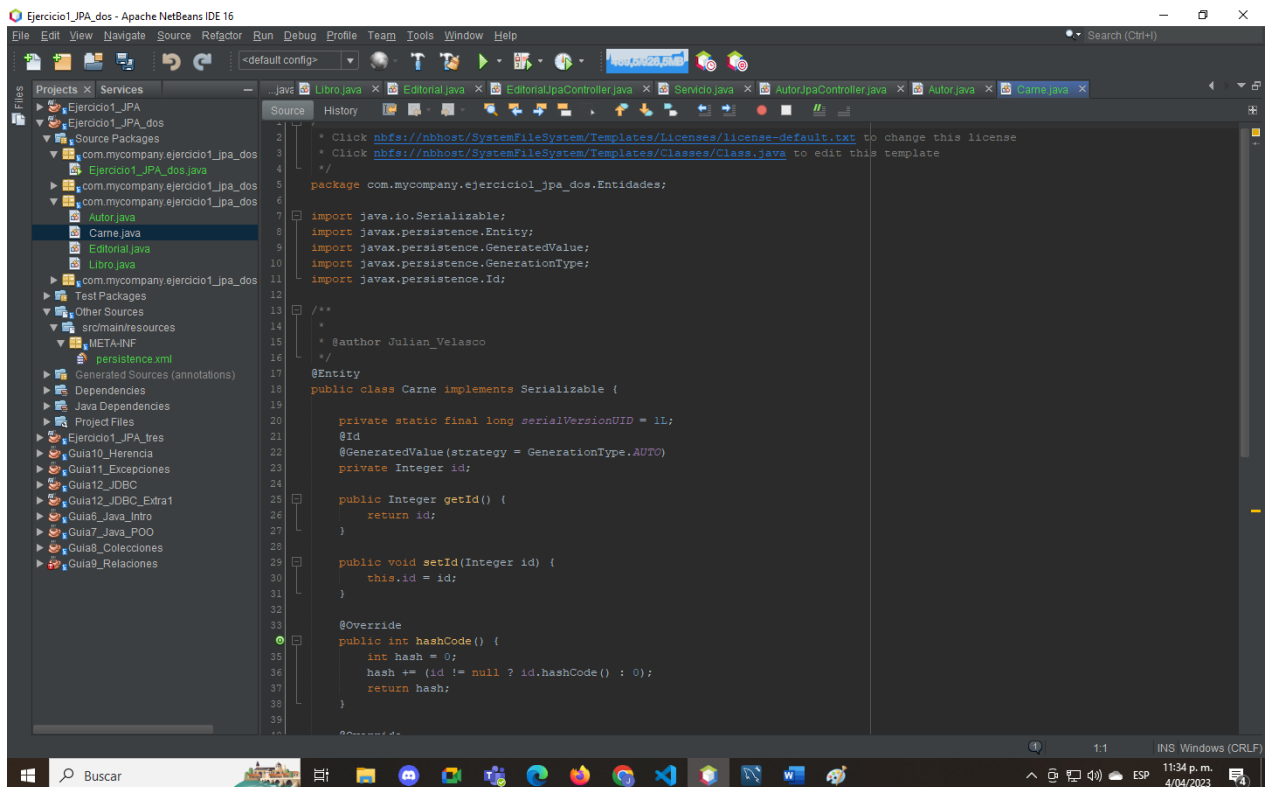
En la opción **Categories**, seleccionamos nuevamente **Persistence** y en **File Types** ahora seleccionamos la opción **Entity Class**. Finalizamos con click en la opción **Next**.

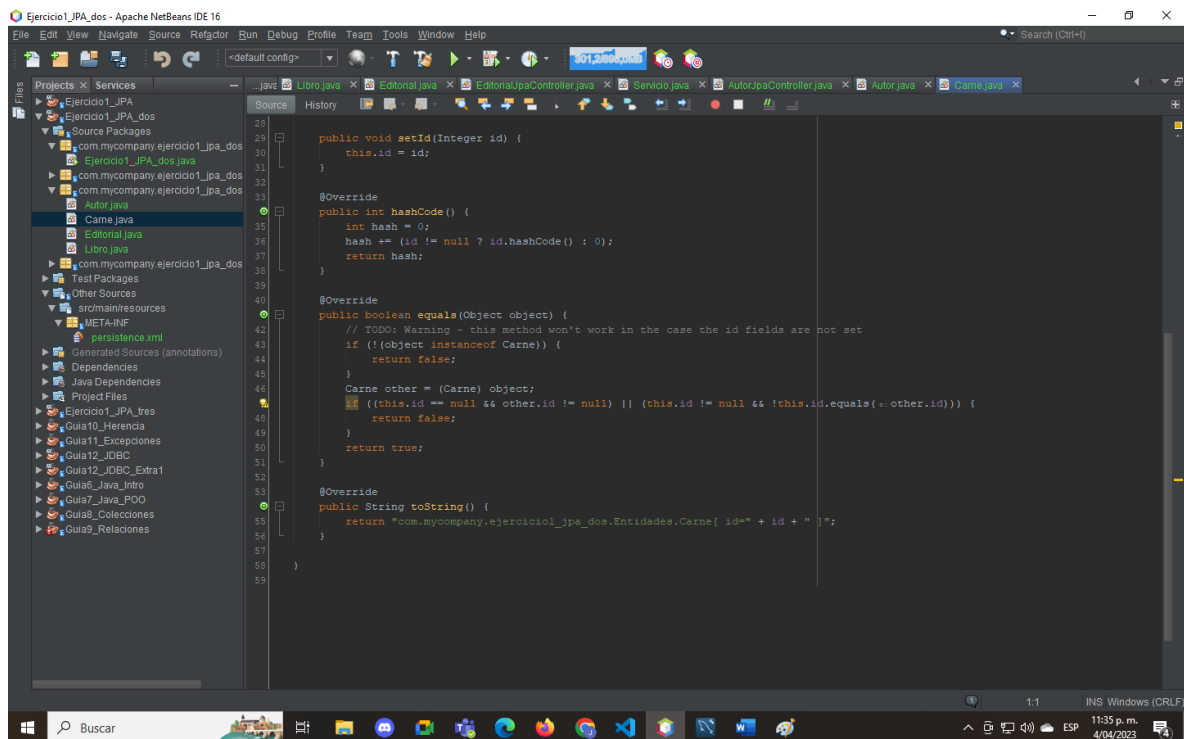


Se abre una nueva ventana en donde le damos un nombre a nuestra clase. Luego algo muy importante, la última opción dice **Primary Key Type**; procedemos a cambiar el tipo Long que aparece de fábrica por **Integer** o **String** según sea el caso; para el ejercicio 1 de la guía JPA podemos utilizar como **Integer** todas nuestras clases ya que son id. Finalizamos con click en la opción **Next**.

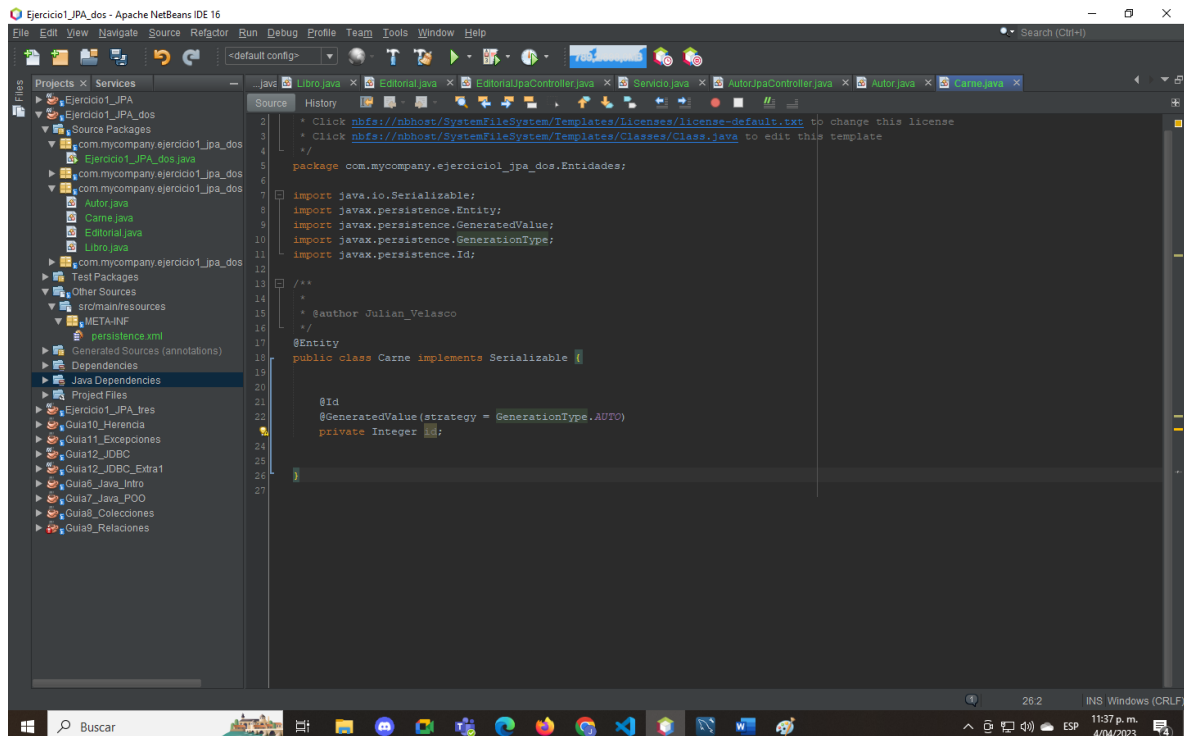


Automáticamente genera una clase con métodos estándar y la llave. (A continuación, un ejemplo)





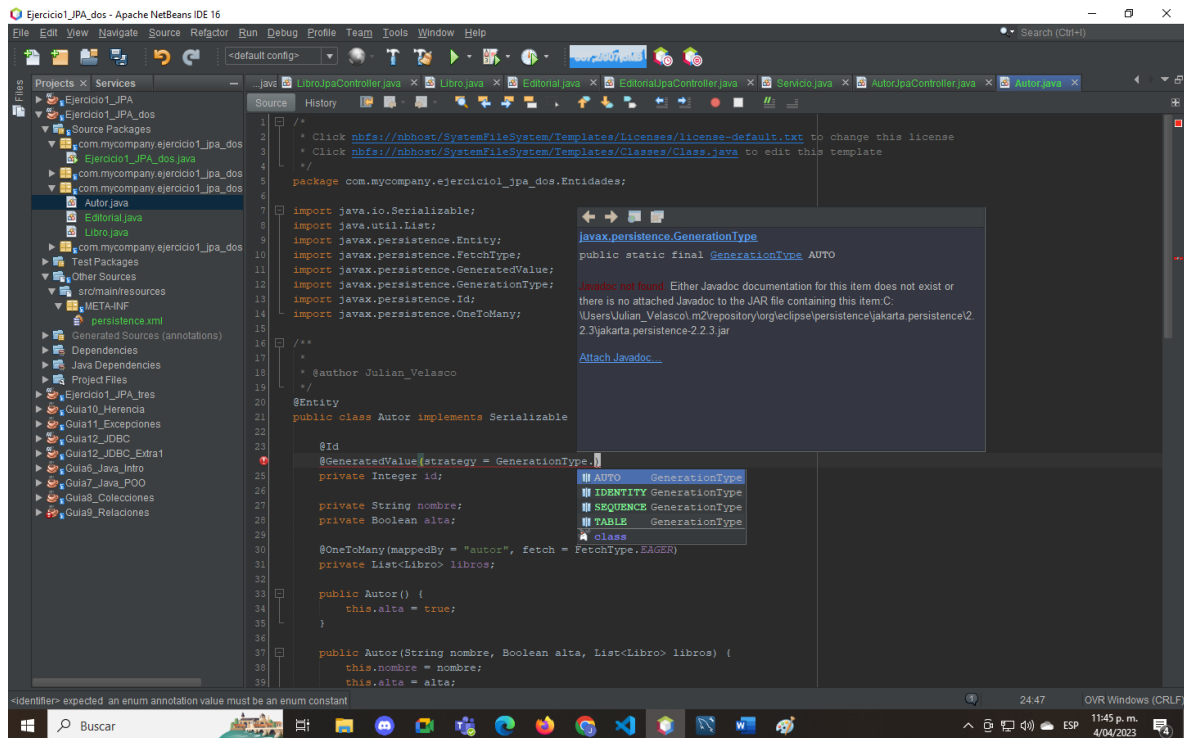
Si ustedes desean, dejan esos métodos. Algunos, por sugerencia, los borramos, dejando solo la asignación de la llave y creamos luego los atributos en la clase.



Si lo desean, cuando tengan la clase de esta manera, pueden traer sus atributos y plasmar las relaciones por cada clase.

Cuando tienen su llave, pueden elegir la estrategia de generación de la llave, borrando la palabra **AUTO**, luego damos, dejando el cursor sobre el punto, las teclas control y barra espaciadora, para

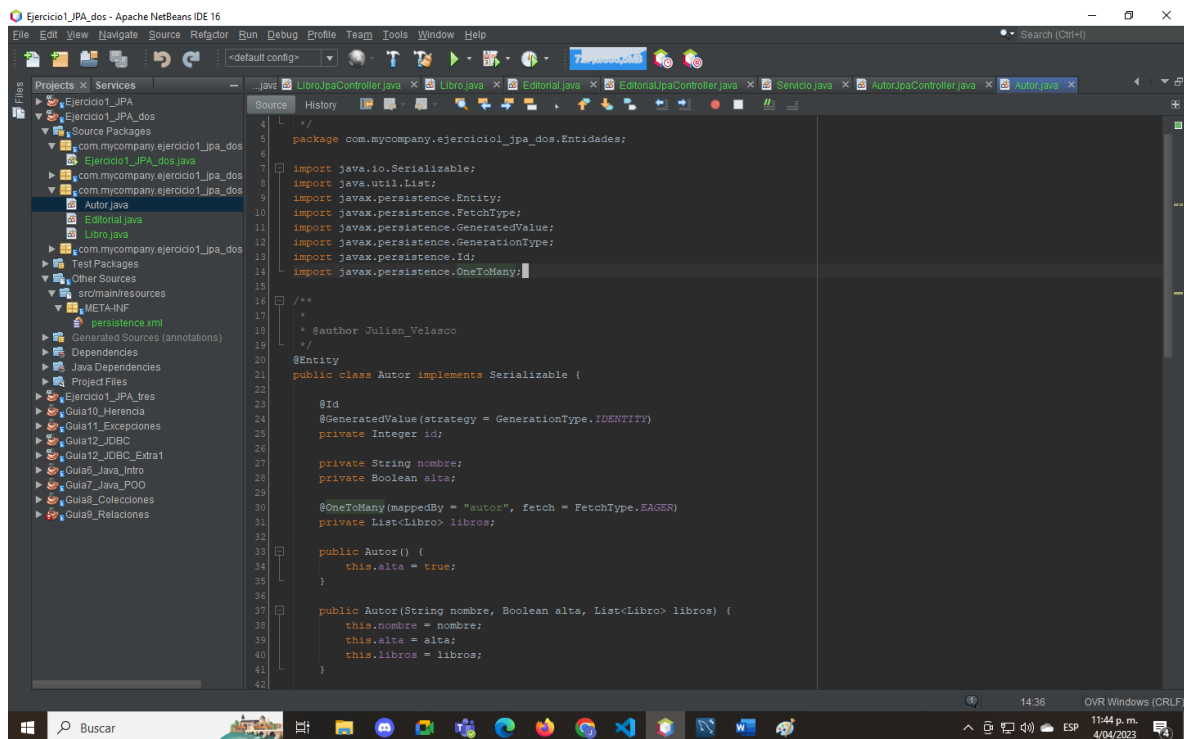
que se despliegan las diferentes estrategias. En mi caso selecciono **IDENTITY**, que me permite crear una llave automática que se autoincrementa.



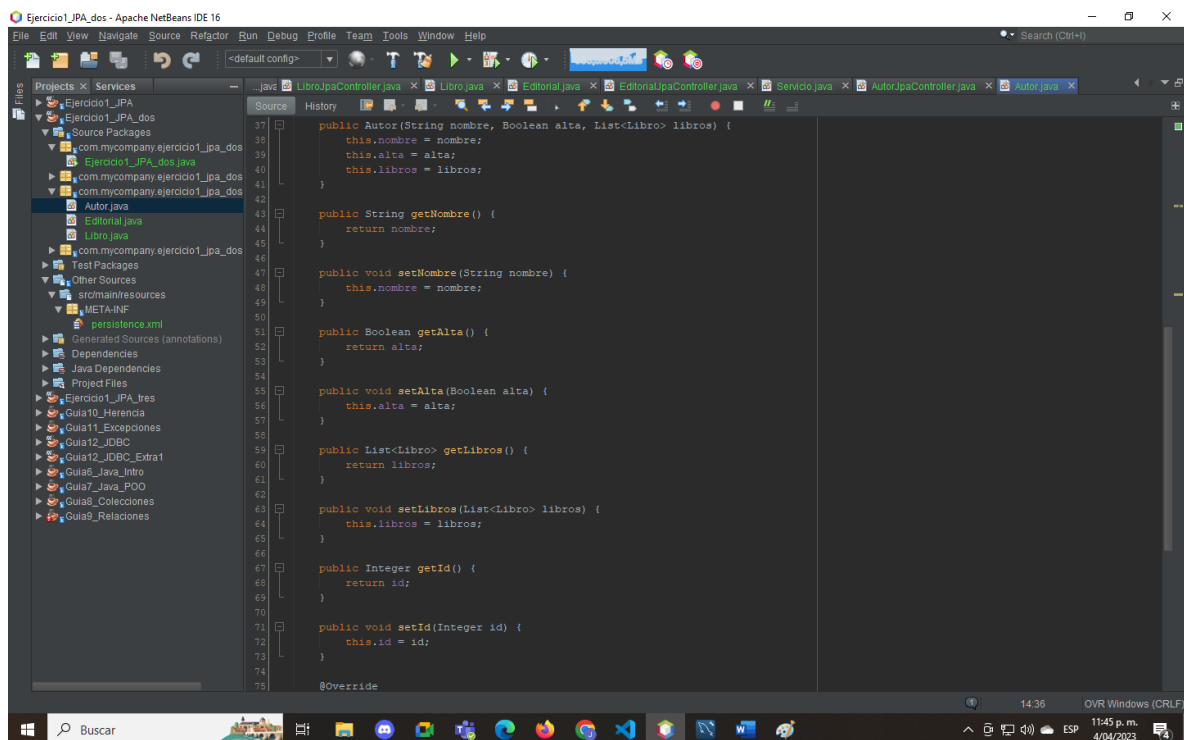
Les enviaré a continuación el ejemplo con la clase autor. En la misma se reportan los atributos y se plantean las relaciones, en este caso, el ejemplo lo haremos con la clase Autor, que tiene como atributos una llave que es el ID, un nombre y un Boolean denominado alta que nos permite saber cuando una entidad ha sido borrada.

Adicional, tenemos otro atributo con una anotación, **@OneToMany**, que indica que un Autor puede tener muchos libros.

mappedBy = "autor", que sigue a esta anotación, es un mapeo que elimina tablas intermedias que se crean, así, en el **Workbench**, al final solo tendremos las tablas solicitadas. Mapeamos en autor porque es la clase en donde nos encontramos; así mismo, si estuviéramos en la clase Editorial, mapearíamos **mappedBy = "editorial"**, siguiendo la misma praxis.



```
4  package com.mycompany.ejercicio1_jpa_dos.Entidades;
5
6
7  import java.io.Serializable;
8  import java.util.List;
9  import javax.persistence.Entity;
10 import javax.persistence.FetchType;
11 import javax.persistence.GeneratedValue;
12 import javax.persistence.GenerationType;
13 import javax.persistence.Id;
14 import javax.persistence.OneToMany;
15
16 /**
17  *
18  * @author Julian_Velasco
19  */
20 @Entity
21 public class Autor implements Serializable {
22
23     @Id
24     @GeneratedValue(strategy = GenerationType.IDENTITY)
25     private Integer id;
26
27     private String nombre;
28     private Boolean alta;
29
30     @OneToMany(mappedBy = "autor", fetch = FetchType.EAGER)
31     private List<Libro> libros;
32
33     public Autor() {
34         this.alta = true;
35     }
36
37     public Autor(String nombre, Boolean alta, List<Libro> libros) {
38         this.nombre = nombre;
39         this.alta = alta;
40         this.libros = libros;
41     }
42 }
```



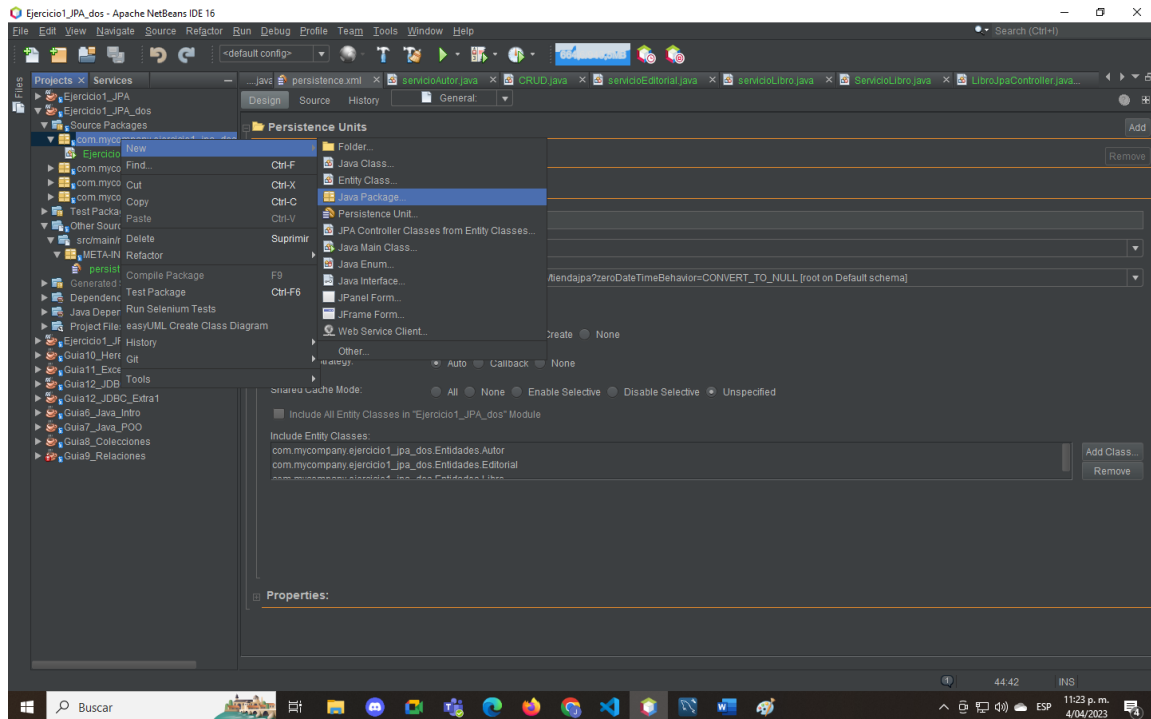
```
37     public Autor(String nombre, Boolean alta, List<Libro> libros) {
38         this.nombre = nombre;
39         this.alta = alta;
40         this.libros = libros;
41     }
42
43     public String getNombre() {
44         return nombre;
45     }
46
47     public void setNombre(String nombre) {
48         this.nombre = nombre;
49     }
50
51     public Boolean getAlta() {
52         return alta;
53     }
54
55     public void setAlta(Boolean alta) {
56         this.alta = alta;
57     }
58
59     public List<Libro> getLibros() {
60         return libros;
61     }
62
63     public void setLibros(List<Libro> libros) {
64         this.libros = libros;
65     }
66
67     public Integer getId() {
68         return id;
69     }
70
71     public void setId(Integer id) {
72         this.id = id;
73     }
74
75     @Override
```

Así, solo bastaría realizar **constructor vacío**, en el que aprovechamos para instanciar alta en **TRUE**, ya que lo solicita la consigna, crear el **constructor con parámetros**, en el **que no vamos a seleccionar la llave** debido a que al seleccionar **IDENTITY**, se crea automáticamente. Finalmente, creamos **Getters, Setters y toString**, en donde si creamos los correspondientes a la llave, que solo se retira del constructor con parámetros.

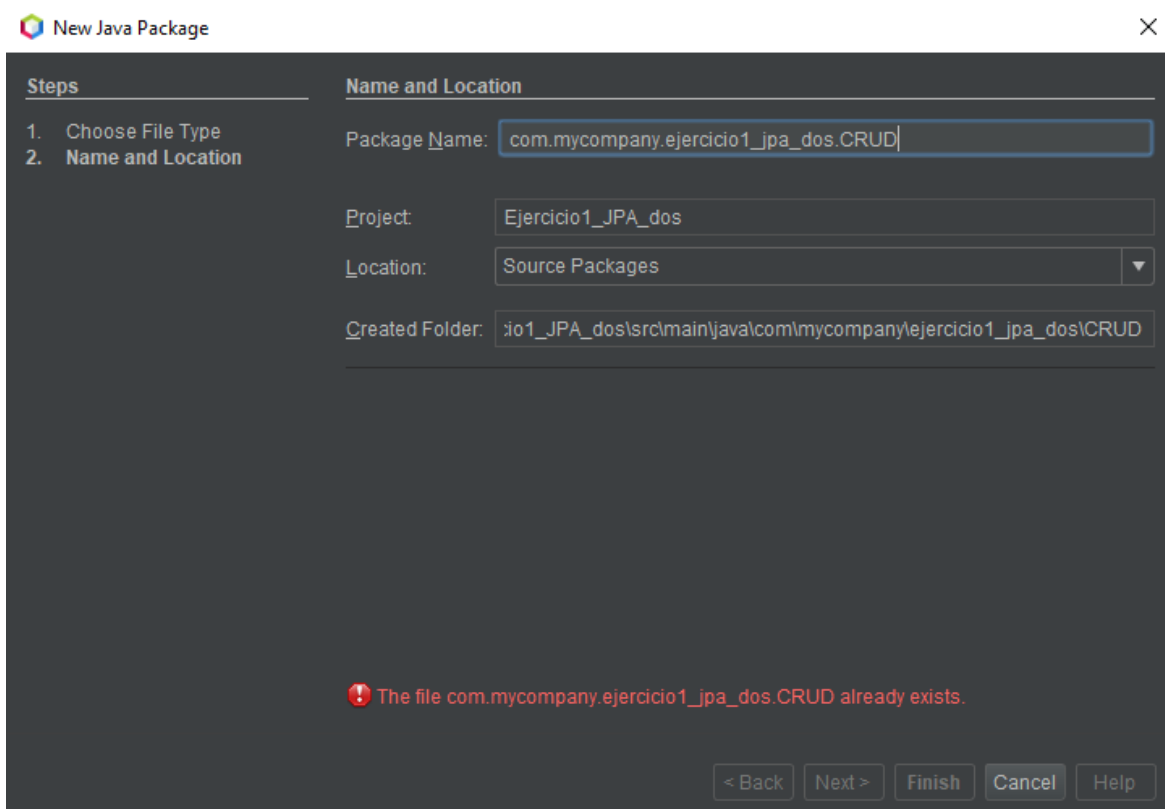
De la misma manera, pueden proceder con las demás clases que deban crear, **repitiendo el paso 8 en cada Entidad**.

9. Creación de paquete CRUD o Controladora

Sobre nuestra **carpeta main**, damos click derecho para luego seleccionar la opción **new** y posteriormente elegir **Java Package**.

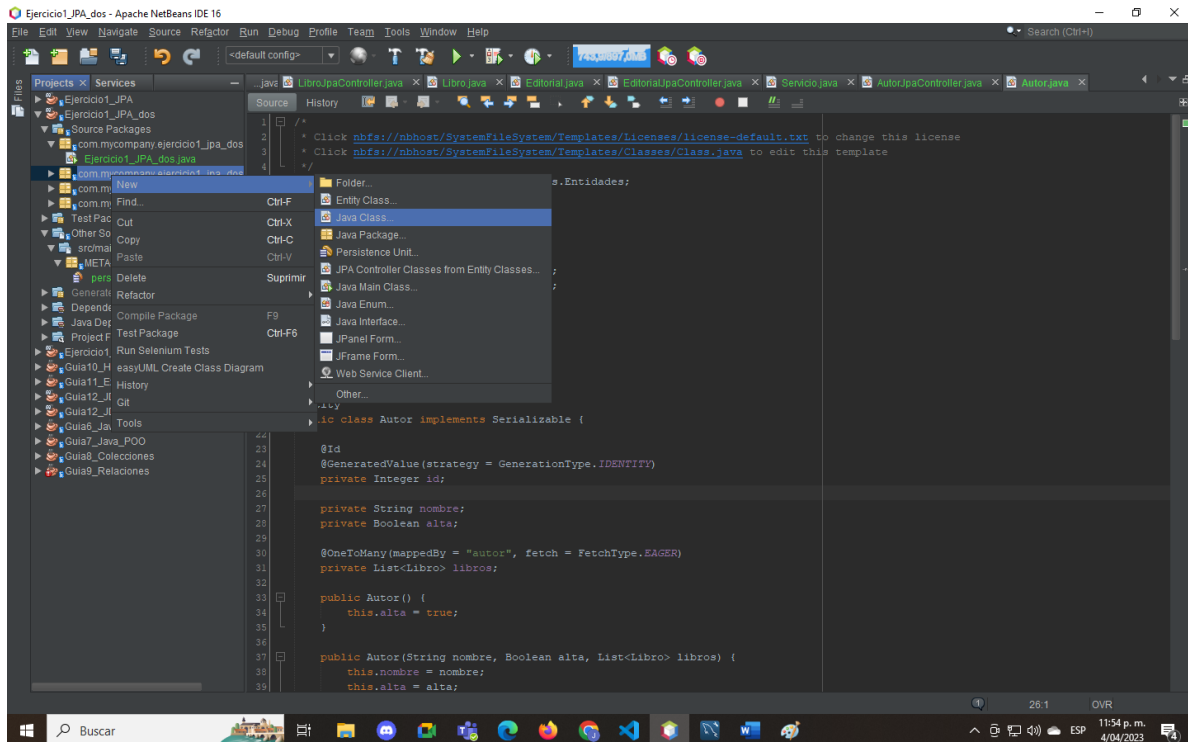


Luego, se despliega otra ventana en la que asignan un nombre, en mi caso, algo que resulte familiar, así que le digo que será mi paquete **CRUD**, aunque algunos lo llaman **Controladora**.

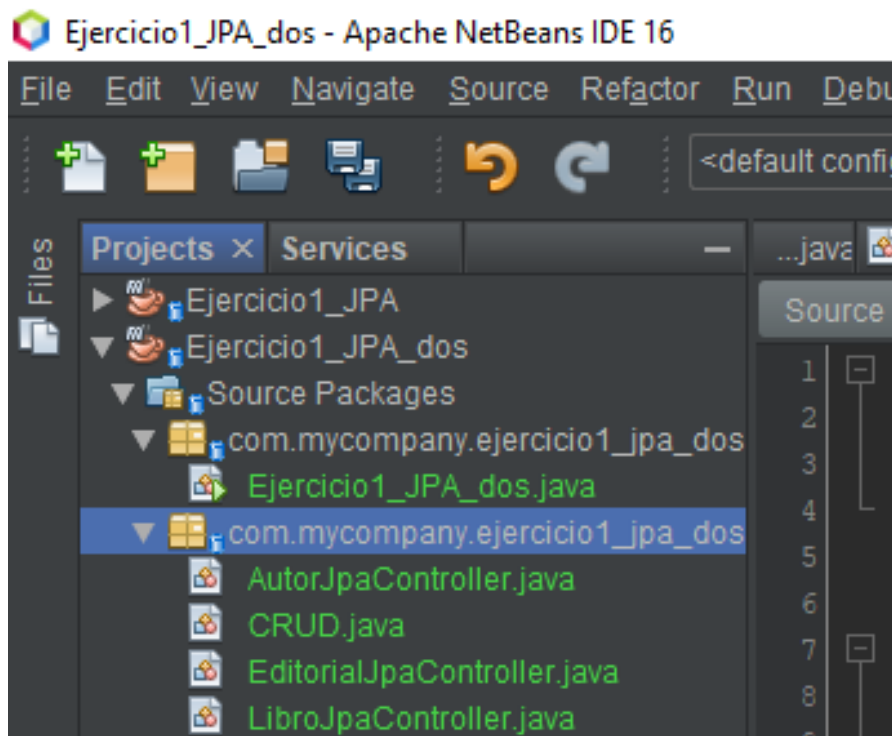


10. Creación de clases CRUD o Controladoras

Sobre el paquete **CRUD** o **Controladora** que acabamos de crear, damos click derecho, seleccionamos la opción **new** y posteriormente la opción **Java Class**. Estas son clases normales.



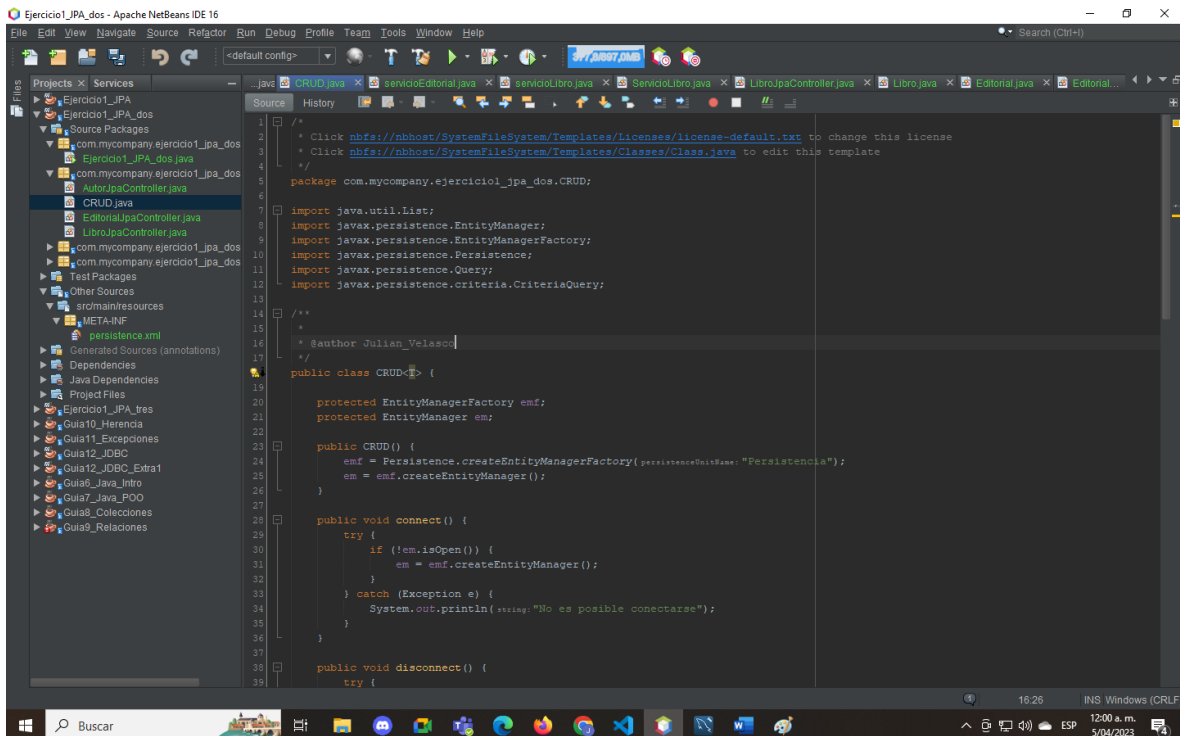
Luego despliega la ventana en donde podremos crear los nombres de cada una de ellas, La recomendación es crear una clase padre **CRUD** o **Controladora** y tres clases más, una para controlar cada Entidad. (A continuación, les comparto mi paquete CRUD y sus clases)



11. Clase CRUD o Controladora

En esta clase, una sugerencia es convertirla en una clase genérica.

public class CRUD<T>, que lo pueden observar en la definición de la clase, la convierte en una clase genérica, que te crea métodos que pueden ser utilizados por cualquier clase y objeto. (A continuación, el ejemplo con mi clase **CRUD**).



Como pueden observar, las líneas del comienzo crean unos elementos nuevos.

protected EntityManagerFactory emf;

protected EntityManager em;

EntityManagerFactory emf es una fábrica, como lo indica su nombre, que crea **EntityManager**, que es una interfaz que nos permite realizar la persistencia, es decir, interactuar con la base de datos, siendo los encargados de todas las transacciones.

Luego, en el constructor vacío instanciamos los **EntityManagerFactory** y los

EntityManager, así, sus clases hijas, que son las otras clases **CRUD**, una por cada entidad, van a poder heredarlos y poder realizar transacciones.

public CRUD() {

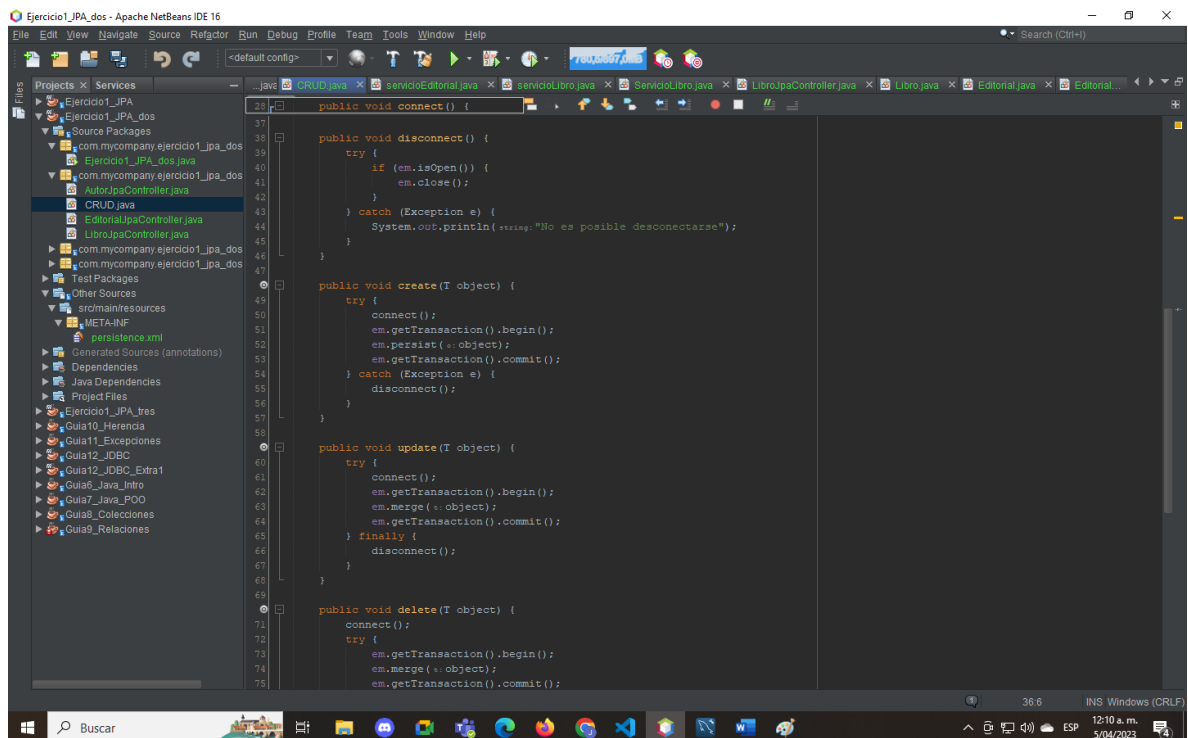
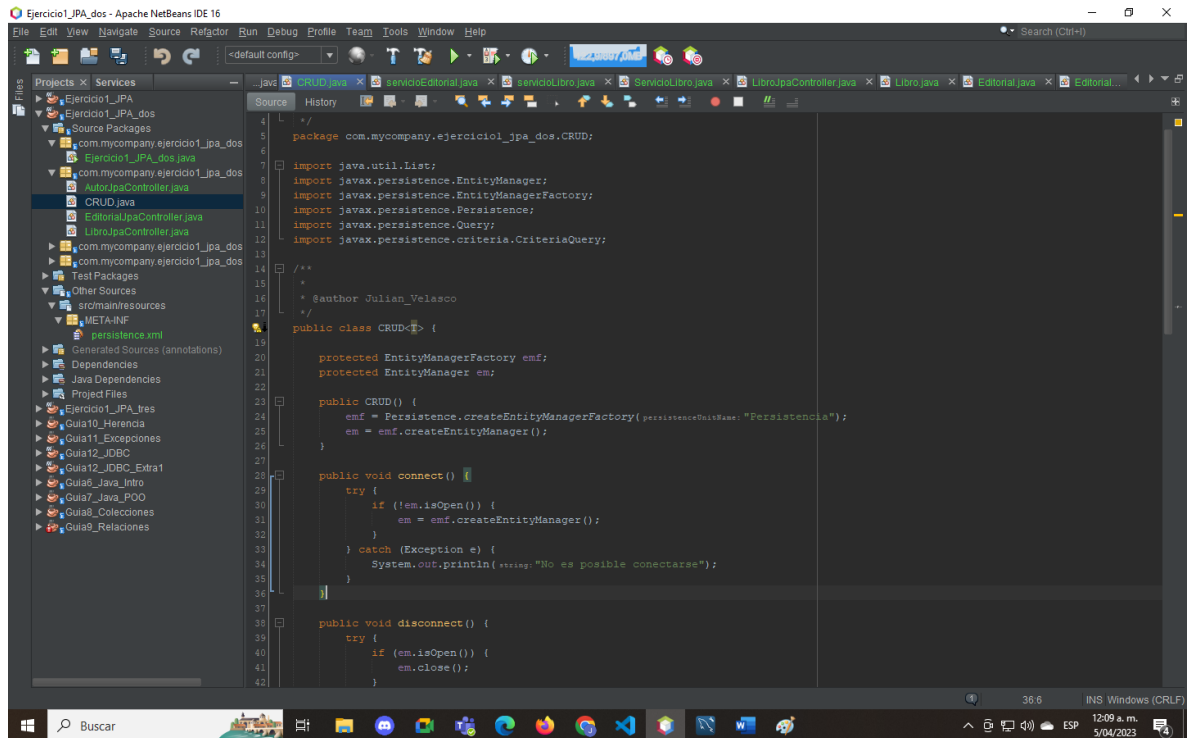
emf = Persistence.createEntityManagerFactory("Persistencia");

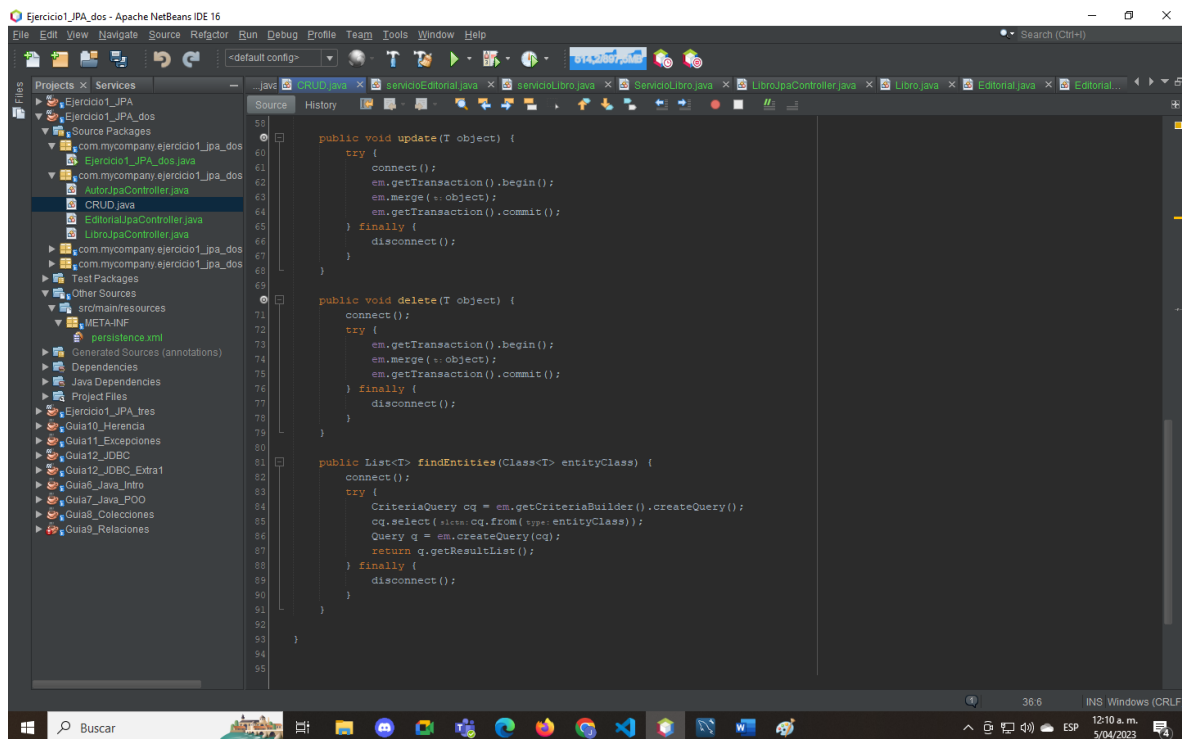
em = emf.createEntityManager();

}

Debemos tener cuidado en esta línea, la cual al final dentro de comillas lleva el nombre que hayan entregado a su unidad de persistencia. (Lo resaltamos en color azul)

emf = Persistence.createEntityManagerFactory("Persistencia");





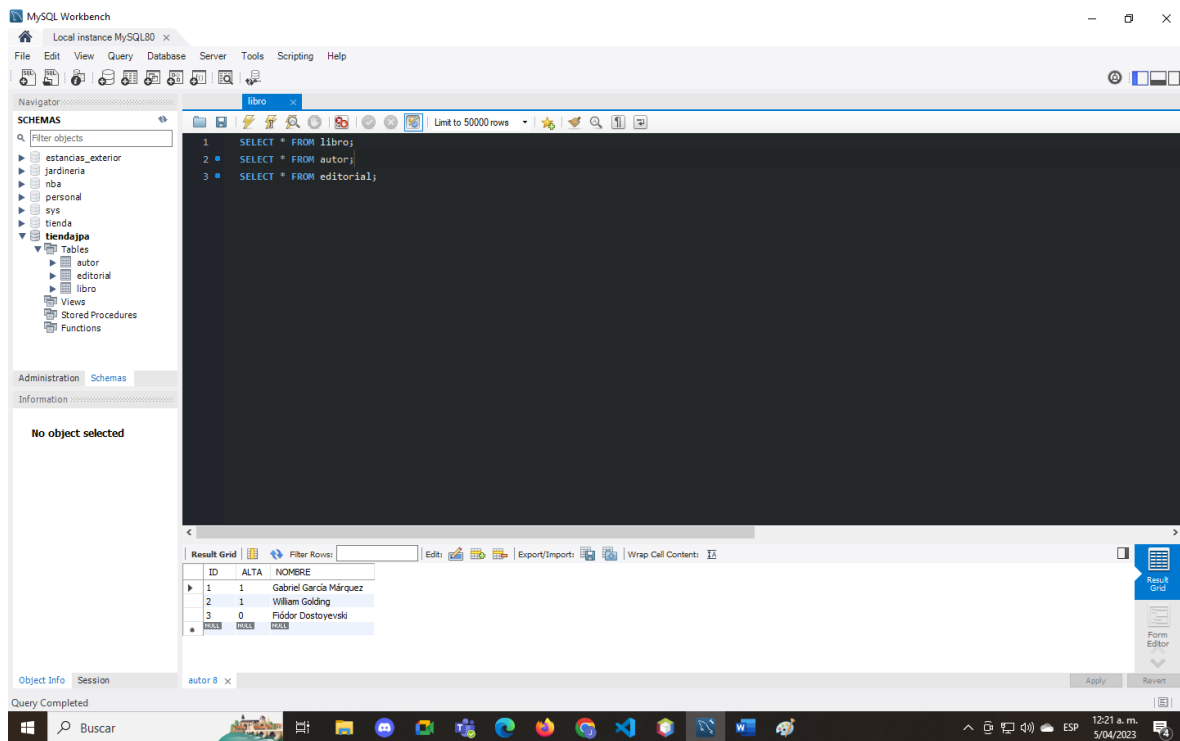
Al igual que en **JDBC**, debemos establecer métodos que nos permitan conectar con la base de datos, desconectar y demás, ya que como su nombre lo indica, **CRUD** está compuesto por las iniciales de **Create, Read, Update y Delete**.

Como pueden observar, los métodos al hacer parte de una clase genérica, tienen parámetros que indican que son genéricos, valga la redundancia, al igual que la clase, es decir, que te sirven para cualquier clase y objeto.

Los métodos se implementan dentro de un bloque Try – Catch, ya que como estamos trabajando con procesos latentes como conexión a bases de datos, debemos manejar excepciones.

Comenzamos con la conexión a la base de datos, luego, generamos el inicio de las transacciones, **em.getTransaction().begin()**; como se indicaba a través de los **EntityManager**, que llamé **em**; para luego llegar a una acción intermedia:

- En la creación: **em.persist(object)**;
- En el borrado: **em.merge(object)**; aunque normalmente se utiliza **em.remove(object)**; en el caso particular utilizamos **merge** porque maneja una sincronización, es decir; nos permite administrar o actualizar entidades desconectadas, mientras que **remove** no lo permite. En nuestro ejercicio particular, consideramos adecuado que, si deseamos eliminar una entidad, no tengamos que borrar los libros asociados a la entidad. Es decir, si borramos un autor o una editorial no se borren los libros asociados, sino que cambie el estado de alta a FALSE según solicita la consigna y conserve los datos, lo cual es muy coherente; además, que al desplegar la tabla de libro, no me haya eliminado el libro, sino que en la columna editorial me reporte un valor nulo.



Por ejemplo, en mi tabla autor, podemos observar en **Workbench**, que borré un autor, cambiando **alta** a **0**, lo que significa que los autores que lo tienen en **1** están **activos** y el que esta en **0** está **inactivo**; conservo sus datos, pero está inactivo.

No object selected

ISBN	ALTA	AÑO	EJEMPLARES	EJEMPLARES PRESTADOS	EJEMPLARES RESTANTES	TÍTULO	AUTOR_ID	EDITORIAL_ID
1	1	1967	200	190	10	Cien Años de Soledad	1	1
2	1	1954	150	100	50	El señor de Las Moscas	2	2
3	1	1866	100	70	30	Crimen y Castigo	NULL	3
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Ahora, al desplegar la tabla libro, observamos que me mantiene el libro pero en la columna autor me aparece null porque fue desasociado. Esto lo podemos hacer utilizando el merge y no el remove.

- Finalmente, en la edición: **em.merge(object)**; también se utiliza el merge.

Luego de estos pasos intermedios durante las transacciones:}

em.getTransaction().commit(); utilizamos el commit para confirmar la transacción y luego procedemos a desconectarnos de la base de datos.

Creamos, además, un método muy importante, que a través de una consulta en **lenguaje JPQL**, que maneja clases, nos encuentra toda la información de una determinada clase. Como pueden observar es el último método de la clase **CRUD** o **Controladora**

```

public List<T> findEntities(Class<T> entityClass) {
    connect();
    try {
        CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
        cq.select(cq.from(entityClass));
        Query q = em.createQuery(cq);
        return q.getResultList();
    } finally {
        disconnect();
    }
}

```

Este método crea un objeto de tipo **CriteriaQuery**, que, a través de mi **EntityManager**, indica que se va a crear una consulta.

cq.select(cq.from(entityClass)); esta es la consulta, le digo que me traiga todo lo que tiene la clase, al ser genérica, la heredan las clases hijas y me traerá lo que tiene cada clase hija derivada de una entidad.

Query q = em.createQuery(cq); establecemos la consulta

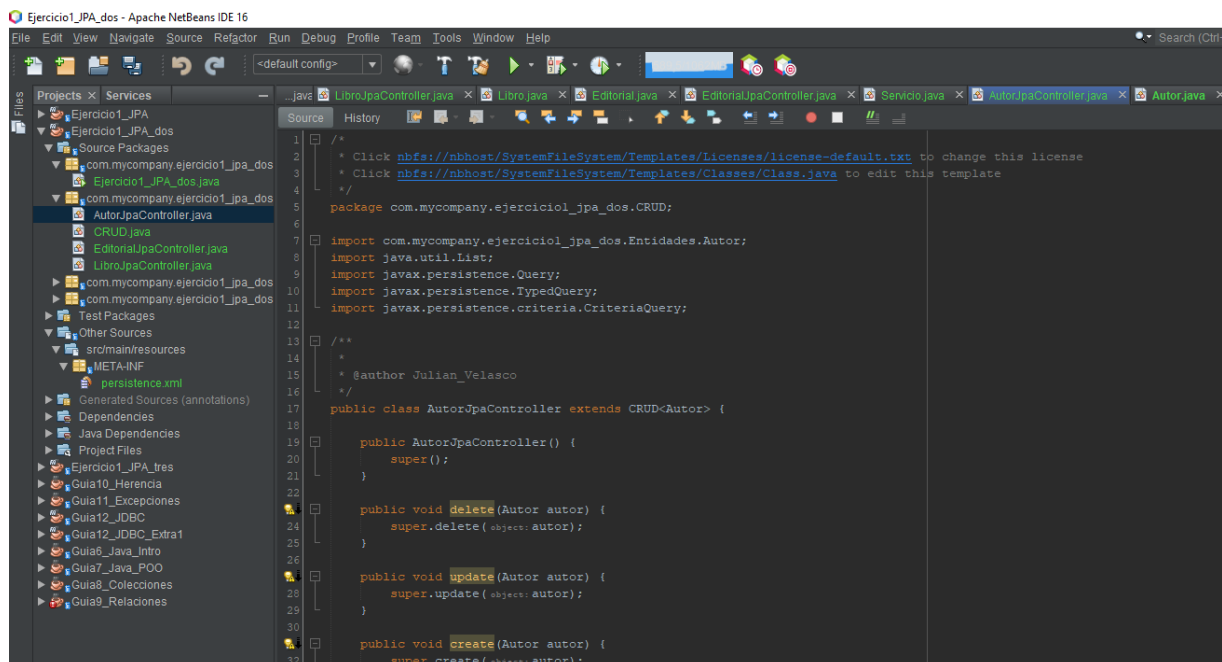
return q.getResultList(); obtenemos un resultado, en este caso, una lista ya que mi método fue creado de tipo Lista genérica.

12. Creación clases hijas controladoras

En cada clase hija, vamos a utilizar la herencia. Con el super, vamos a heredar de la clase padre **CRUD** o **Controladora** (Ejemplo con mi clase **AutorController** o **AutorCRUD**).

Debemos tener presente que heredan de una clase genérica, tipo T; por lo cual se debe indicar que tipo de datos tendremos al heredar de la clase padre.

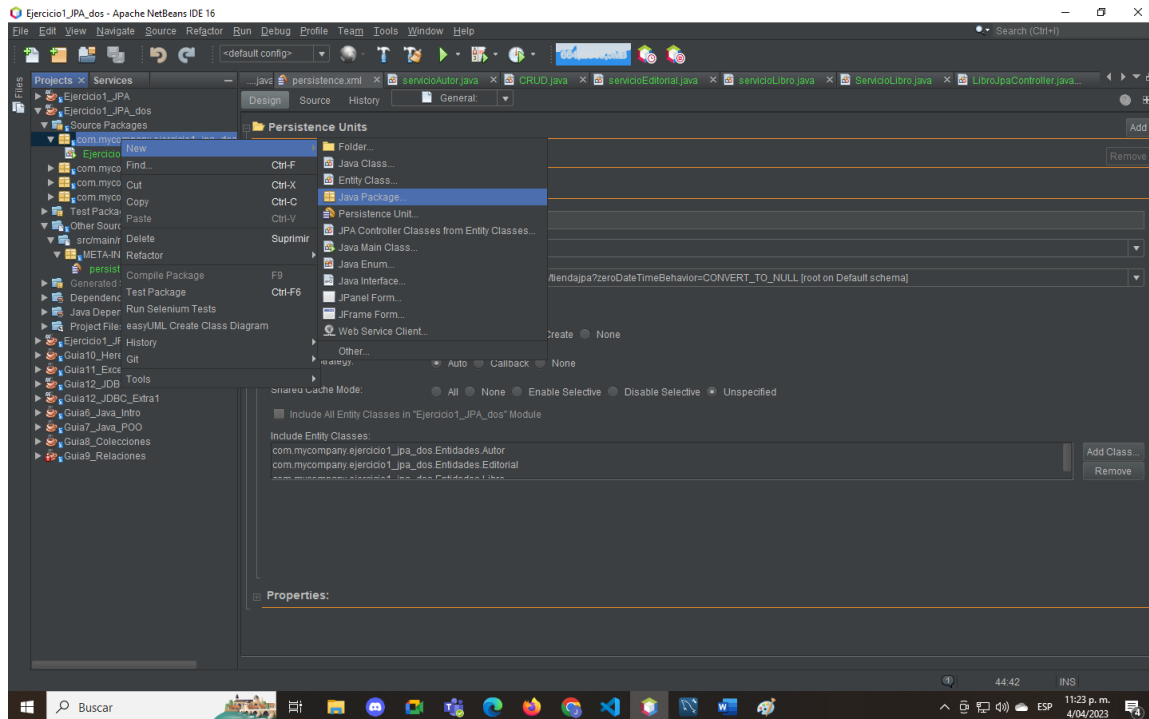
public class AutorJpaController extends CRUD<Autor> (Esto al ser mi clase Autor)



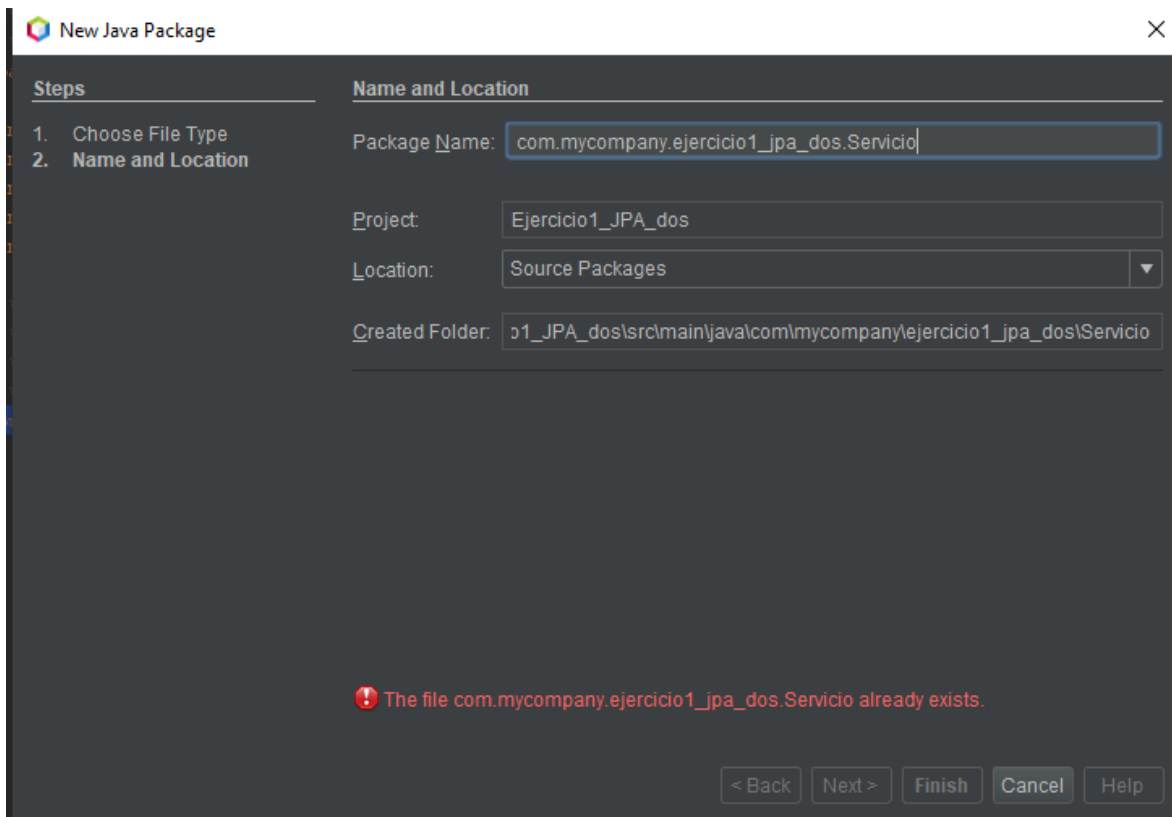
Así, repetimos el paso 12 para cada una de las clases controladores, una por cada entidad.

13. Creación de paquete Servicio

Sobre nuestra **carpeta main**, damos click derecho para luego seleccionar la opción **new** y posteriormente elegir **Java Package**.

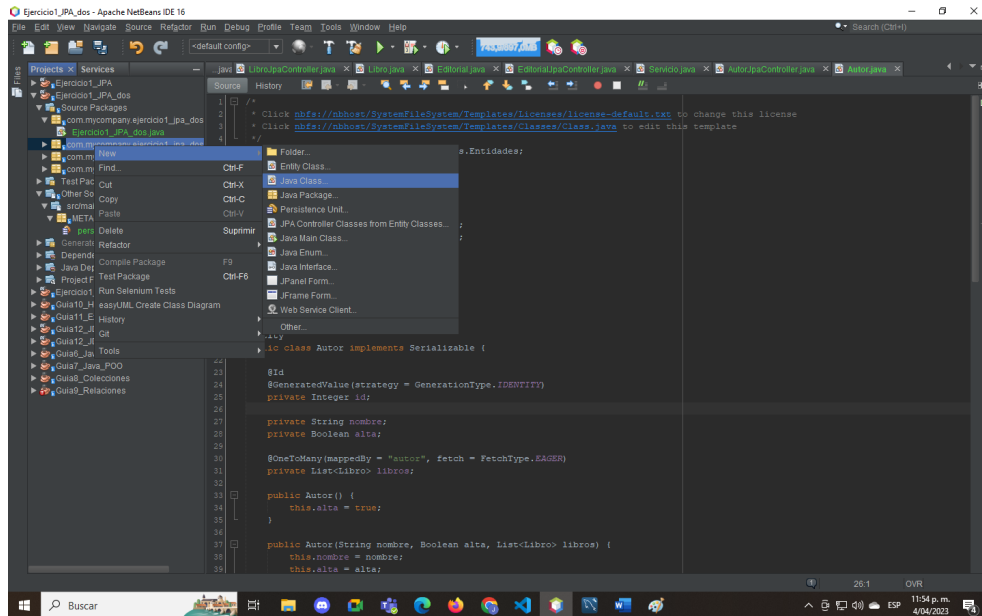


Luego, se despliega otra ventana en la que asignan un nombre, en mi caso, algo que resulte familiar, así que le digo que será mi paquete **Servicio**.

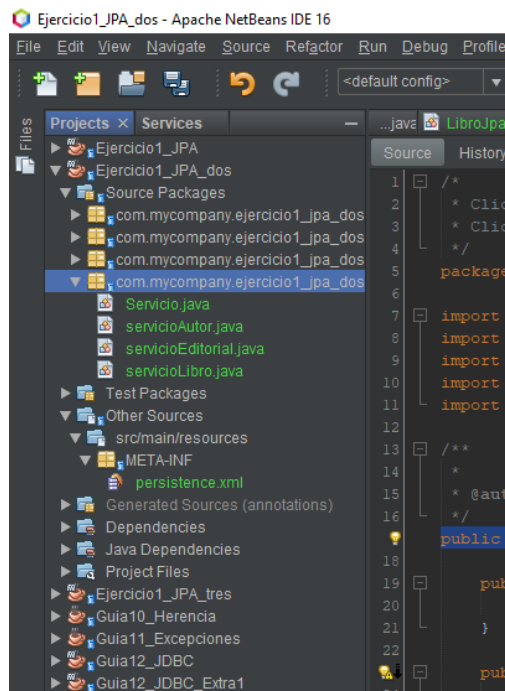


14. Creación de clases Servicio

Sobre el **paquete Servicio** que acabamos de crear, damos click derecho, seleccionamos la opción **new** y posteriormente la opción **Java Class**. Estas son clases normales.



Luego despliega la ventana en donde podremos crear los nombres de cada una de ellas, La recomendación es crear una clase servicio para en ella insertar un menú, ya que hay varias consultas por elaborar y tres clases más, una por cada entidad. (A continuación, les comparto mi paquete Servicio sus clases)



Así, las consultas y métodos de cada entidad quedan en un servicio independiente, el código será legible y prolijo.

Con esto, ya estamos listos para comenzar a implementar las consultas de la guía.