CURSO DE PROGRAMACIÓN FULL STACK

ACCESO A BASE DE DATOS DESDE JAVA: JPA

Objetivos de la Guía

En esta guía aprenderemos:

- Qué es JPA y cómo funciona.
- Mapear entidades y atributos.
- Mapear relaciones entre clases.
- Persistir, buscar, modificar y eliminar entidades de la base de datos desde Java
- Crear consultas a la base de datos desde Java

PERSISTENCIA EN JAVA CON JPA

JPA (Java Persistence API) es la propuesta estándar que ofrece Java para implementar un Framework **Object Relational Mapping** (ORM), que permite interactuar con la base de datos por medio de objetos, de esta forma. JPA es el encargado de **convertir los objetos Java en instrucciones para el Manejador de Base de Datos** (DBMS). El objetivo que persigue el diseño de esta API es no perder las ventajas de la orientación a objetos al interactuar con una base de datos (siguiendo el patrón de mapeo objeto-relacional).

Cuando empezamos a trabajar con bases de datos en Java utilizamos el API de JDBC el cual nos permite realizar consultas directas a la base de datos a través de consultas SQL nativas. JDBC por mucho tiempo fue la única forma de interactuar con las bases de datos, pero representaba un gran problema y es que Java es un lenguaje orientado a objetos y se tenían que convertir los atributos de las clases en una consulta SQL como SELECT, INSERT, UPDATE, DELETE, etc. Lo que ocasionaba un gran esfuerzo de trabajo y provocaba muchos errores en tiempo de ejecución, debido principalmente a que las consultas SQL se tenían que generar frecuentemente al vuelo.

JPA es una especificación, es decir, no es más que un documento en el cual se plasman las reglas que debe de cumplir cualquier proveedor que desee desarrollar una implementación de JPA, de tal forma que cualquier persona puede tomar la especificación y desarrollar su propia implementación de JPA. **Existen varios proveedores como lo son los siguientes:**

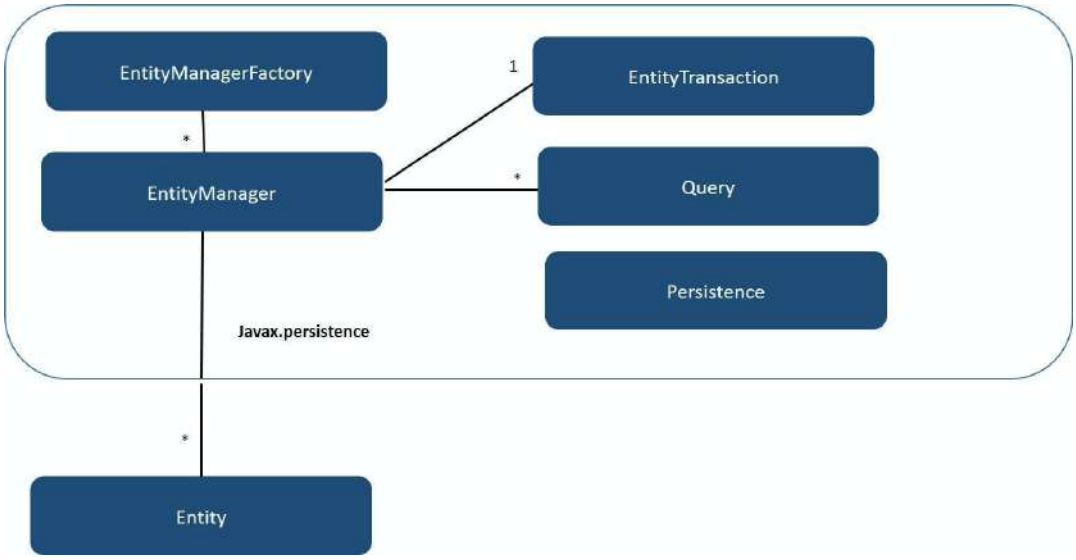
- Hibernate
- ObjectDB
- EclipseLink
- OpenJPA

PERSITENCIA DE OBJETOS

JPA representa una **simplificación del modelo de programación de persistencia**. La especificación JPA define explícitamente la correlación relacional de objetos, en lugar de basarse en implementaciones de correlación específicas del proveedor. JPA crea un estándar para la importante tarea de la correlación relacional de objetos mediante la **utilización de anotaciones o XML para correlacionar objetos con una o más tablas** de una base de datos. Para simplificar aún más el modelo de programación de persistencia:

- La API EntityManager puede actualizar, recuperar, eliminar o aplicar la persistencia de objetos de una base de datos.
- JPA proporciona un lenguaje de consulta, que amplía el lenguaje de consulta EJB independiente, conocido también como JPQL, el cual puede utilizar para recuperar objetos sin grabar consultas SQL específicas en la base de datos con la que está trabajando.
- El programador no necesita programar código JDBC ni consultas SQL.
- El entorno realiza la conversión entre tipos Java y tipos SQL.
- El entorno crea y ejecuta las consultas SQL necesarias.

ARQUITECTURA JPA - COMPONENTES



La arquitectura de JPA está diseñada para gestionar Entidades y las relaciones que hay entre ellas. A continuación, detallamos los principales componentes de la arquitectura

Entity: Clase Java simple que representa una fila en una tabla de base de datos con su formato más sencillo. Los objetos de entidades pueden ser clases concretas o clases abstractas. Podemos decir que cada Entidad corresponderá con una tabla de nuestra Base de Datos

Persistence: Clase con métodos estáticos que nos permiten obtener instancias de EntityManagerFactory.

EntityManagerFactory: Es una factoría de EntityManager. Se encarga crear y gestionar múltiples instancias de EntityManager

EntityManager: Es una interfaz que gestiona las operaciones de persistencia de las entidades, ya sea crear, editar, eliminar, traer de la base de datos una entidad, etc. Es la base de todo proyecto de JPA. A su vez trabaja como factoría de las Querys.

Query: Es una interfaz para obtener la relación de objetos que cumplen un criterio

EntityTransaction: Agrupa las operaciones realizadas sobre un EntityManager en una única transacción de Base de Datos.

Hay muchos términos nuevos en esta guía. Recuerda que siempre puedes volver y buscar la información. Es más importante entender el proceso que tener todo de memoria.

MAPEO CON ANOTACIONES

Como sabemos las bases de datos relacionales almacenan la información mediante tablas, filas, y columnas, de manera que para almacenar un objeto en la base de datos, hay que realizar **una correlación entre el sistema orientado a objetos de Java y el sistema relacional de nuestra base de datos**. JPA nos permite realizar dicha correlación de forma sencilla, realizando por nosotros toda la **conversión entre nuestros objetos y las tablas** de una base de datos. Esta conversión se llama **ORM** (Object Relational Mapping - Mapeo Relacional de Objetos), y puede configurarse a través de metadatos (**anotaciones**). A estos objetos, las cuales son clases comunes y corrientes, los llamaremos desde ahora **entidades**.

Las anotaciones nos permiten configurar el mapeo de una entidad dentro del mismo archivo donde se declara la clase, de este modo, relaciona las clases contra las tablas y los atributos contra las columnas. Mediante las anotaciones vamos a explicarle al ORM, como transformar la entidad en una tabla de base de datos.

Las anotaciones comienzan con el símbolo “@” seguido de un identificador. Las anotaciones son utilizadas antes de la declaración de clase, propiedad o método. A continuación, se detallan las principales:

***@Entity**:* Declara la clase como una Entidad

***@Table**:* Declara el nombre de la Tabla con la que se mapea la Entidad

***@Id**:* Declara un atributo como la clave primaria de la Tabla

***@GeneratedValue**:* Declara como el atributo que va a ser a la clave primaria va a ser inicializada. Manualmente, Automático o a partir de una secuencia.

***@Column**:* Declara que un atributo se mapea con una columna de la tabla

***@Enumerated**:* Declara que un atributo es de alguno de los valores definidos en un Enumerado (lista de valores constantes). Los valores de un tipo enumerado tienen asociado implícitamente un tipo ordinal que será asociada a la propiedad de este tipo.

@Temporal: Declara que se está tratando de un atributo que va a trabajar con fechas, entre paréntesis, debemos especificarle que estilo de fecha va a manejar en la base de datos: @Temporal(TemporalType.DATE), @Temportal(TemporalType.TIME), @Temporal(TemporalType.TIMESTAMP)

DECLARAR ENTIDADES CON @ENTITY

Como ya discutimos hace un momento, las entidades son simples clases Java como cualquier otra, sin embargo, JPA debe de ser capaz de identificar que clases son entidades para de esta forma poder administrarlas y convertirlas en tablas. Es aquí donde nace la importancia de la anotación @Entity, esta anotación se debe de definir a nivel de clase y sirve únicamente para indicarle a JPA que esa clase es un Entity, veamos el siguiente ejemplo:

```
public class Empleado {

    private Long id;

    private String nombre;
```

En el ejemplo vemos una clase común y corriente la cual representa a un Empleado, hasta este momento la clase Empleado, no se puede considerar una entidad, pues a un no tiene la **anotación @Entity que la señale como tal**. Ahora bien, si a esta misma clase le agregamos la anotación @Entity le estaremos diciendo a JPA **que esta clase es una entidad** y deberá ser **administrada por el EntityManager**, veamos el siguiente ejemplo:

```
@Entity
public class Empleado {

    private Long id;

    private String nombre;

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

En este punto la clase ya se puede considerar una Entidad.

DEFINIR LLAVE PRIMARÍA CON @ID

Al igual que en las tablas, las entidades también requieren un identificador o clave primaria(ID). Dicho identificador deberá de diferenciar a la entidad del resto. Como regla general, todas las entidades deberán definir un ID, de lo contrario provocaremos que el EntityManager marque error a la hora de instanciarlo.

El ID es importante porque será utilizado por el EntityManager a la hora de persistir un objeto, y es por este que puede determinar sobre que registro hacer el select, update o delete.

```
@Entity
public class Empleado {

@Id
private Long id;

private String nombre;

}
```

Se ha agregado @Id sobre el atributo id, de esta manera, **cuando el EntityManager inicie sabrá que el campo id es el Identificador de la clase Empleado.**

ANOTACIÓN @GENERATEDVALUE

Esta anotación se utiliza para **autogenerar el ID** (Identity) como en el caso de MySQL. JPA cuenta con la anotación @GeneratedValue para indicarle a JPA qué regla de autogeneración de la lleva primaria vamos a utilizar.

Identity

Esta estrategia para generar Id es la más fácil de utilizar pues solo hay que indicarle la estrategia y listo, no requiere nada más, JPA cuando persista la entidad **no enviará este valor**, pues asumirá que la columna **es auto generada**. Esto provoca que el contador de la columna **incremente en 1** cada vez que un nuevo objeto es insertado.

```java
@Entity
public class Empleado {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nombre;

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

MAPEO DE FECHAS CON @TEMPORAL

Mediante la anotación @Temporal es posible mapear las fechas con la base de datos de una forma simple. Una de las principales complicaciones cuando trabajamos con fecha y hora es determinar el formato empleado por el manejador de base de datos. Sin embargo, esto ya no será más problema con @Temporal.

Mediante el uso de @Temporal es posible determinar si el atributo almacena Hora, Fecha u Hora y Fecha. Para esto podemos utilizar la clase Date o Calendar. Se pueden establecer tres posibles valores para la anotación:

DATE: Acotara el campo solo a la **fecha**, descartando la hora.

```java
@Temporal(TemporalType.DATE)
```

TIME: Acotara el campo solo a la **hora**, descartando a la fecha.

```java
@Temporal(TemporalType.TIME)
```

TIMESTAMP: Toma la **fecha** y **hora**.

```java
@Temporal(TemporalType.TIMESTAMP)
```

Ejemplo:

```java
@Entity
public class Persona {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nombre;

    @Temporal(TemporalType.DATE)
    private Date fechaNacimiento;

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

LAS RELACIONES

Como sabemos en Java, los objetos pueden estar relacionados entre sí mediante las relaciones entre clases y sabemos que en MySQL las tablas tienen 4 tipos de relaciones posibles. Entonces, supongamos que tenemos dos objetos que están relacionados entre sí y queremos que esa relación también esté representada en las tablas.

Es por esto que JPA, nos da cuatro anotaciones para cuando tenemos una relación entre dos clases en Java y le queremos explicar a la base de datos, que tipo de relación tendrán las tablas entre sí. Estas anotaciones solo van a afectar a las tablas, sirven para especificar como se van a relacionar los registros de una tabla, con los registros de otra tabla. Recordemos que las anotaciones cumplen el propósito de "traducir" nuestro código de Java para que lo entienda la base de datos, por lo que las anotaciones, no van a afectar nunca a nuestro código.

Las anotaciones que nos da JPA, son los tipos de relaciones entre tablas que vimos en la guía de MySQL. Estas anotaciones son:

- **@OneToOne:** relación entre tablas uno a uno
- **@OneToMany:** relación entre tablas uno a muchos
- **@ManyToOne:** relación entre tablas muchos a uno
- **@ManyToMany:** relación entre tablas muchos a muchos

Es importante entender la manera en la que pensamos nuestras relaciones. La primera palabra de la anotación se aplica a la clase que tendrá la relación y la segunda palabra se aplica a la clase que será atributo de la primera.

@ONETOONE

Entonces, supongamos que tenemos dos clases, Curso y Profesor, entre las cuales existe una relación de 1 a 1 en Java. Un Curso por lo tanto tiene 1 Profesor y un Profesor pertenece a un Curso. Esto en nuestro código Java sería algo así:

```java
@Entity
public class Profesor {

    @Id
    private Long id;
    private String nombre;
```

```java
@Entity
public class Curso {

@Id
private Long id;
private Integer precio;
private String nombreCurso;
private Profesor profesor;
```

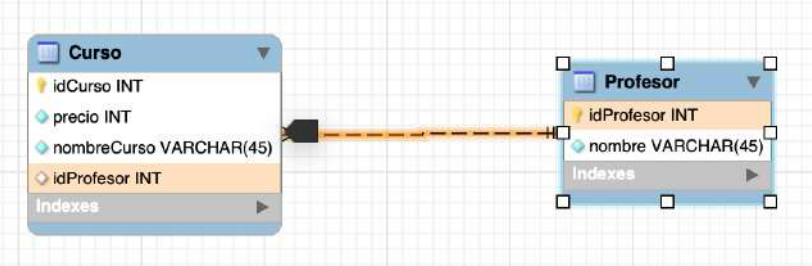
Por ahora lo único que hemos creado es una referencia a la clase Profesor **sin** utilizar JPA para nada. El siguiente paso será anotar la clase con anotaciones de JPA para que se construya la relación a nivel de persistencia. Decidimos que la relación entre tablas también sea 1 a 1, por lo que pondremos la anotación @OneToOne

```java
@Entity
public class Curso {

@Id
private Long id;
private Integer precio;
private String nombreCurso;

@OneToOne
private Profesor profesor;
```

Con esta anotación nos quedará unas tablas en MySQL de la siguiente manera:



Como podemos observar en la tabla Curso, existe una llave foránea de la tabla Profesor, de la misma manera que en nuestra clase Curso existe un objeto de tipo Profesor.

La relación OneToOne en nuestra tabla va a especificar que, para un registro de un Curso, solo hay un registro de un Profesor. En otras palabras, sería que un Curso no puede tener dos Profesores o que, a cada Curso, solo podemos asignarle/persistir un Profesor.

@MANYTOONE

Usamos esta anotación cuando queremos que entre nuestras tablas haya una relación de **muchos a uno**. Por ejemplo, muchos Álbumes pueden pertenecer a un Autor. Esta relación se representa en Java de la siguiente manera:

```java
@ManyToOne
private Autor autor;
```

La relación ManyToOne en nuestra tablas va a especificar que para uno o varios registros de Álbumes va a haber un Autor. En otras palabras sería que uno o más Álbumes van a tener el mismo Autor. Esto nos daría la posibilidad de que, a uno o muchos Álbumes asignarle el mismo Autor, esta posibilidad no existe con la OneToOne, ya que a cada registro solo le podemos asignar **un** registro.

Como el Autor solo va a ser 1, nosotros lo representamos en Java con un solo objeto, si fuera muchos autores, pondríamos una colección, esto es igual a las relaciones entre clases normales de Java.

@MANYTOMANY

Usamos esta anotación para tablas que están relacionadas con muchos elementos de un tipo determinado, pero al mismo tiempo, estos últimos registros no son exclusivos de un registro en particular, si no que pueden ser parte de varios. Por lo tanto, tenemos una Entidad A, la cual puede estar relacionada como **muchos** registros de la Entidad B, pero al mismo tiempo, la Entidad B puede pertenecer a **varias** instancias de la Entidad A. Por ejemplo: podríamos tener una familia donde cada padre tiene varios hijos y a su vez cada hijo tiene dos padres.

Algo muy importante a tomar en cuenta cuando trabajamos con relaciones @ManyToMany o @OneToMany, es que en realidad este tipo de relaciones no existen físicamente en la base de datos, y en su lugar, es necesario crear una **tabla intermedia** que relacione las dos entidades.

```
@ManyToMany
private List<Tarea> tareas;
```

Cuál clase va a tener la referencia a la otra clase va a ser decisión del programador.

TABLA INTERMEDIA

El concepto de tabla intermedia se presenta cuando tenemos una entidad que va a pertenecer a varias instancias de otra entidad. Por ejemplo, un **Empleado** o varios **Empleados** pueden hacer *una o varias* **Tareas** y a su vez, *una o varias* **Tareas** pueden ser realizadas por *uno o varios* **Empleados**. Esto en SQL sería que un registro tiene varios registros asignados a el mismo.

El problema es que en SQL solo podemos poner un dato por columna, supongamos que el **Empleado** tiene tres **Tareas** con los identificadores(id) 1,2,3. Nosotros no podemos tener una columna que tenga tres valores separados. Lo que podríamos hacer es que se repita el registro de **Empleado** tres veces con los tres identificadores, pongamos un ejemplo de una tabla que cumpla ese requisito:

Empleado			
ID	Nombre	Apellido	IdTarea
1	Adriana	Cardello	1
1	Adriana	Cardello	2
1	Adriana	Cardello	3

Esto parecería estar bien, pero si pensamos en las reglas de SQL, no podemos tener dos identificadores iguales en el mismo registro y en nuestra tabla **Empleado** hay tres veces el mismo identificador para el **Empleado**, ya que necesitamos que se repita.

Por lo que, para este tipo de relación se crea una tabla intermedia conocida como tabla asociativa. Por convención, el nombre de esta tabla debe estar formado por el nombre de las tablas participantes (en singular y en orden alfabético) separados por un guion bajo (_).

Esta tabla está compuesta por las claves primarias de las tablas que se relacionan con ella, así se logra que la relación sea de uno a muchos, en los extremos, de modo tal que la relación se lea:

Un **empleado** tiene muchas **tareas.**

y

Una **tarea** puede ser hecha por muchos **empleados**.

Las tablas se verían así:

Empleado	
ID	**Nombre**
95	Adriana
28	Mariela
31	Agustín

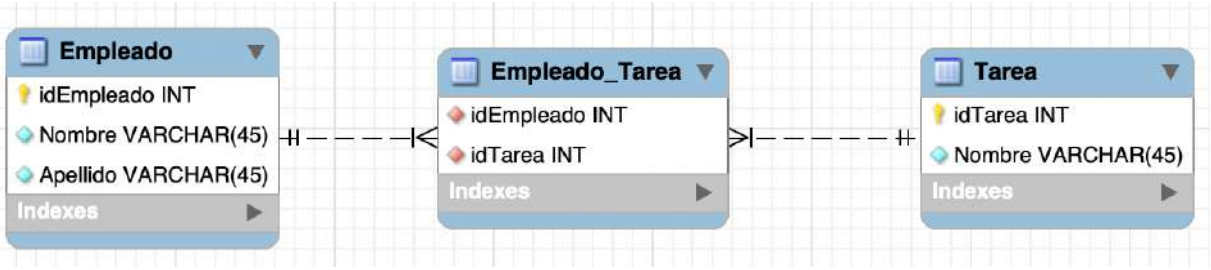
Tarea	
ID	**Nombre**
915	Ordenar
624	Limpiar
567	Cocinar

Tabla intermedia

Empleado_Tarea	
ID_Empleado	**ID_Tarea**
95	624
95	915
28	567
28	624

Como podemos observar la tabla intermedia solo tiene dos columnas, el id del empleado y el id de la tarea. **Esta tabla no toma las columnas como llaves primarias, sino como llaves foráneas**, esto nos permite repetir los valores para asignar las tareas al empleado.

Las tablas en MySQL se verían así:



Notemos que en la tabla Empleado no existe una columna que haga referencia a Tarea, ni en Tarea a Empleado, si no que es la tabla intermedia la encargada de hacer el cruce entre las dos tablas.

Recordemos que esta tabla intermedia se va a generar con las anotaciones @OneToMany y @ManyToMany.

RELACIONES JPA Y UML

En Java, nosotros tenemos dos posibles relaciones entre clases, **uno a uno o de uno a muchos**, además son las dos que podemos representar en código. En cambio, en MySQL tenemos cuatro tipo de relaciones, el problema es que hay relaciones de MySQL que no podemos representar en Java.

Por ejemplo, la relación muchos a uno, no podemos representarla en Java, ya que una clase no puede ser un List para representar el muchos. Por lo que Java al ver que hay una referencia a una clase de un solo objeto, en el caso, por ejemplo, de la ManyToOne, esto lo va a tomar como una relación de uno a uno.

Esto va a generar que cuando veamos un UML de nuestro proyecto JPA, no veamos las relaciones ManyToOne o ManyToMany, ya que, como dijimos, en Java solo existen las uno a uno o uno a muchos

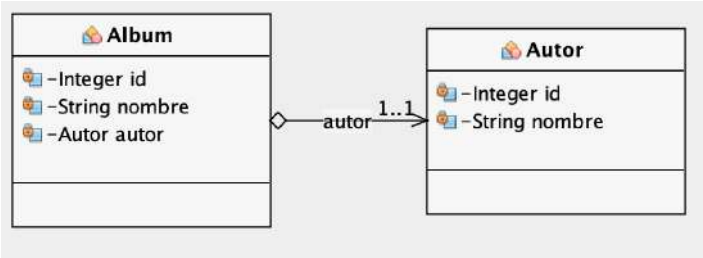
Pongamos un ejemplo, tenemos la Clase **Autor** y la clase **Álbum**, vamos a decir que para **muchos** Álbumes existe **el mismo** autor, por lo que sería una **ManyToOne**. Si la representásemos en código sería así:

```
@ManyToOne
private Autor autor;
```

Ahora si tuviéramos una OneToOne, la representación en código sería esta:

```
@OneToOne
private Autor autor;
```

Si miramos el código, podemos observar que para las dos relaciones escribimos lo mismo. Entonces, para Java la relación ManyToOne la va a representar como una OneToOne (1...1) en UML y lo mismo nos pasaría con la OneToMany y la ManyToMany.



Por lo que si nos encontramos con un UML que tiene una relación uno a uno (1...1) y nos piden decidir qué relación entre tablas usar, deberíamos considerar, que puede ser una OneToOne o una ManyToOne; o que si tiene una relación uno a muchos (1...n), puede ser una OneToMany o una ManyToMany.

Esto es porque, como habíamos visto previamente las relaciones muchos a uno y muchos a muchos son propias de MySQL, no de Java. Entonces es importante, porque a la hora de pensar en qué anotaciones le vamos a poner a nuestras entidades, pensemos en las tablas, ya que estamos trabajando sobre cómo va a ser la relación entre las tablas y no las clases.

PERSISTENCIA EN JPA CON ENTITYMANAGER

Ahora que entendemos cómo a **través de las anotaciones del ORM** podemos unificar las tablas de la base de datos con los objetos, **que pasan a llamarse entidades**. Ahora tenemos que ver cómo hacemos para poder **guardar, editar, eliminar, etc. a esas entidades en la base de datos.**

JPA tiene como interface medular al EntityManager, el cual es el **componente que se encarga de controlar el ciclo de vida de todas las entidades** definidas en la unidad de persistencia (cómo configurar la unidad de persistencia se va a encontrar al final del PDF).

El EntityManager nos dará la posibilidad de **poder crear, borrar, actualizar y consultar todas estas entidades de la base de datos**. También es la clase por medio de la cual se controlan las transacciones. Los EntityManager son configurados siempre a partir de las unidades de persistencia definidas en el archivo persistence.xml.

```
EntityManager em =
Persistence.createEntityManagerFactory("nombreUnidadDePersistencia")
.createEntityManager();
```

En esta línea se puede ver que se obtiene una instancia de la **Interfaz EntityManagerFactory, mediante la clase Persistence**, esta última recibe como parámetro el nombre de la unidad de persistencia que definimos en el archivo persistence.xml. Una vez con el EntityManagerFactory se obtiene una instancia de EntityManager para finalmente ser retornada para ser utilizada.

OPERACIONES ENTITYMANAGER

Las entidades pueden ser cargadas, creadas, actualizadas y eliminadas a través del EntityManager. Vamos a mostrar los métodos del EntityManager que nos permiten lograr estas operaciones.

Persist()

Este método nos deja **persistir una entidad en nuestra base de datos**. Persistir es la acción de **preservar la información de un objeto de forma permanente**, en este caso en una base de datos, pero a su vez también se refiere a poder recuperar la información del mismo para que pueda ser nuevamente utilizado.

Antes de ver cómo persistimos un objeto, también tenemos que entender el concepto de transacciones, ya que para persistir un objeto en la base de datos, la operación debe estar marcada como una transacción.

Una transacción es un conjunto de operaciones sobre una base de datos, que suelen **crear, editar o eliminar** un registro de la base de datos, que se deben ejecutar como una unidad. Por lo que **una consulta a la base de datos no se la considera una transacción.**

Entendiendo esto veamos un ejemplo del método Persist():

```
// Creamos un EntityManager

EntityManager em =
Persistence.createEntityManagerFactory("nombreUnidadDePersistencia").creat
eEntityManager();

//Creamos un objeto Alumno y le asignamos un nombre

Alumno alumno = new Alumno();

a1.setNombre("Nahuel");
```

```
//Iniciamos una transacción con el método getTransaction().begin();

em.getTransaction().begin();

//Persistimos el objeto

em.persist(alumno);

//Terminamos la transacción con el método commit. Commit en programación
significa confirmar un conjunto de cambios, en este caso persistir el
objeto

em.getTransaction().commit();
```

Find()

Este método se encarga de buscar y devolver una Entidad en la base de datos, a través de su clave primaria(Id). Para ello necesita que le pasemos la clave y el tipo de Entidad a buscar.

```
// Creamos un EntityManager

EntityManager em =
Persistence.createEntityManagerFactory("nombreUnidadDePersistencia").creat
eEntityManager();

// Usamos el método find para buscar una persona con el id 123 en nuestra
base de datos

Persona persona = em.find(Persona.class, 123);
```

De esta manera podremos obtener una Persona de la base de datos para usar ese objeto como queramos.

Merge()

Este método funciona igual que el método persist pero, sirve para actualizar una entidad en la base de datos.

```
EntityManager em =
Persistence.createEntityManagerFactory("nombreUnidadDePersistencia").creat
eEntityManager();

//Usamos el método find para buscar el alumno a editar

Alumno alumno = em.find(Alumno.class,1234);

//Le asignamos un nuevo nombre

alumno.setNombre("Francisco");

em.getTransaction().begin();

//Actualizamos el alumno

em.merge(alumno);

em.getTransaction().commit();
```

Remove()

Este método se encarga de eliminar una entidad de la base de datos.

```
EntityManager em =
Persistence.createEntityManagerFactory("nombreUnidadDePersistencia").creat
eEntityManager();

//Usamos el método find para buscar el alumno a borrar

Alumno alumno = em.find(Alumno.class,1234);

em.getTransaction().begin();

//Borramos el alumno

em.remove(alumno);

em.getTransaction().commit();
```

JAVA PERSISTENCE QUERY LANGUAGE (JPQL)

Es un lenguaje de consulta orientado a objetos independiente de la plataforma definido como **parte de la especificación Java Persistence API** (JPA). JPQL es usado para hacer consultas contra las entidades almacenadas en una base de datos relacional. Está inspirado en gran medida por SQL, y sus consultas se asemejan a las consultas SQL en la sintaxis, pero opera con objetos entidad de JPA en lugar de hacerlo directamente con las tablas de la base de datos.

CLAUSULAS SELECT – FROM

La cláusula FROM define de qué **entidades se seleccionan los datos**. Cualquier implementación de JPA, mapea las entidades a las tablas de base de datos correspondientes. Esto significa que **vamos a utilizar el nombre de las entidades en vez del nombre de las tablas y los atributos de las entidades en vez de las columnas de las tablas.**

La sintaxis de una cláusula FROM de JPQL es similar a SQL pero usa el modelo de entidad en lugar de los nombres de tabla o columna. El siguiente fragmento de código muestra una consulta JPQL simple en la que selecciono todas las entidades Autor.

```
SELECT a FROM Autor a;
```

En la query se ve que, se hace referencia a la entidad Autor en lugar de la tabla de autor y **se le asigna la variable de identificación a.** La variable de identificación a menudo se llama **alias** y es similar a una variable en su código Java.

Se utiliza en todas las demás partes de la consulta para hacer referencia a esta entidad. Por ejemplo, si queremos seleccionar un atributo de la entidad Autor, en vez de todos, usaríamos el alias asi:

```
SELECT a.nombre, a.apellido FROM Autor a;
```

CLAUSULA WHERE

La sintaxis es muy similar a SQL, pero JPQL admite solo un pequeño subconjunto de las características de SQL.

JPQL admite un conjunto de operadores básicos para definir expresiones de comparación. La mayoría de ellos son idénticos a los operadores de comparación admitidos por SQL, y puede combinarlos con los operadores lógicos AND, OR y NOT en expresiones más complejas.

Operadores:

- **Igual:** `author.id = 10`
- **Distinto:** `author.id <> 10`
- **Mayor que:** `author.id > 10`
- **Mayor o Igual que:** `author.id => 10`
- **Menor que:** `author.id < 10`
- **Menor o igual que:** `author.id <= 10`
- **Between:** `author.id BETWEEN 5 and 10`
- **Like:** `author.firstName LIKE :'%and%'`
- **Is null:** `author.firstName IS NULL`

  Se lo puede negar con el operador NOT, para traer todos los que no son nulos
- **In:** `author.firstName IN ('John','Jane')`

  Va a traer todos los autores con el nombre John o Jane.

UNIR ENTIDADES

Si necesitamos seleccionar datos de más de una entidad, por ejemplo, todos los libros que ha escrito un autor, debe unir las entidades en la cláusula FROM. La forma más sencilla de hacerlo es utilizar las asociaciones definidas de una entidad como en el siguiente fragmento de código.

```
SELECT a FROM Libro a JOIN a.autor b;
```

La definición de la entidad Libro proporciona toda la información necesaria para unirla a la entidad Autor, y no es necesario que proporcione una declaración ON adicional.

También podemos utilizar el operador ".", para navegar a través del atributo de autor de la entidad Libro y traer los libros que tengan un autor con un nombre a elección. Esto generaría una relación implícita entre las dos entidades, sin la necesidad de usar un Join.

```
SELECT a FROM Libro a WHERE a.autor.nombre LIKE : "Homero";
```

CREAR CONSULTAS / QUERYS CON ENTITYMANAGER

El EntityManager nos permite crear querys dinámicas con el lenguaje JPQL. Esto se logra mediante el método createQuery(query). El método createQuery, recibe una query SQL, la envía a la base de datos, que está anclada con la Unidad de Persistencia y devuelve el resultado de la consulta.

Para obtener estos resultados vamos a usar dos métodos de la clase Query. Un método es getResultList(), que nos deja atrapar el resultado de la query y guardarlo en una lista y el otro método es getSingleResult(), que sirve para cuando queremos traer un solo resultado de la consulta.

Igualmente, recomendamos **usar siempre getResultList(),** ya que, si la consulta llega a traer más de un resultado nos va a generar una excepción, entonces, para evitar esto usamos el getResultList por las dudas.

```
// Para esto vamos a tener que crear un EntityManager

EntityManager em =
Persistence.createEntityManagerFactory("nombreUnidadDePersistencia").creat
eEntityManager();

// Usamos el metodo createQuery y le ponemos la query de JPQL

List<Autor> autores = em.createQuery("SELECT a FROM Autor a")
.getResultList();

// Ponemos una lista del tipo de dato que vamos a traer en la query y
// usamos el getResultList() para atrapar todos los resultados de la
query.
```

De esta manera ya tenemos todos los autores de la base de datos guardados en una lista, solo nos quedaría imprimir la lista de autores y mostrar todos los autores.

AGREGAR PARÁMETROS QUERY

Supongamos que tenemos la siguiente query:

```
“SELECT p FROM Persona p WHERE p.edad LIKE :20”
```

De esta manera la query no es dinámica, ya que siempre va a buscar personas con la edad de 20, si queremos hacer que los parámetros de las querys sean dinámicas, vamos a hacerlo mediante el método de la clase Query, setParameter().

El método recibe un String y una variable para ingresar a la Query. El String tiene que tener el mismo nombre que el parámetro que vamos a poner en la query y el método setParameter(), va a asignarle esa variable al parámetro de la query. Pongamos un ejemplo

```
EntityManager em = Persistence.createEntityManagerFactory("nombreUnidadDePersistencia")
                              .createEntityManager();


public List listarPorEdad(int age) {

    List<Persona> personas = em.createQuery("SELECT p FROM Persona p WHERE p.edad LIKE :edad")
                           .setParameter("edad", age).getResultList();

    return personas;

}
```

La query, utiliza un parámetro llamado edad, y en el setParameter(), le decimos que el parámetro **“edad”** de la query va a ser igual al valor que está en la variable entera **age**. Esto nos permite “dinamizar” la query. Solo deberíamos pedirle al usuario un valor distinto para edad, cada vez que se haga la query.

EJERCICIOS DE APRENDIZAJE

Para la realización de los ejercicios que se describen a continuación, sigue siendo necesario el conector de MySQL y es necesario leer el **Instructivo Unidad de Persistencia** este se encuentra al **final de la guía** o lo tenemos para descargar en el aula virtual.

VIDEOS: Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.

1. Sistema de Guardado de una Librería

El objetivo de este ejercicio es el desarrollo de un sistema de guardado de libros en JAVA utilizando una base de datos MySQL y JPA como framework de persistencia.

Creación de la Base de Datos MySQL:

Lo primero que se debe hacer es crear la base de datos sobre el que operará el sistema de reservas de libros. Para ello, se debe abrir el IDE de base de datos que se está utilizando (Workbench) y ejecutar la siguiente sentencia:

```
CREATE DATABASE libreria;
```

De esta manera se habrá creado una base de datos vacía llamada librería.

Paquetes del Proyecto Java:

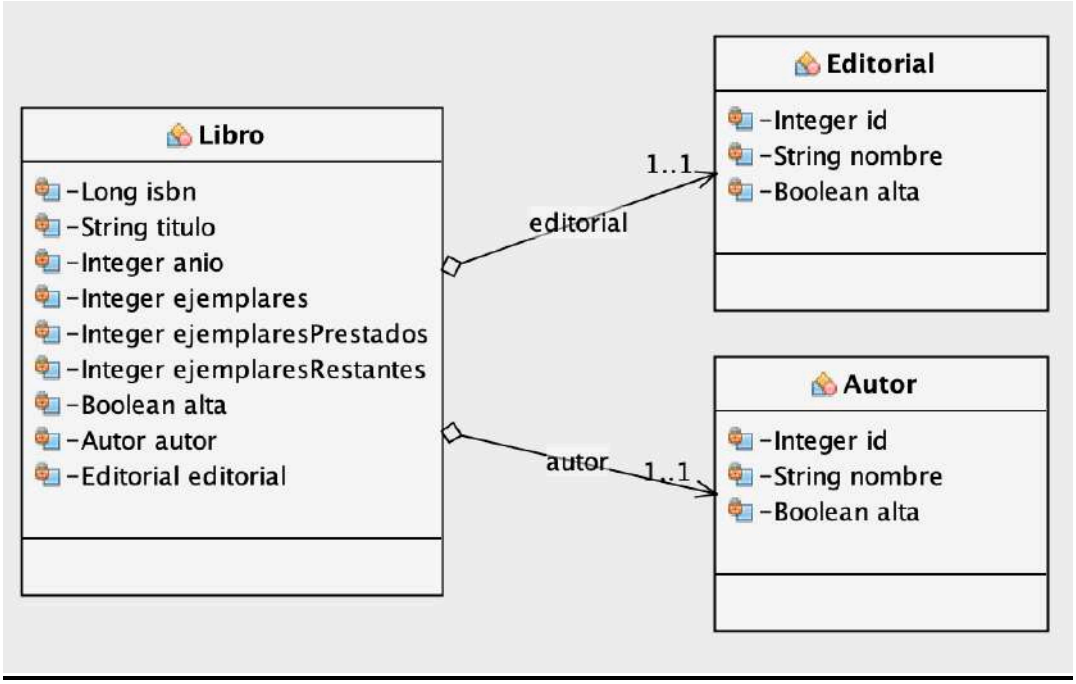
Los paquetes que se utilizarán para este proyecto son los siguientes:

- **entidades**: en este paquete se almacenarán aquellas clases que se quiere persistir en la base de datos.
- **servicios:** en este paquete se almacenarán aquellas clases que llevarán adelante la lógica del negocio. En general se crea un servicio para administrar las operaciones **CRUD** (Create, Remove, Update, Delete) cada una de las entidades y las consultas de cada entidad.

Nota: En este proyecto vamos a eliminar entidades, pero no es considerado una buena práctica. Por esto, además de eliminar nuestras entidades, vamos a practicar que nuestras entidades estén dadas de alta o de baja. Por lo que las entidades tendrán un atributo “activo” de tipo booleano, que estará en true al momento de crearlas y en false cuando las demos de baja, para evitar eliminarlas de la base de datos.

a) Entidades

Crearemos el siguiente modelo de entidades:



Entidad Libro

La entidad libro modela los libros que están disponibles en la biblioteca para ser prestados. En esta entidad, el atributo “ejemplares” contiene la cantidad total de ejemplares de ese libro, mientras que el atributo “ejemplaresPrestados” contiene cuántos de esos ejemplares se encuentran prestados en este momento y el atributo “ejemplaresRestantes” contiene cuántos de esos ejemplares quedan para prestar.

Entidad Autor

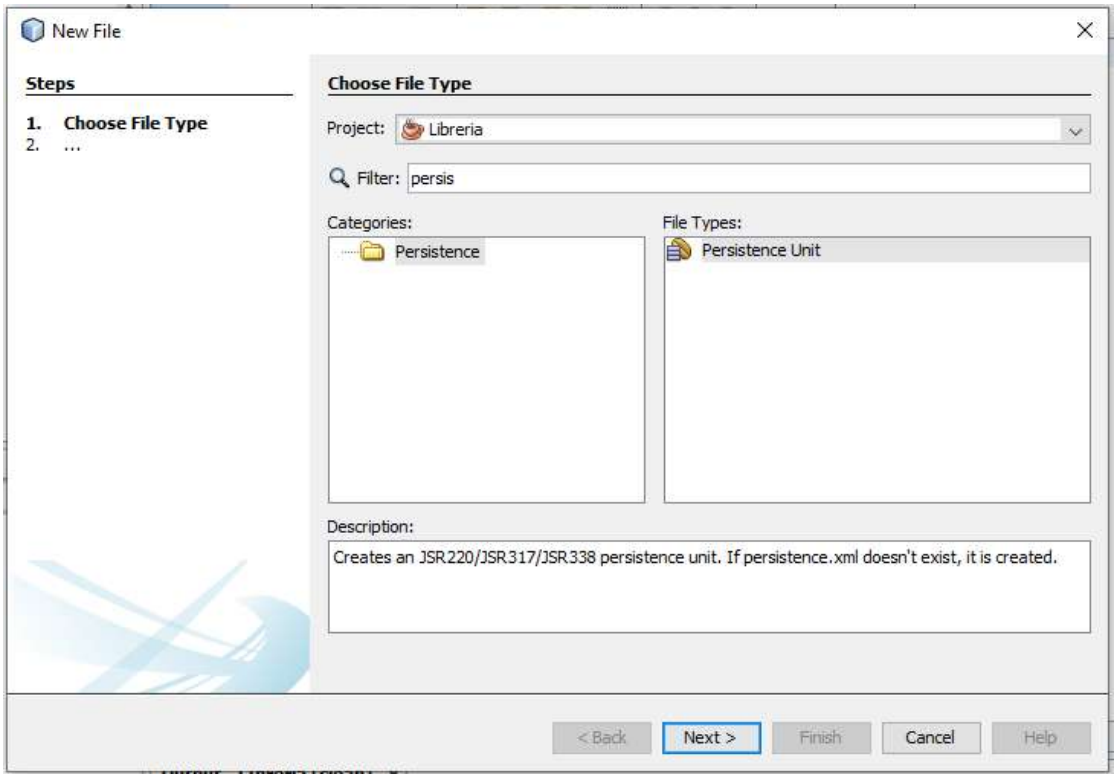
La entidad autor modela los autores de libros.

Entidad Editorial

La entidad editorial modela las editoriales que publican libros.

b) Unidad de Persistencia

Para configurar la unidad de persistencia del proyecto, se recomienda leer el **Instructivo Unidad de Persistencia** recuerde hacer click con el botón derecho sobre el proyecto y seleccionar nuevo. A continuación, se debe seleccionar la opción de Persistence Unit como se indica en la siguiente imagen.



Base de Datos

Para este proyecto nos vamos a conectar a la base de datos Librería, que creamos previamente.

Generación de Tablas

La estrategia de generación de tablas define lo que hará JPA en cada ejecución, si debe crear las tablas faltantes, si debe eliminar todas las tablas y volver a crearlas o no hacer nada. Recomendamos en este proyecto utilizar la opción: “Create”

Librería de Persistencia

Se debe seleccionar para este proyecto la librería “EclipseLink”.

c) Servicios

AutorServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar autores (consulta, creación, modificación y eliminación).

EditorialServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar editoriales (consulta, creación, modificación y eliminación)

LibroServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar libros (consulta, creación, modificación y eliminación).

d) Main

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para interactuar con el usuario. En esta clase se muestra el menú de opciones con las operaciones disponibles que podrá realizar el usuario.

e) Tareas a realizar

Al alumno le toca desarrollar, las siguientes funcionalidades:

1) Crear base de datos Librería

2) Crear unidad de persistencia

3) Crear entidades previamente mencionadas (excepto Préstamo)

4) Generar las tablas con JPA

5) Crear servicios previamente mencionados.

6) Crear los métodos para persistir entidades en la base de datos librería

7) Crear los métodos para dar de alta/bajo o editar dichas entidades.

8) Búsqueda de un Autor por nombre.

9) Búsqueda de un libro por ISBN.

10) Búsqueda de un libro por Título.

11) Búsqueda de un libro/s por nombre de Autor.

12) Búsqueda de un libro/s por nombre de Editorial.

13) Agregar las siguientes validaciones a todas las funcionalidades de la aplicación:

- Validar campos obligatorios.
- No ingresar datos duplicados.

EJERCICIOS DE APRENDIZAJE EXTRAS

Estos van a ser ejercicios para reforzar los conocimientos previamente vistos. Estos pueden realizarse cuando hayas terminado la guía y tengas una buena base sobre lo que venimos trabajando. Además, si ya terminaste la guía y te queda tiempo libre en las mesas, podes continuar con estos ejercicios extra, recordando siempre que no es necesario que los termines para continuar con el tema siguiente. Por ultimo, recordá que la prioridad es ayudar a los compañeros de la mesa y que cuando tengas que ayudar, lo más valioso es que puedas explicar el ejercicio con la intención de que tu compañero lo comprenda, y no sólo mostrarlo. ¡Muchas gracias!

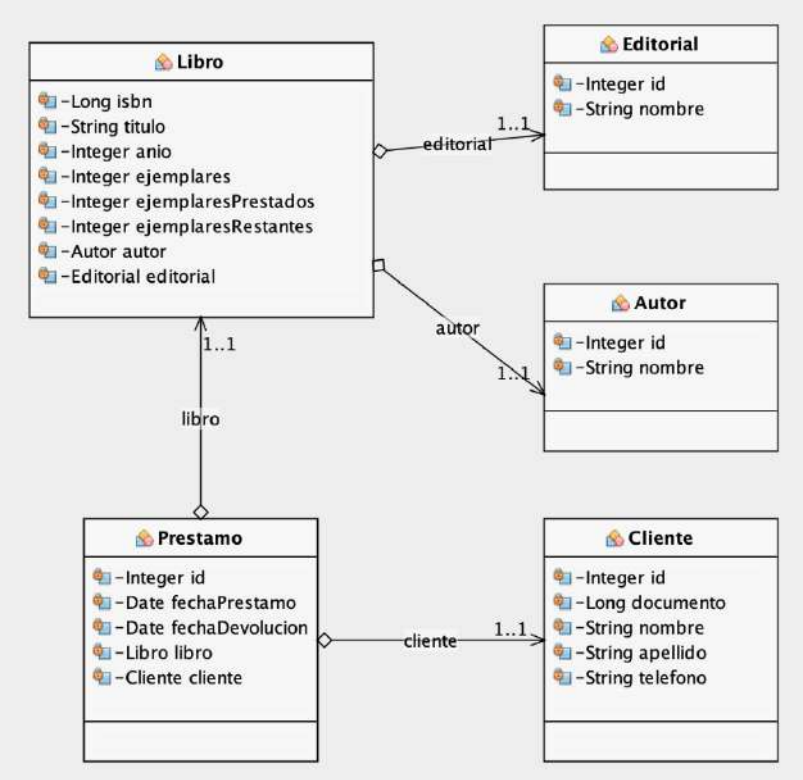
1. Sistema de Reservas de una Librería

Vamos a continuar con el ejercicio anterior. Ahora el objetivo de este ejercicio es el desarrollo de un sistema de reserva de libros en JAVA. Para esto vamos a tener que sumar nuevas entidades a nuestro proyecto en el paquete de entidades y crearemos los servicios de esas entidades.

Usaremos la misma base de datos y se van a crear las tablas que nos faltan. Deberemos agregar las entidades a la unidad de persistencia.

a) Entidades

Crearemos el siguiente modelo de entidades:



Entidad Cliente

La entidad cliente modela los clientes (a quienes se les presta libros) de la biblioteca. Se almacenan los datos personales y de contacto de ese cliente.

Entidad Préstamo

La entidad préstamo modela los datos de un préstamo de un libro. Esta entidad registra la fecha en la que se efectuó el préstamo y la fecha en la que se devolvió el libro. Esta entidad también registra el libro que se llevo en dicho préstamo y quien fue el cliente al cual se le prestaron.

b) Servicios

ClienteServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para administrar clientes (consulta, creación, modificación y eliminación).

PrestamoServicio

Esta clase tiene la responsabilidad de llevar adelante las funcionalidades necesarias para generar prestamos, va a guardar la información del cliente y del libro para persistirla en la base de datos. (consulta, creación, modificación y eliminación).

c) Tareas a realizar

1) Al alumno le toca desarrollar, las siguientes funcionalidades:

2) Creación de un Cliente nuevo

3) Crear entidad Préstamo

4) Registrar el préstamo de un libro.

5) Devolución de un libro

6) Búsqueda de todos los préstamos de un Cliente.

- Agregar validaciones a todas las funcionalidades de la aplicación:
- Validar campos obligatorios.
- No ingresar datos duplicados.
- No generar condiciones inválidas. Por ejemplo, no se debe permitir prestar más ejemplares de los que hay, ni devolver más de los que se encuentran prestados. No se podrán prestar libros con fecha anterior a la fecha actual, etc.

INSTRUCTIVO CONEXIÓN NETBEANS - MYSQL

DESCARGAR EL CONECTOR

1) Primero debemos descargar el conector a utilizar (Por única vez). Este se encuentra en el drive en la carpeta de persistencia.

   - Descargar el archivo **mysql-connector-java-5.1.48. Nota:** si estamos usando alguna versión de Netbeans que no sea la 8.2, deberán descargar de internet cualquier conector de versión 8.

   - Ubicarlo en la carpeta:

     - C:\Program Files\NetBeans 8.2\ide\modules\ext

2) Segundo debemos verificar los permisos en MySQL (Por única vez).

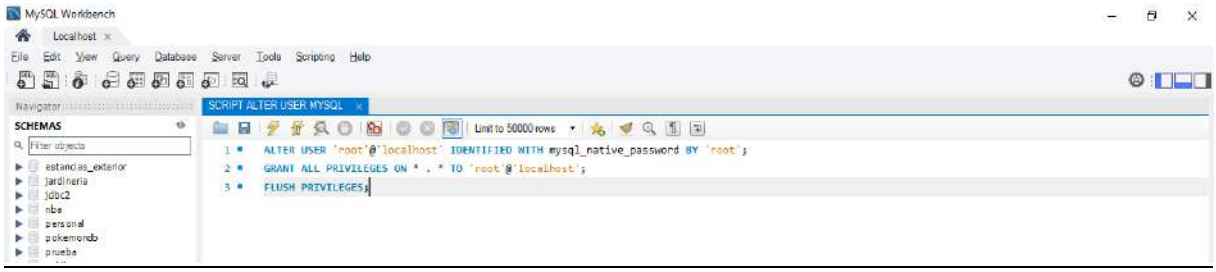
   - Descargar Script ALTER USER del drive.

   - Ingresar MySQL Worbench.

   - Ingresar con nuestra contraseña a la conexión LocalHost.

   - Ejecutar el Script provisto.

   - Se deberá ver de la siguiente manera:



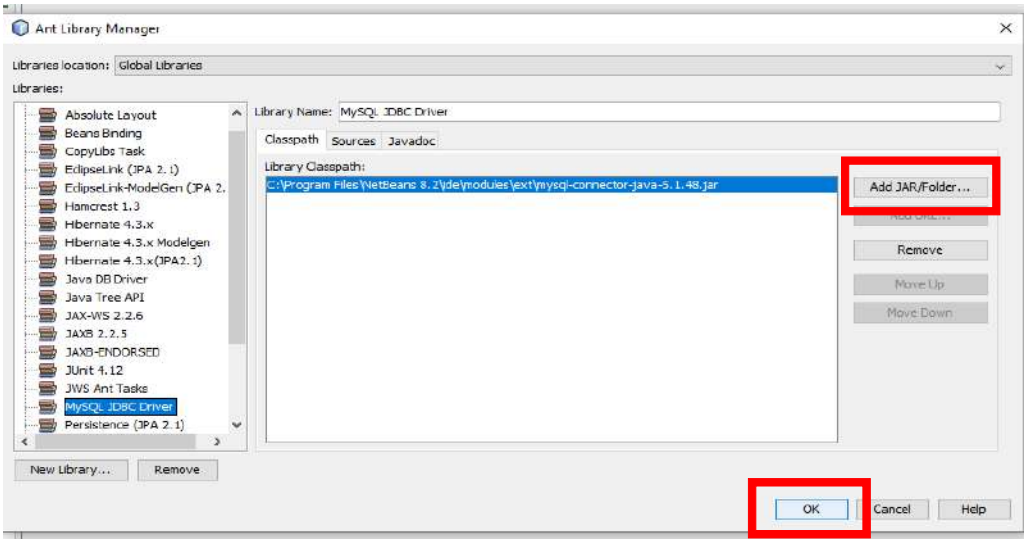
3) Tercero cargar la librería en NetBeans (Por única vez)

   - Dejar prevista la librería en mi IDE de trabajo.

   - Tools → Libraries → MySQL JDBC Driver → Add JAR / Folder → Busco el archivo cargado con antelación → OK (Es decir, vinculo el conector).

     Nota: Cuando carguemos nuestro conector es importante eliminar, el archivo **mysql-connector-java-5.1.23** (Es el que viene por defecto cuando instalamos NetBeans). Para eso sale la opción de *remove* en Netbeans.

     El conector cargado nos debería quedar así:



OPCION A PARA CREAR UNA CONEXIÓN

Crear un proyecto en NetBeans (Para cada proyecto que hará uso de JDBC)

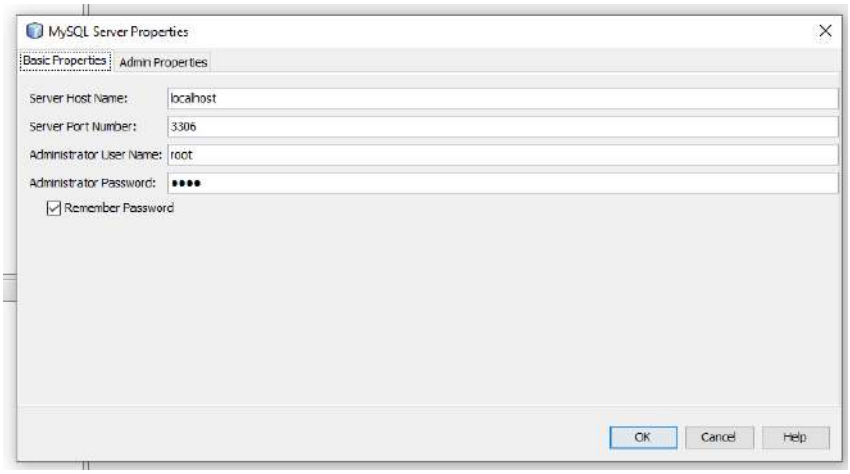
- En Worbench MySQL dejar creada mi base de datos a utilizar con el proyecto puntual.
- Creo mi proyecto en NetBeans.
- En la carpeta de Libraries de mi proyecto → Botón Derecho → Add Library → Selecciono MySQL JDBC Driver

  Nota: Esto le avisara a NetBeans que el proyecto que cree va a necesitar de esta librería específica para ejecutarse..

- Voy a la pestaña Services de mi panel de exploración → Databases → Botón Derecho → Register MySQL Server → Completar la configuración de PROPERTIES

  Nota: Si no aparece la pestaña Services en mi explorador. Barra → Windows → Reset Windows (se reiniciara la vista de NetBeans).

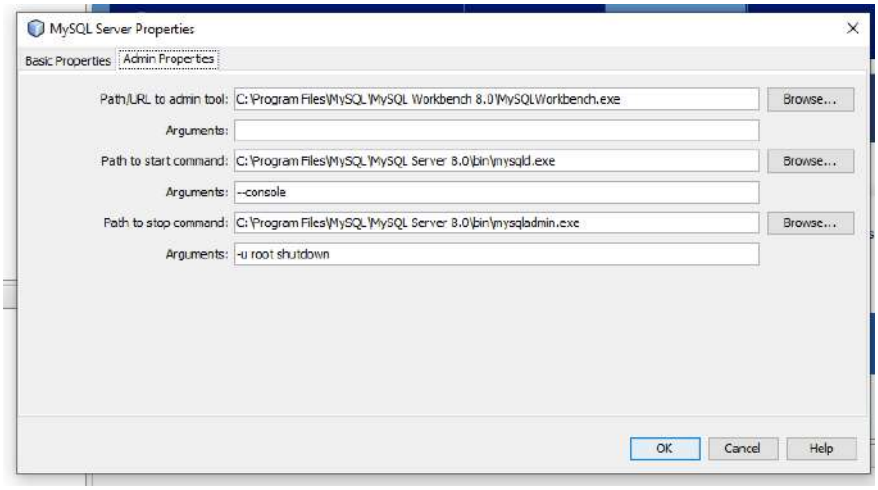
Configuración Pestaña Basic Properties:



Importante verificar que los 4 campos estén completos.

Configuración Pestaña Admin Properties:

- **Path/URL to admin tool:** C:\Program Files\MySQL\MySQL Workbench 8.0\MySQLWorkbench.exe
- **Arguments:** DEJAR EN BLANCO
- **Path to start command**: C:\Program Files\MySQL\MySQL Server 8.0\bin\mysqld.exe
- **Arguments:** --console
- **Path to start command:** C:\Program Files\MySQL\MySQL Server 8.0\bin\mysqladmin.exe
- **Arguments:** -u root shutdown



- Si todo funciono bien, podrá ver una conexión creada al Localhost
- Posicionarme sobre la misma → Connect (Esto permite dejar NetBeans y MySQL conectados)
- Una vez que me conecte, debo realizar Connect sobre la base con la que voy a trabajar (Esto permite dejar conectados Netbeans con la base de datos específica a trabajar).

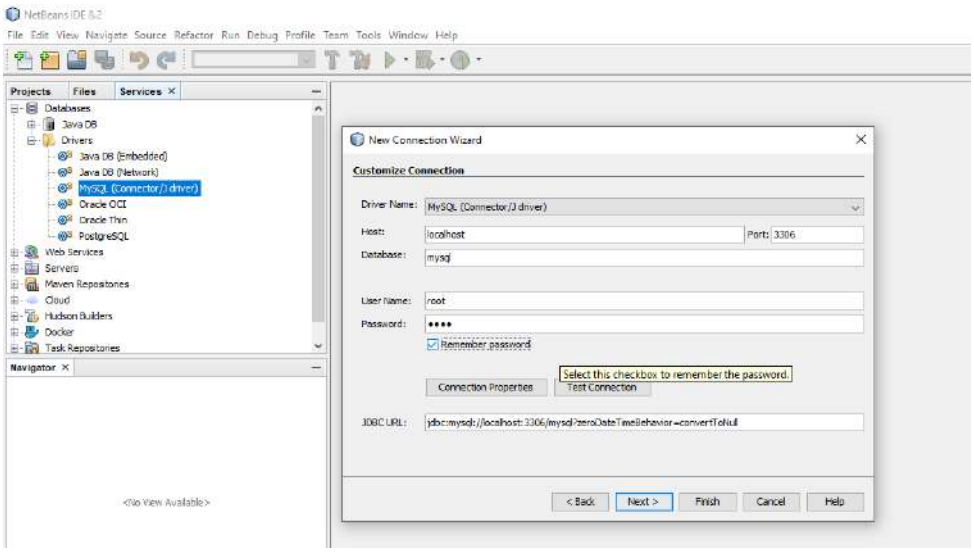
OPCION B PARA CREAR UNA CONEXIÓN (SOLO UTILIZAR SI NO FUNCIONA LA OPCION A O USAS MAC)

- Crear una conexión individual a la base con la que voy a trabajar.
- Voy a la pestaña Services de mi panel de exploración → Drivers → Mysql (Connector – J Driver) → Connect Using →

  Nos saldrá otra pantalla donde deberemos rellenar lo siguiente:

  - **Database**: escribir el nombre de la base de datos con la que voy a trabajar. Debe estar creada previamente
  - **Username:** revisar que nuestro UserName esté correcto
  - **Password:** ingresar el pass (Dejar marcado el box de remember pass)

  Nos quedará asi:



- Verificar si la conexión quedo OK → Test Connection
- Finish
- Verificaremos que la conexión esta activa con la base vinculada.

SUMAR DRIVER A PROYECTO NETBEANS

Una vez hecho todo esto a cada proyecto que queramos conectar con NetBeans deberemos sumarle el driver a las propiedades del proyecto.

Para esto vamos a darle click derecho al proyecto → Properties → Libraries → Add Library → MySQL JDBC Driver → Add Library.

Nos va a quedar así: