IMPLEMENTAR BASE DE DATOS

1- Object Relational Mapping

Se trata de mapear o convertir datos del tipo Objeto a datos del tipo relacionales y viceversa.

La capa ORM tiene como función principal asociar un objeto a datos en la BD, posibilitando escribir las clases de persistencia utilizando OO e interactuar con las tablas y columnas de una base de datos relacional.

Hay muchos ORM para Java: EJB, JDO, JPA.

Mientras que estos son especificaciones, Hibernate, es una implementación.

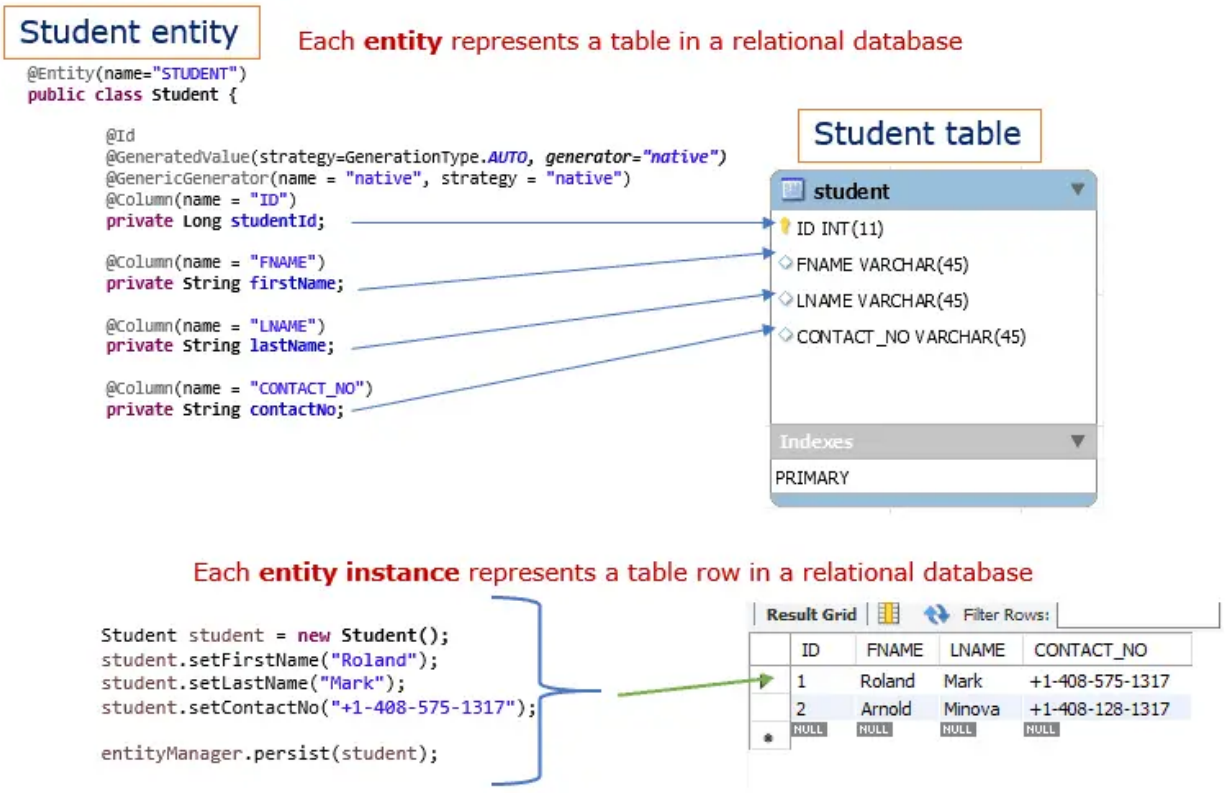
2- Java Persistence API

Java Persistence API es una colección de clases y métodos que almacenan de forma persistente grandes cantidades de datos en una BD. Es un conjunto de conceptos que pueden ser implementados por cualquier herramienta o framework.

JPA busca solucionar la problemática planteada al intentar traducir un modelo orientado a objetos a un modelo relacional.

Permite almacenar entidades del negocio como entidades relacionales. Al usar JPA se crea un mapa
de la BD a los objetos del modelo de datos de la aplicación. La conexión entre la BD relacional y la aplicación es gestionada por JDBC (Java Database Connectivity API).

Entities, Queries, Criteria:



Query:

Es una interfaz utilizada para controlar la ejecución de las consultas. Un EntityManager ayuda a crear un objeto Query, su implementación dependerá del proveedor de persistencia

Entidad:

Es un objeto de persistencia. Cada Entidad representa una tabla en una base de datos relacional y cada instancia de una entidad corresponde a un registro en esa tabla. JPA utiliza anotaciones o xml para mapear entidades a una BD relacional.

Criteria API:

Permite construir queries SQL usando objetos java. Es posible realizar consultas tipadas seguras, que pueden ser chequeadas en tiempo de compilación.

3- Anotaciones

@Entity: Se etiqueta a la clase como un Bean del tipo entity que va a ser mapeado por el ORM con una tabla de la BD.

@Table: Especifica detalles de la tabla que va a ser usada para persistir la entidad en la BD. Con el atributo **«name»** , podemos explicitar el nombre de la tabla a la que debe asociarse la clase. No es necesario usarlo si la tabla se llama exactamente igual que nuestra clase.

@Id: Cada Bean del tipo entity va a tener una Primary Key (que puede ser simple o compuesta). Especifica cuál es el índice, permite que la BD genere un nuevo valor con cada operación de inserción.

@GeneratedValue: Si no utilizamos esta anotación la aplicación es responsable de gestionar por sí misma el campo **@Id**. El atributo **strategy = GenerationType** puede tener los siguientes valores.

- **AUTO:** Por defecto. El tipo de id generada puede ser numérica o UUID.
- **IDENTITY:** Asigna claves primarias para las entidades que utilizan una columna de identidad, son auto-incrementales.
- **SEQUENCE:** Asigna claves primarias para las entidades utilizando una secuencia que puede ser customizada.
- **TABLE** : Asigna claves primarias para las entidades utilizando una tabla de la BD, guardando en una tabla el último valor de clave primaria.

@Column: Especifica los detalles de una columna, para indicar a qué atributo o campo será mapeada. Puede usarse con los siguientes atributos:

- **name =** explicita el nombre.
- **length =** especifica el tamaño de columna utilizado para mapear un valor especialmente en el caso de un String. Indica la característica del largo de la columna en la BD. Por ejemplo con length **=** 3, genera una columna del tipo VARCHAR(3) . Intentar insertar una String más larga daría un error de SQL.
- **nullable =** puede marcarse la columna con nullable = false cuando se genera el schema.
- **unique**= permite sólo valores únicos en esa columna.

4- Hibernate

Hibernate es un servicio ORM de persistencia y consultas para Java. Se trata de una implementación de Java Persistence API (JPA), pero no la única. Mapea las clases Java en tablas de BD, y provee mecanismos para consultar datos. El mapeo lo hace a través de una configuración .xml o de anotaciones. Si es necesario un cambio en la BD, solo deberá cambiarse el archivo de configuración o las anotaciones.

Relaciones:

Una relación es una conexión entre dos tipos de entidades.

- OneToOne: En el caso de que una clase tenga un atributo del tipo de otra clase, y sólo uno, podemos realizar el mapeo utilizando esta anotación.
- OneToMany: Es necesaria cuando una fila en la tabla debe mapearse a múltiples filas en otra tabla.Generalmente son unidireccionales del lado del OneToMany o son bidireccionales.
- ManyToOne: Es necesaria cuando muchas instancias de una entidad son mapeadas a una instancia de otra entidad.
- ManyToMany: Se trata del caso en que muchas entidades se relacionan con muchas entidades de otro tipo (aunque en ocasiones puede ser del mismo).

Anotaciones:

mappedBy : se usa para indicar que la variable “address” del lado de la clase User es quien establece la relación.

cascade : Atributo de la anotación. Lo usamos para indicar el tipo de propagación que usaremos en la base de datos. La mayoría de las veces las relaciones entre entidades dependen de la existencia de otra entidad. Sin una, la otra no podría existir, y si modificamos una deberíamos modificar ambas. Por lo tanto, si realizamos una acción sobre una entidad, la misma acción debe aplicarse a la entidad asociada.

Objeto Literal	JSON
CascadeType.ALL	Realiza la propagación de todas las operaciones.
CascadeType.PERSIST	Propaga las operaciones de persistencia desde una entidad padre a sus hijas.
CascadeType.MERGE	Propaga las operaciones de combinación de una entidad padre a una hija.
CascadeType.REMOVE	Propaga las operaciones de borrado de una entidad padre a una hija.

@JoinColumn(name="id_persona") : esto se usa para modificar la relacion, pero no hace falta ponerlo.

5- HQL

Significa **Hibernate Query Language** y es un lenguaje totalmente orientado a objetos y comprende nociones como herencia, polimorfismo y asociación. Cuando escribimos una consulta en **HQL**, lo que ocurre por detrás, es que Hibernate las convierte (traduce) a SQL, según la base de datos que estemos utilizando.

Una consulta HQL puede constituirse de los siguientes elementos:

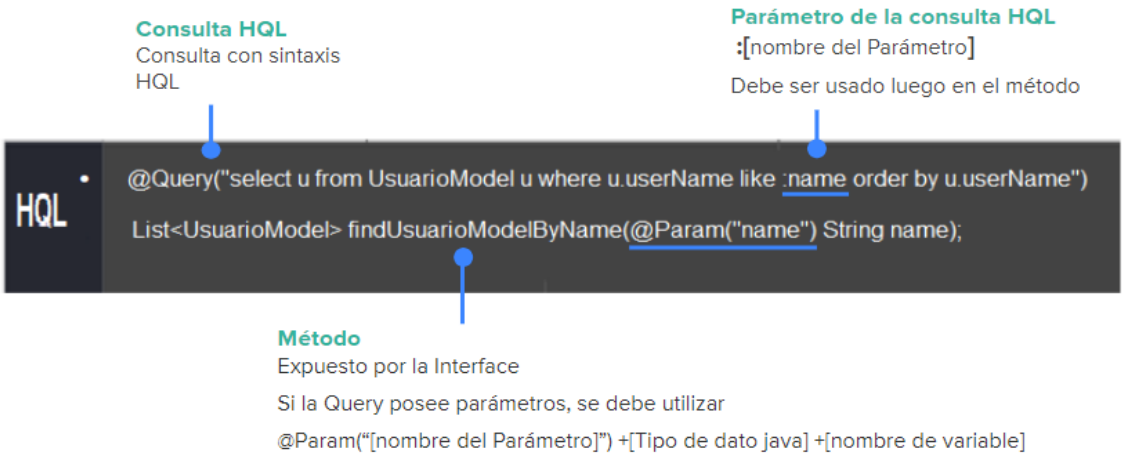
Elementos									
Cláusulas	from	as	select	where	order by	group by	update	delete	Insert
Funciones Agregadas	avg	sum	min	max	count(*) count(...) count(distinct ...) count(all...)				
Sub Consultas	Consultas dentro de consultas								

Clausulas:

- **FROM: carga un objeto persistente en memoria**
- **AS: asignar alias a la clase**

- **SELECT: obtiene propiedades de objetos, en lugar del objeto completo**
- **WHERE: recupera objetos específicos**
- **ORDER BY: ordena resultados por propiedad de los objetos, ascendente ASC o descendente DESC**
- **GROUP BY: devuelve valores agregados que se agrupan por cualquier propiedad**
- **UPDATE: actualiza una o más propiedades**
- **DELETE: elimina uno o más objetos**
- **INSERT: inserta objetos**

Sintaxis de Consultas en Spring



Consultas: Nombres de métodos

Keyword	*Sample*
And	**findBy**Lastname**And**Firstname
Or	**findBy**Lastname**Or**Firstname
Is,Equals	**findBy**Firstname, **findByFirst**name**Is**, **findByFirst**name**Equals**
Between	**findBy**StartDate**Between**
LessThan	**findBy**Age**LessThan**
LessThanEqual	**findBy**Age**LessThanEqual**
GreaterThan	**findBy**Age**GreaterThan**
GreaterThanEqual	**findBy**Age**GreaterThanEqual**
After	**findBy**StartDate**After**
Before	**findBy**StartDate**Before**
IsNull	**findBy**Age**IsNull**
IsNotNull,NotNull	**findBy**Age**NotNull**, **findBy**Age**IsNotNull**
Like	**findBy**Firstname**Like**
NotLike	**findBy**Firstname**NotLike**
StartingWith	**findBy**Firstname**StartingWith**
EndingWith	**findBy**Firstname**EndingWith**
Containing	**findBy**Firstname**Containing**
OrderBy	**findBy**Age**OrderBy**Lastname**Desc**

6- PASOS INTELLIJ:

1- En el **pom.xml** de nuestro proyecto SpringBoot debemos agregar las siguientes **dependencias**:

- **spring-boot-starter-data-jpa: Esta dependencia incluye la API JPA, la implementación de JPA, JDBC y otras librerías. Como la implementación por defecto de JPA es Hibernate, esta dependencia también lo trae incluído.**
- **com.h2database: Para hacer una prueba rápida, podemos agregar H2, que se trata de una base de datos en memoria muy liviana. En application.properties habilitamos la consola de la BD H2, para poder acceder a ella través de una UI.**

2- Mismas anotaciones que deben tener las entidades:

// LOMBOK

@Data

@AllArgsConstructor

@NoArgsConstructor

```
// JPA
@Entity
@Table(name = "duenio")
```

3- Convertir los atributos de las entidades en columnas:

```
public class Person {

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

@Column(name = "nombre", nullable = false, length = 40)
 private String firstname;

@Column(name = "apellido", nullable = false, length = 20)
private String lastname;

@Column(name = "documento", nullable = false, length = 8)
private String dni;

@Column(name = "fechaNacimiento", nullable = false)

@JsonFormat(pattern = "dd/MM/yyyy")
private LocalDate birthDate;

@Column(name = "edad", nullable = false, scale = 2)
```

```
private Short age;

@Column(name = "salario", precision = 2)
 private Double salary;
}
```

4- En workbench, vamos a crear una nueva Base de datos llamada **“prueba_jpa”**

5- Vamos a nuestro archivo properties, y vamos a configurar los siguientes parámetros:

```
# ------------------------------|  MySQL Configurations  |------------------------------- ###
##Conectar con la base de datos a traves de dependecia.
spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver

# El dialecto es la version del lenguaje SQL
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

#create-drop, update, create, none
spring.jpa.hibernate.ddl-auto=create-drop

# CONNECTION
# Database's User & Password
spring.datasource.username=root
spring.datasource.password=1234

# URL & Name
spring.datasource.url= jdbc:mysql://localhost:3306/prueba_jpa  ejemplo

spring.jpa.show-sql=true
## Estas tres lineas son para levantar un archivo SQL
```

```
spring.sql.init.mode=always
spring.sql.init.data-locations=classpath:data.sql
spring.jpa.defer-datasource-initialization=true
```

6- Una vez que tenemos todo configurado, podemos probar si funciona.

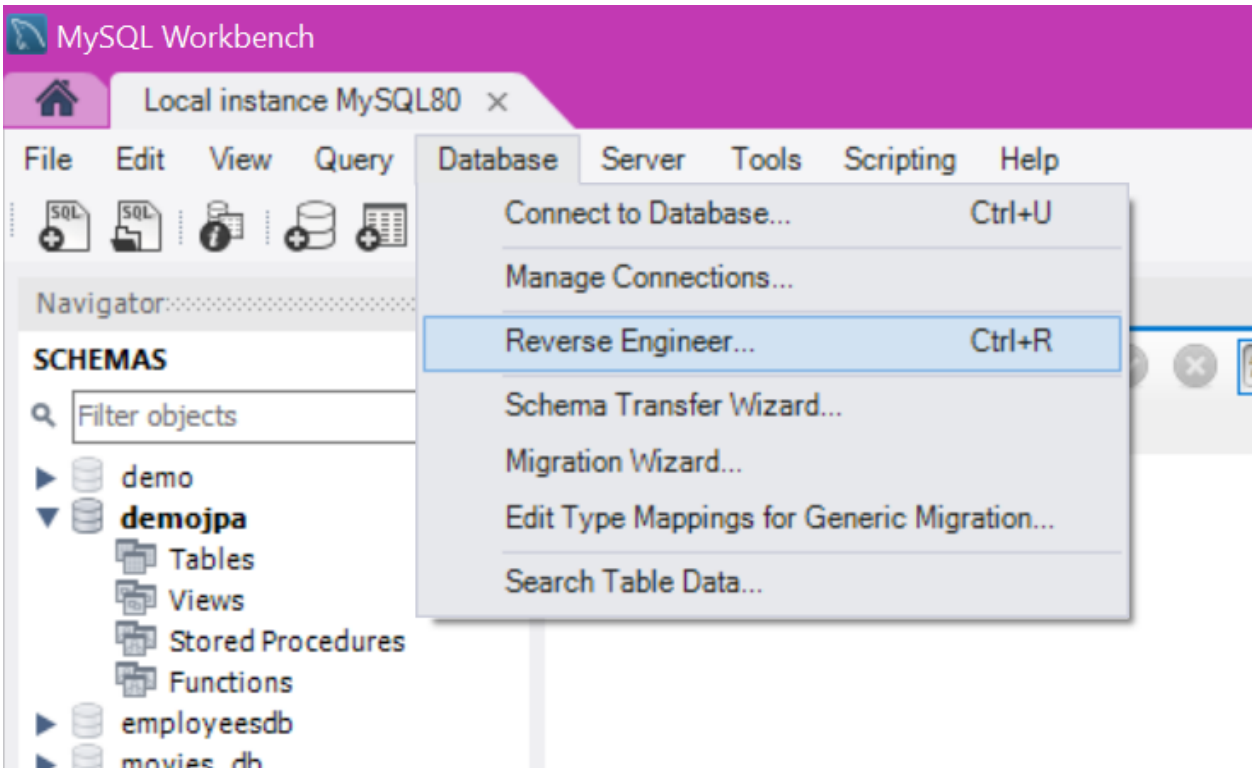
7- Crear relaciones entre las entidades :

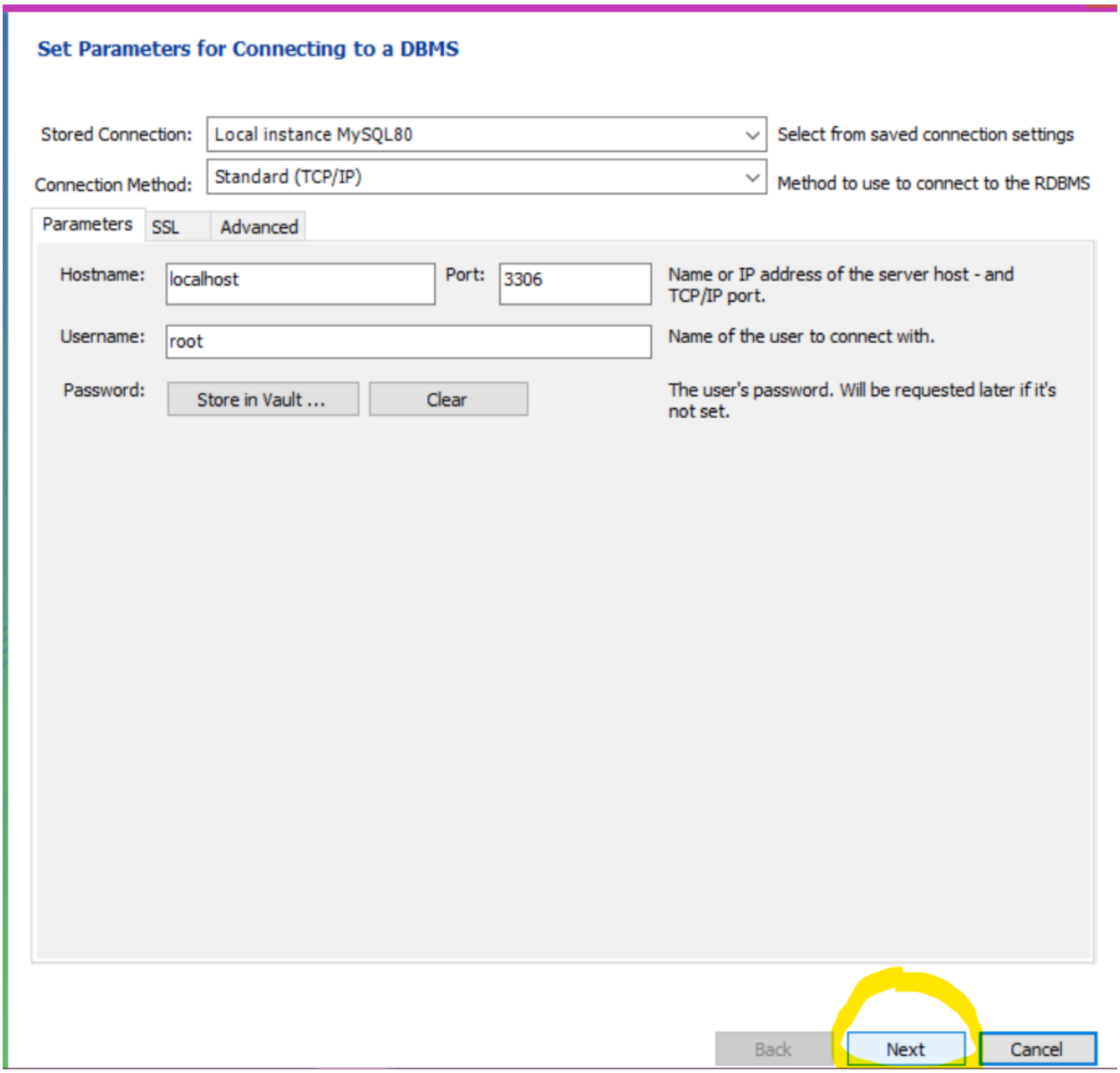
```
public class Mascota {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer id;

@Column(length = 2)
private Short edad;

@Column(length = 100)
private String tipoAnimal;

@Column(length = 100)
private String raza;

@Column(length = 100)
private String nombre;

@ManyToOne(cascade = {CascadeType.ALL})
private Duenio duenio;

@ManyToOne()
```

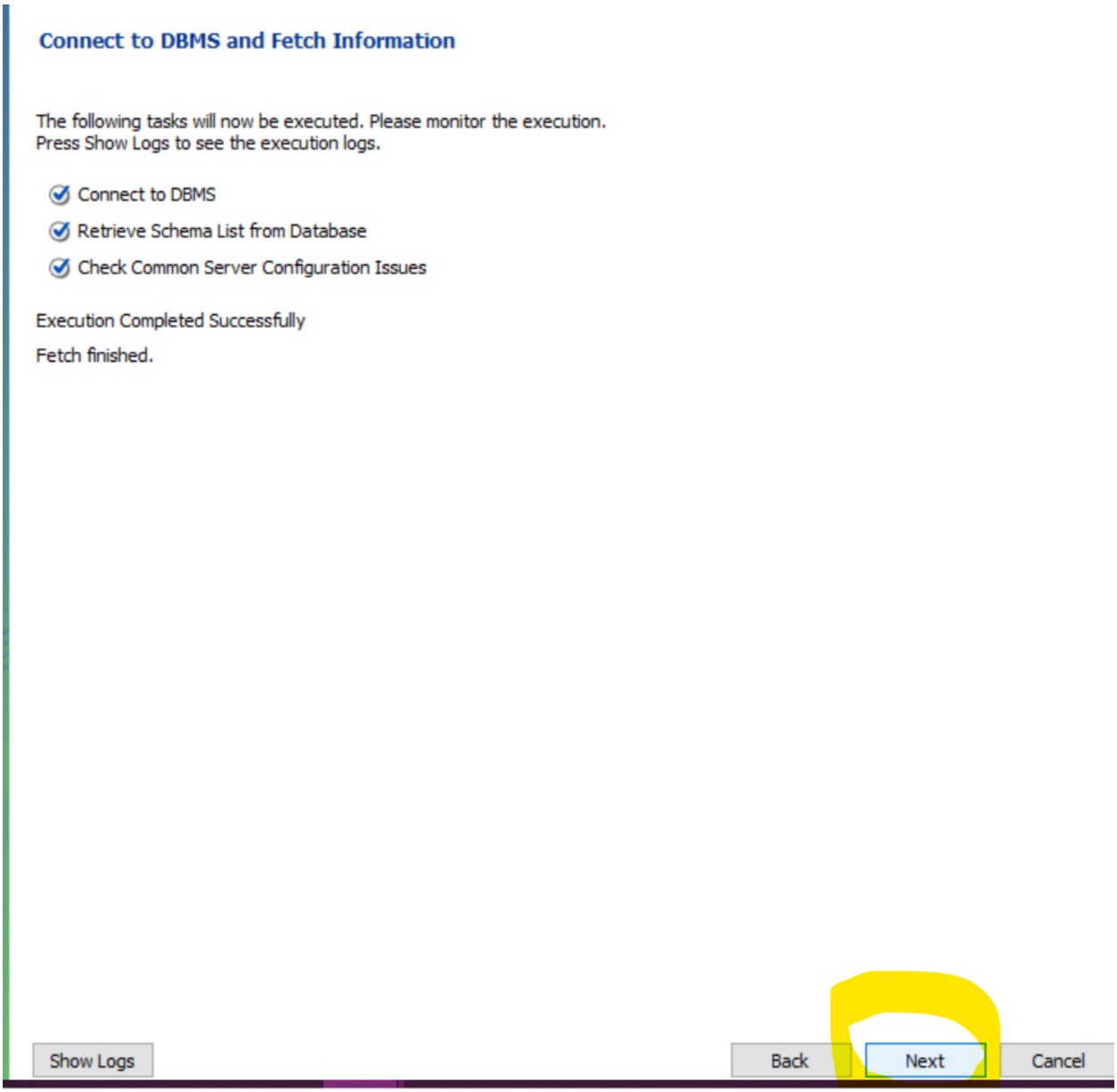
private Veterinario veterinario;

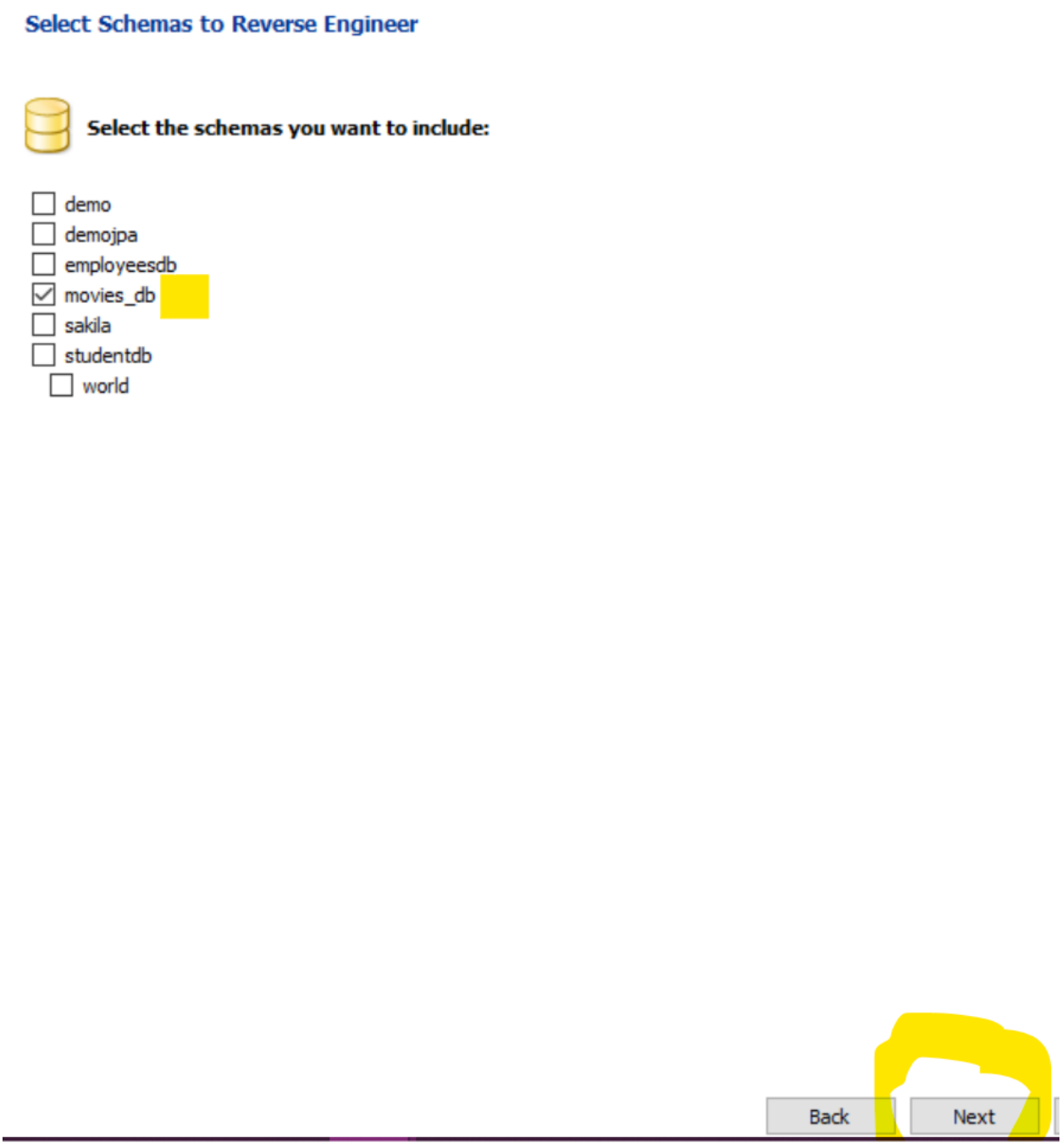
}

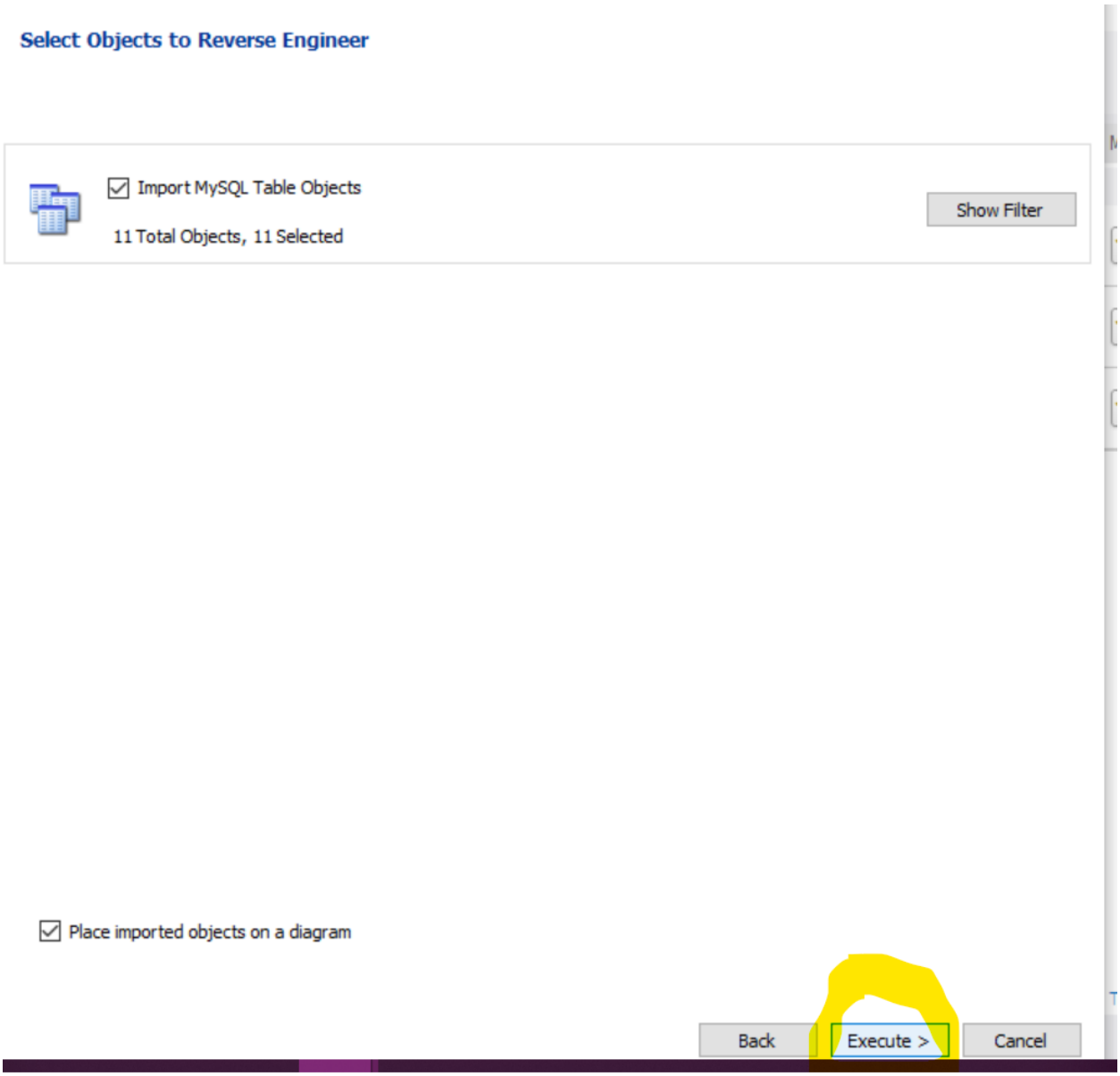
8- Diagrama Entidad-Relación (DER) completo de la base de datos implementada.

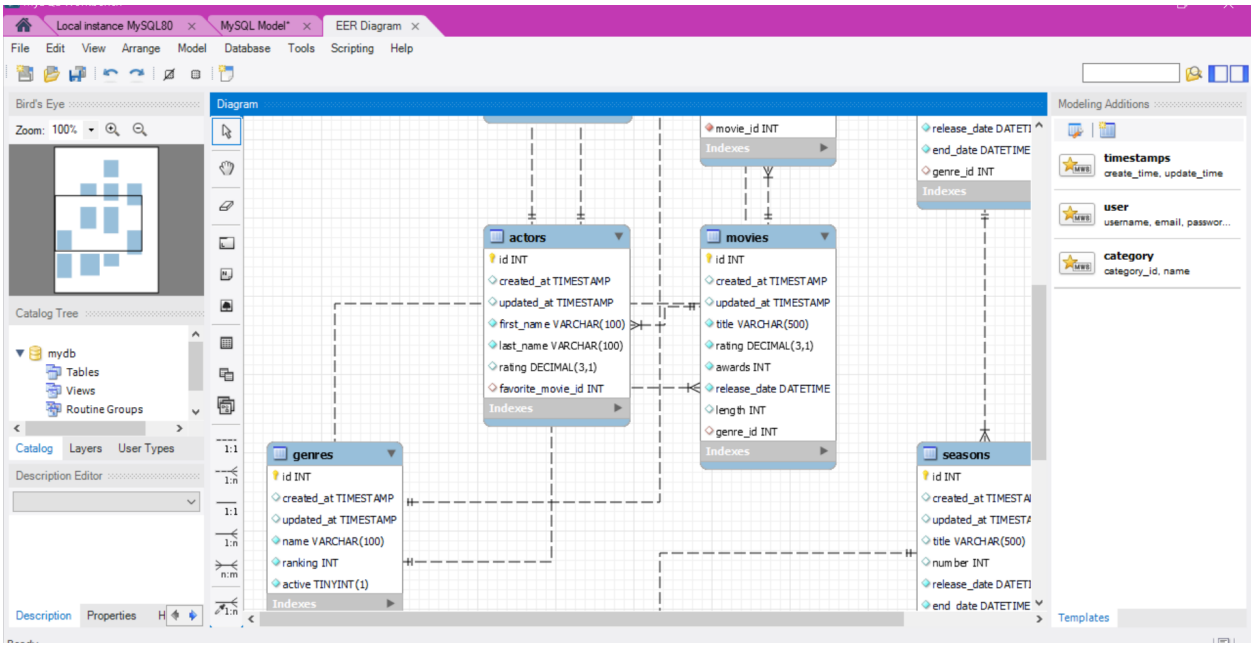



Set Parameters for Connecting to a DBMS
Stored Connection:
Local instance MySQL80
Select from saved connection settings
Connection Method:
Standard (TCP/IP)
Method to use to connect to the RDBMS
Parameters
SSL
Advanced
Hostname:
localhost
Port:
3306
Name or IP address of the server host - and TCP/IP port.
Username:
root
Name of the user to connect with.
Password:
Store in Vault ...
Clear
The user's password. Will be requested later if it's not set.
Back
Next
Cancel

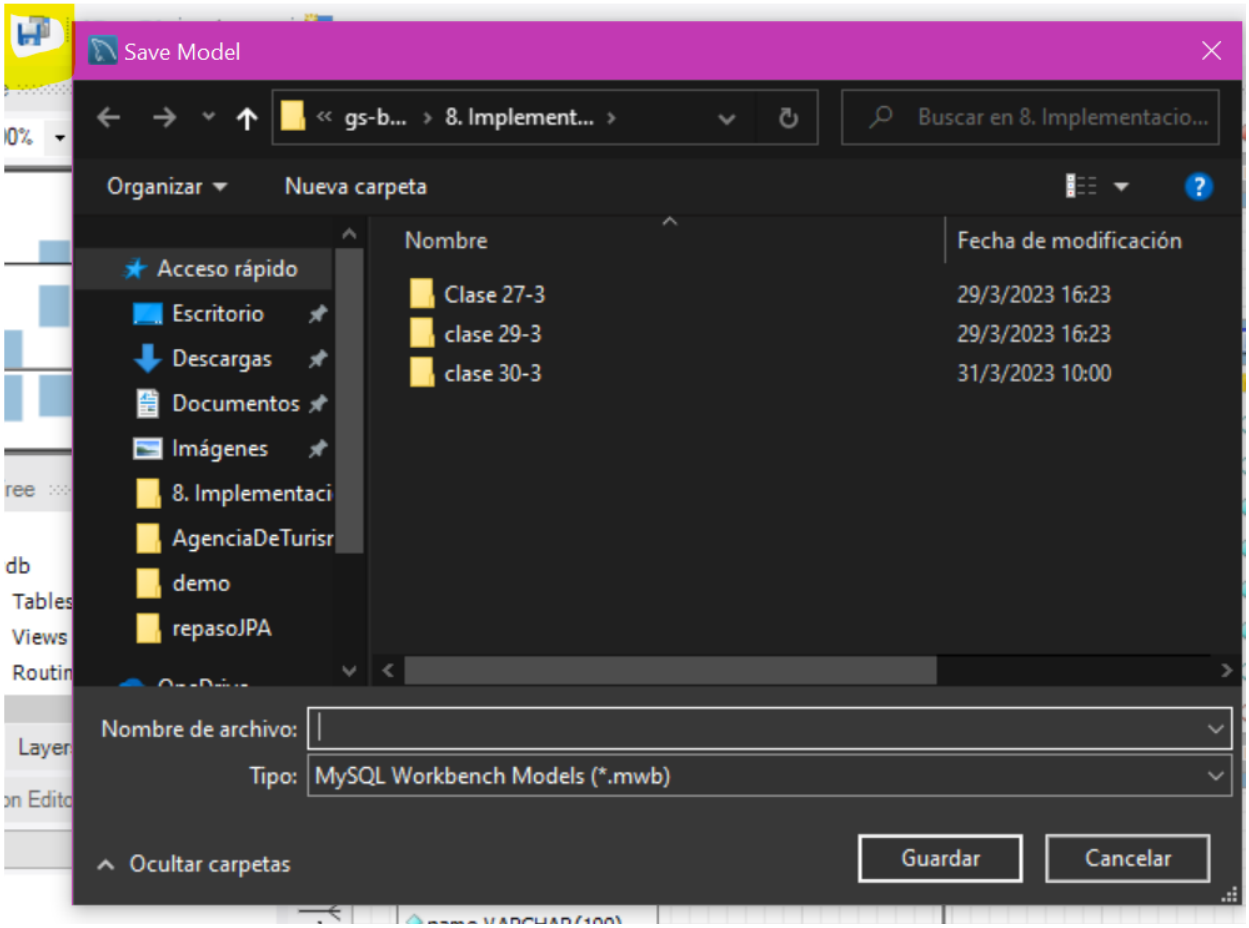

Connect to DBMS and Fetch Information
The following tasks will now be executed. Please monitor the execution.
Press Show Logs to see the execution logs.
Connect to DBMS
Retrieve Schema List from Database
Check Common Server Configuration Issues
Execution Completed Successfully
Fetch finished.
Show Logs
Back
Next
Cancel


Select Schemas to Reverse Engineer
Select the schemas you want to include:
demo
demojpa
employeesdb
movies_db
sakila
studentdb
world
Back
Next

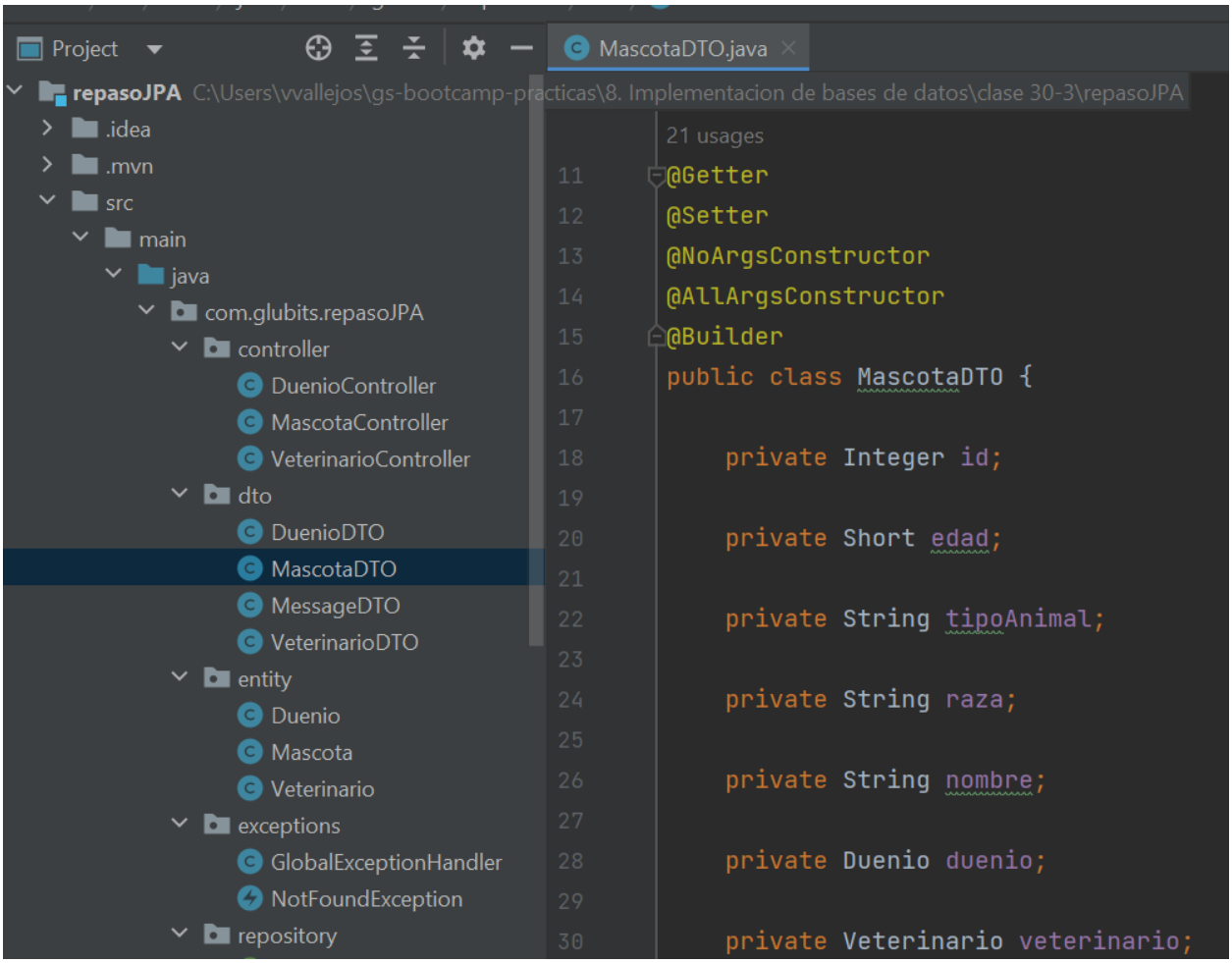

Select Objects to Reverse Engineer
Import MySQL Table Objects
11 Total Objects, 11 Selected
Show Filter
Place imported objects on a diagram
Back
Execute >
Cancel



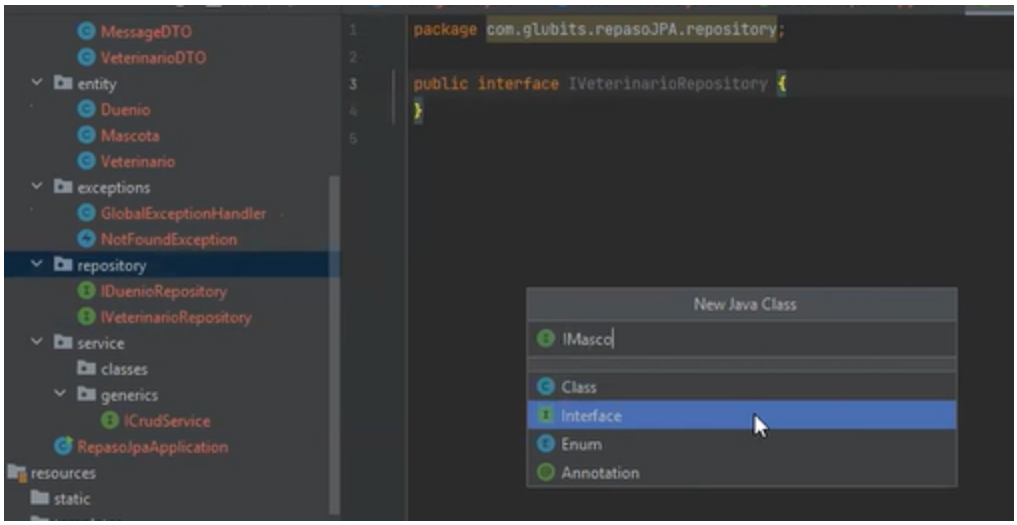
Guardar diagrama:



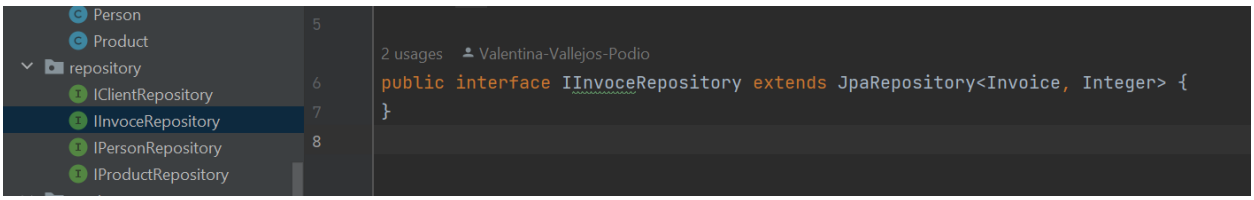
9- Creamos los DTO más basicos que se que vamos a necesitar:



10- Crear los repositorios:



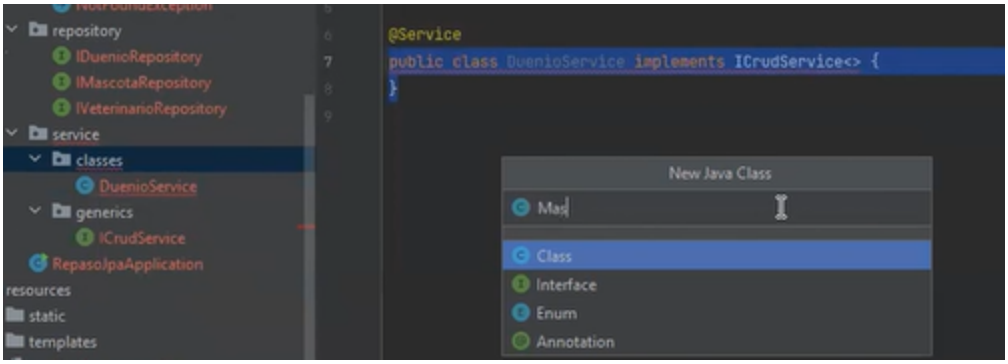
11- A todos les tengo que agregar que extienden de JpaRepository y pasarle (la clase, y el tipo de dato)



12- Creamos ICrudService: tiene los metodos principales que son comunes a todos los servicios.

```java
public interface ICrudService<T, ID> {

    3 usages   3 implementations
    T saveEntity(T objectDTO);

    3 usages   3 implementations
    List<T> getAllEntities();

    3 usages   3 implementations
    T getEntityById(ID id);

    3 usages   3 implementations
    MessageDTO deleteEntity(ID id);
}
```

13- Crear los servicios uno por cada entidad, deben implementar de ICrudService




```java
3 usages
@Service
public class VeterinarioService implements ICrudService<VeterinarioDTO, Integer>
```

14- Implementar los metodos del CRUD en los servicios:

Pero primero debemos colocar las anotaciones ,los repositorios y el mapper

```java
@Service
public class MascotaService implements ICrudService<MascotaDTO, Integer> {

    8 usages
    @Autowired
    IMascotaRepository mascotaRepository;

    1 usage
    @Autowired
    IDuenioRepository duenioRepository;

    6 usages
    ModelMapper mapper = new ModelMapper();
```

```java
@Override
public MascotaDTO saveEntity(MascotaDTO objectDTO) {
    // mappear de dto a entity para llevar al repo
    var entity = mapper.map(objectDTO, Mascota.class);

    // En este caso usamos cascada y por ende debemos ir a buscar la entidad duenio existente a la bbdd
    if(objectDTO.getDuenio().getId() != null)
        entity.setDuenio(
                duenioRepository.findById(objectDTO.getDuenio().getId()).get()
        );

    // guardar
    mascotaRepository.save(entity);

    // mappear de entity a dto para llevar al controller

    return mapper.map(entity, MascotaDTO.class);
}
```

```java
@Override
public List<MascotaDTO> getAllEntities() {
    // buscar todos los resultados en el repo
    var list = mascotaRepository.findAll();
    // luego convertir de entidad a DTO
    return list.stream().map(
                    mascota -> mapper.map(mascota, MascotaDTO.class)
            )
            .collect(Collectors.toList());
}
```

```java
@Override
public MascotaDTO getEntityById(Integer id) {
    // llamo al repo y le paso el id - me devuelve optional
    var opt = mascotaRepository.findById(id);

    // si optional esta vacio devuelvo excepcion. Si no, la entidad.
    var entity = opt.orElseThrow(
            () -> {
                throw new NotFoundException("No encontre ningun mascota con el id: " + id);
            }
    );
    // mapeo de entidad a dto.
    return mapper.map(entity, MascotaDTO.class);
}
```

```java
@Override
public MessageDTO deleteEntity(Integer id) {
    // buscar el dato en la base de datos y asegurarnos que exista
    var exists = mascotaRepository.existsById(id);
    // eliminar efectivamente
    if(exists)
        mascotaRepository.deleteById(id);
    else
        throw new NotFoundException("No pude encontrar la mascota con id " + id);
    // devolver el mensaje DTO
    return MessageDTO.builder()
            .message("Se elimino la mascota con id" + id)
            .action("ELIMINACION")
            .build();
}
```

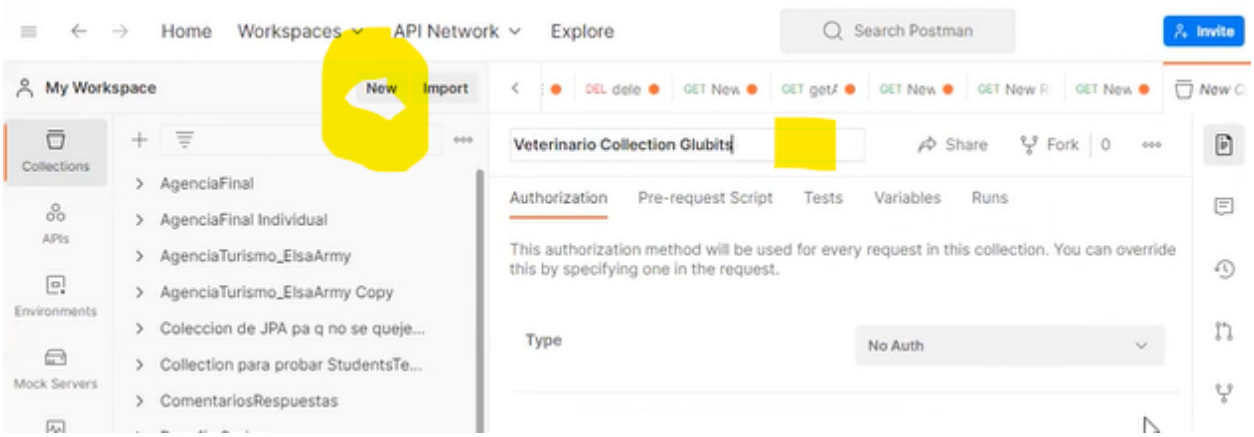
15- Generamos los controladores:

```java
@RestController
@RequestMapping("mascota")
public class MascotaController {

    7 usages
    @Autowired
    MascotaService mascotaService;
```

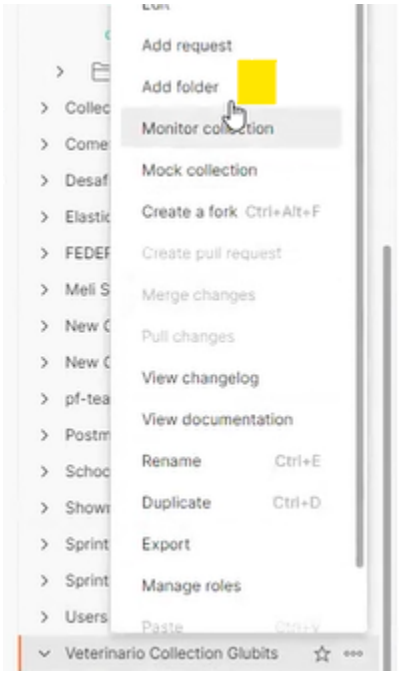
```
no usages
@PostMapping("/create")
public ResponseEntity<MascotaDTO> create(@RequestBody MascotaDTO dto){
    return ResponseEntity.ok(
            mascotaService.saveEntity(dto)
    );
}

no usages
@GetMapping("/")
public ResponseEntity<List<MascotaDTO>> findAll(){
    return ResponseEntity.ok(
            mascotaService.getAllEntities()
    );
}
```

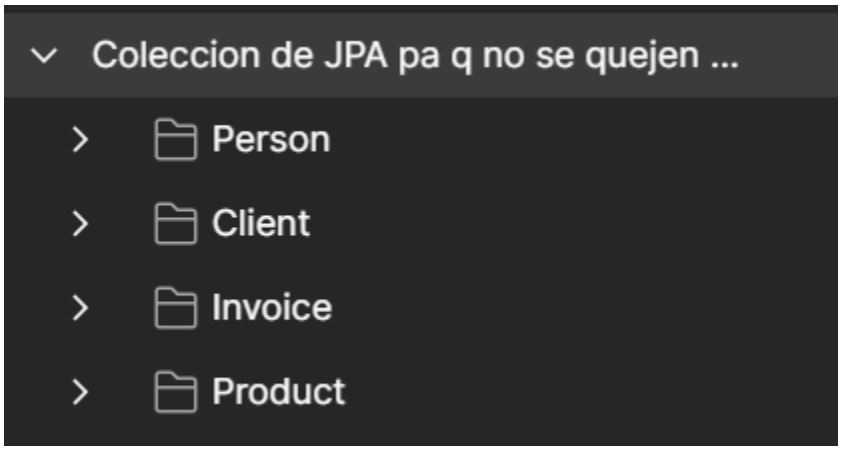
```
@GetMapping("/{id}")
public ResponseEntity<MascotaDTO> findById(@PathVariable Integer id){
    return ResponseEntity.ok(
            mascotaService.getEntityById(id)
    );
}

no usages
@DeleteMapping("/{id}")
public ResponseEntity<MessageDTO> deleteById(@PathVariable Integer id){
    return ResponseEntity.ok(
            mascotaService.deleteEntity(id)
    );
}
```

16- A medida que vamos creando una por una, vamos probando en postman: Para esto debemos crear una nuevo collections.

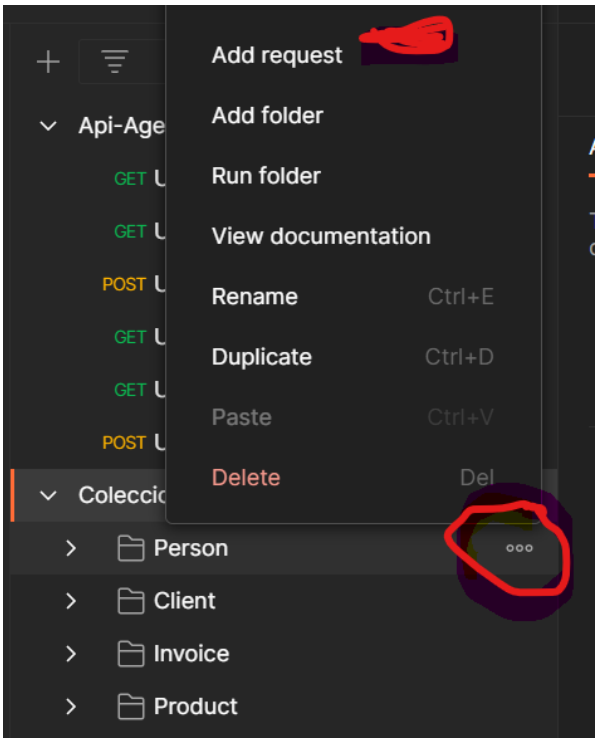


17- Dentro de este collections, una buena practica es crear folders (carpetas) , para poder dividir las rutas, según los controllers.

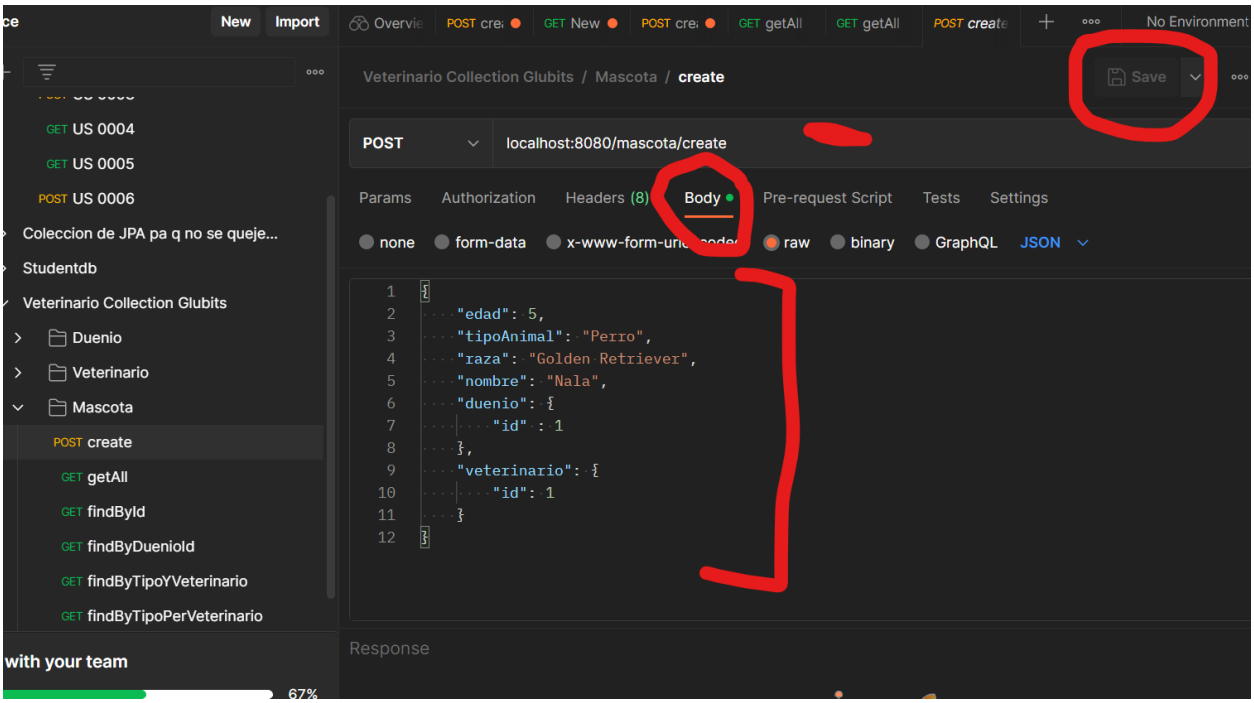




18- Dentro de los folders creamos los nuevos request:

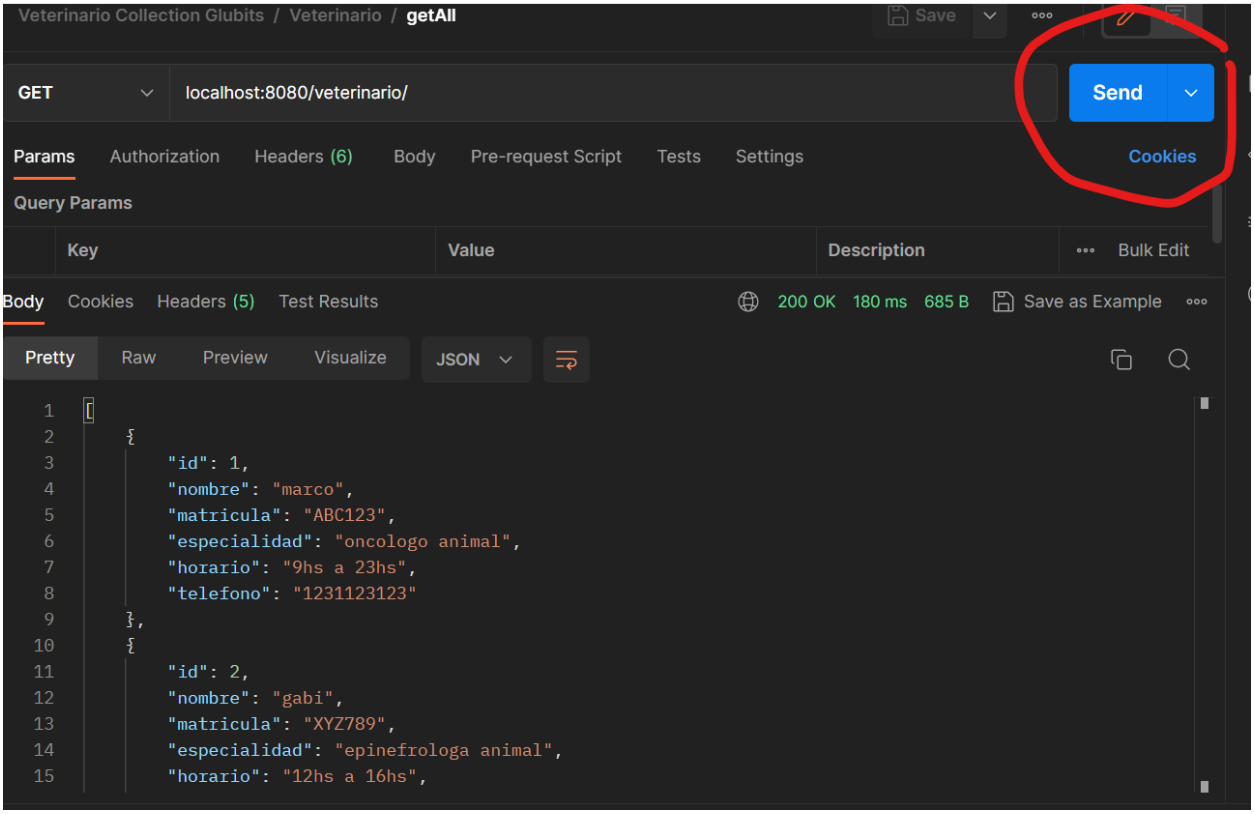



```java
@Builder
public class MascotaDTO {

    private Integer id;

    private Short edad;

    private String tipoAnimal;

    private String raza;

    private String nombre;

    private Duenio duenio;

    private Veterinario veterinario;

}
```

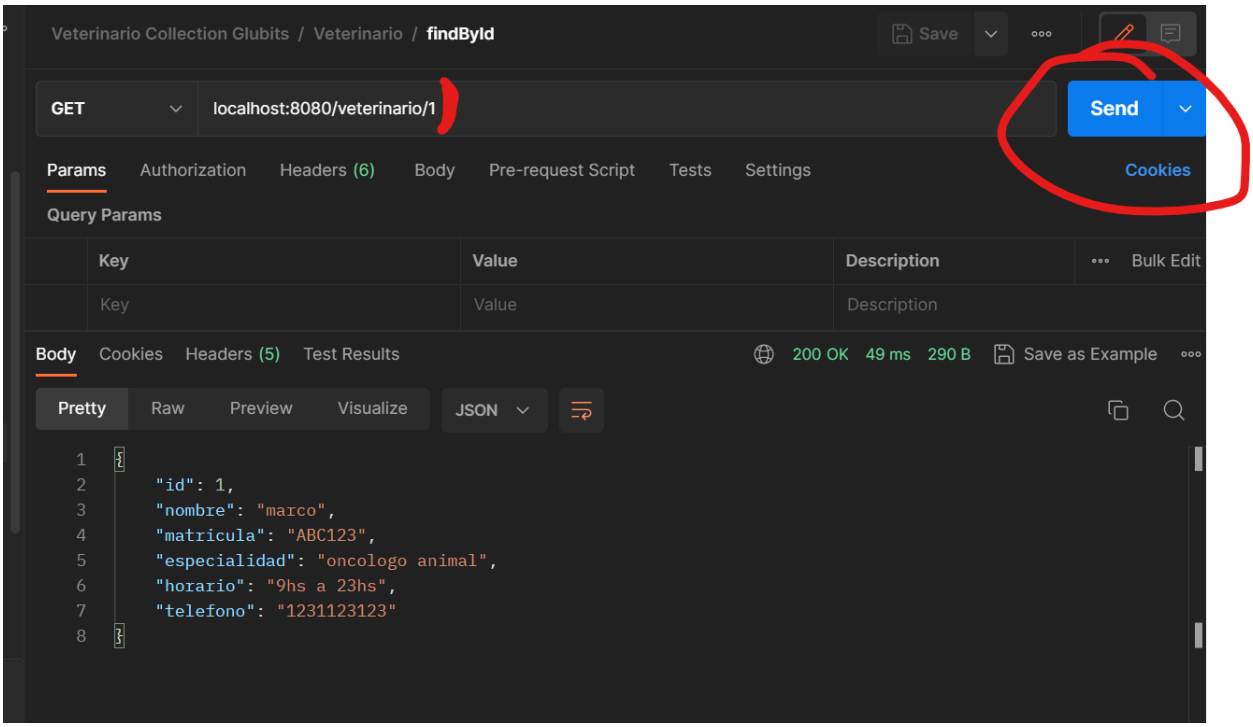


el save : es para que se guarde

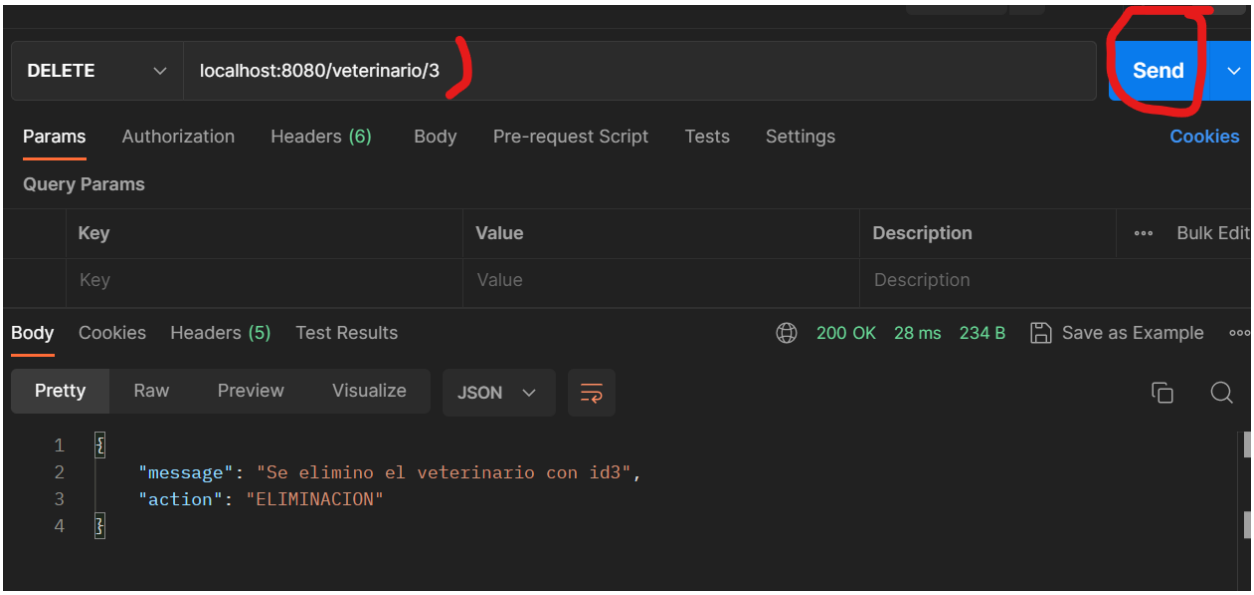
lo del body: acá se le coloca en formato json, lo que esta en el DTO. En este caso por ser Create la url , no se le agrega el ID. Esto es para cuando se implementa la url de save.



Caso getAll le das a send para que te traiga toda la lista de veterinarios, previamente deben estar creados.



Caso findById: en este caso en la url como se puede ver colocamos el 1, para que nos traiga al veterinario con el id 1. Y enviamos send



Caso eliminar: Es igual que el anterior en la url se le coloca el id que deseamos eliminar y mandamos send. Para corroborar que se elimino podemos volver a enviar la

url de getAll y ahi verificamos que al traernos la lista de todos los veterinarios nos trae todos menos el que eliminamos.

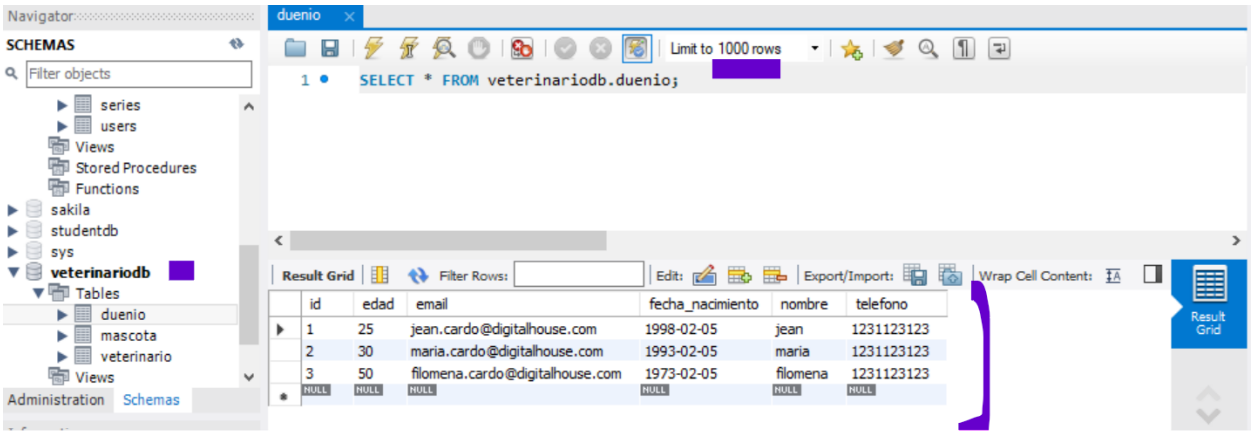
`DATOS EXTRAS PARA LOS MENSAJES DE ERROR:`

```
no usages
@ExceptionHandler(NotFoundException.class)
public ResponseEntity<MessageDTO> notFoundException(NotFoundException e){
    return ResponseEntity.status(404).body(
            MessageDTO.builder()
                    .message(e.getMessage())
                    .action("BUSQUEDA")
                    .build()
    );
}
```

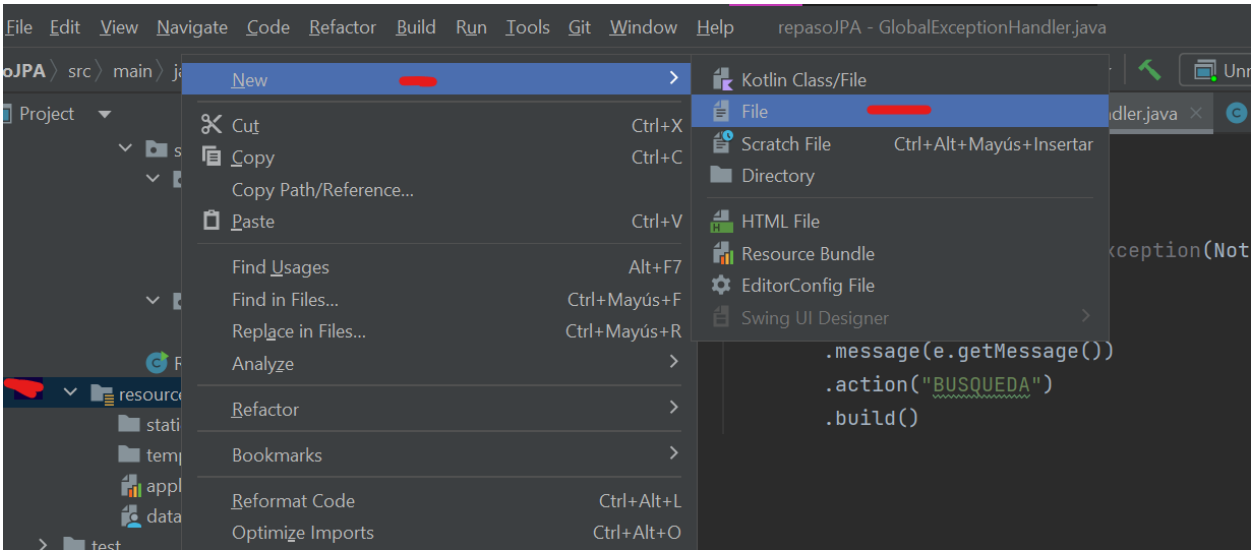
```
@Override
public MascotaDTO getEntityById(Integer id) {
    // llamo al repo y le paso el id - me devuelve optional
    var opt = mascotaRepository.findById(id);

    // si optional esta vacio devuelvo excepcion. Si no, la entidad.
    var entity = opt.orElseThrow(
            () -> {
                throw new NotFoundException("No encontre ningun mascota con el id: " + id);
            }
    );
    // mapeo de entidad a dto.
    return mapper.map(entity, MascotaDTO.class);
}
```

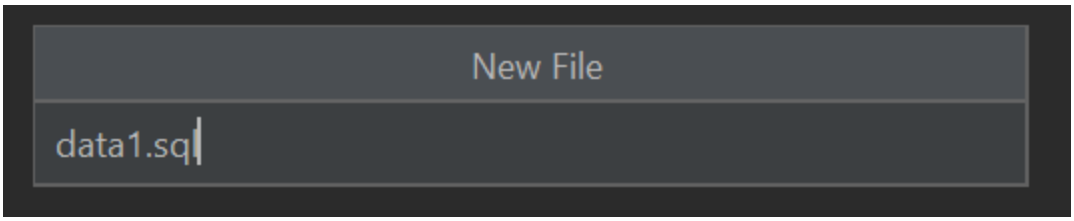
`Acá podemos ver implementado este mensaje.`

19- Verificamos que todo este creado,eliminado…. en la base da datos.



20- Crear un set de datos:





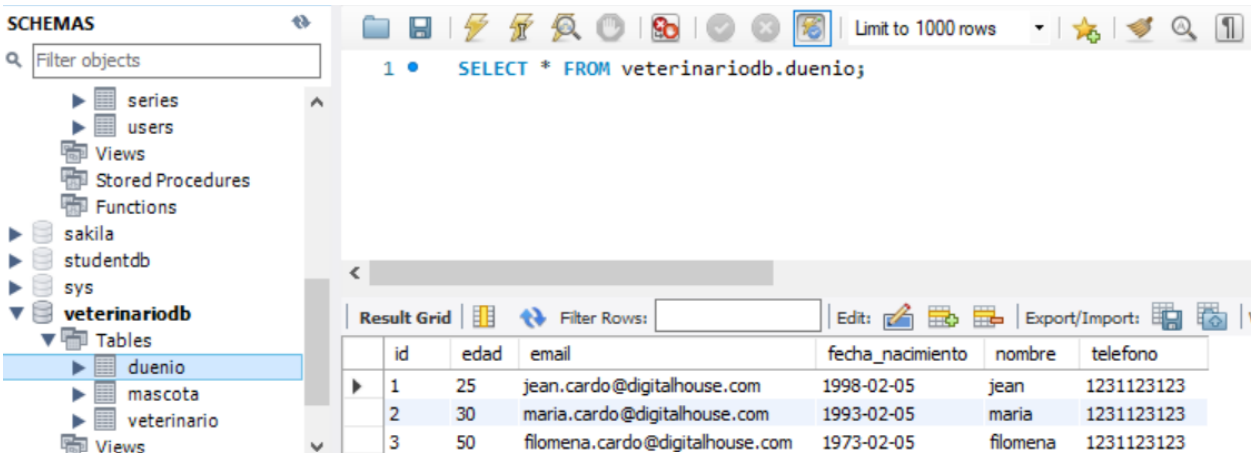
Y generamos:

```sql
INSERT INTO duenio (edad, email, fecha_nacimiento, nombre, telefono)
VALUES(25, 'jean.cardo@digitalhouse.com', '1998-02-05', 'jean', '1231123123');

INSERT INTO duenio (edad, email, fecha_nacimiento, nombre, telefono)
VALUES(30, 'maria.cardo@digitalhouse.com', '1993-02-05', 'maria', '1231123123');

INSERT INTO duenio (edad, email, fecha_nacimiento, nombre, telefono)
VALUES(50, 'filomena.cardo@digitalhouse.com', '1973-02-05', 'filomena', '1231123123');


INSERT INTO veterinario (especialidad, horario, matricula, nombre, telefono)
VALUES('oncologo animal', '9hs a 23hs', 'ABC123', 'marco', '1231123123');

INSERT INTO veterinario (especialidad, horario, matricula, nombre, telefono)
VALUES('epinefrologa animal', '12hs a 16hs', 'XYZ789', 'gabi', '1231123123');



INSERT INTO mascota (edad, nombre, raza, tipo_animal, duenio_id, veterinario_id)
VALUES (5, 'nala', 'golden retriver', 'perro', 1, 1);

INSERT INTO mascota (edad, nombre, raza, tipo_animal, duenio_id, veterinario_id)
VALUES (25, 'negra', 'pastor aleman', 'perro', 1, 2);
```

Para que esto se agregue a la base da tos esto debe estar no comentado en el properties:LU

```
spring.jpa.show-sql=true
## Estas tres lineas son para levantar un archivo SQL
spring.sql.init.mode=always
spring.sql.init.data-locations=classpath:data.sql
spring.jpa.defer-datasource-initialization=true
```

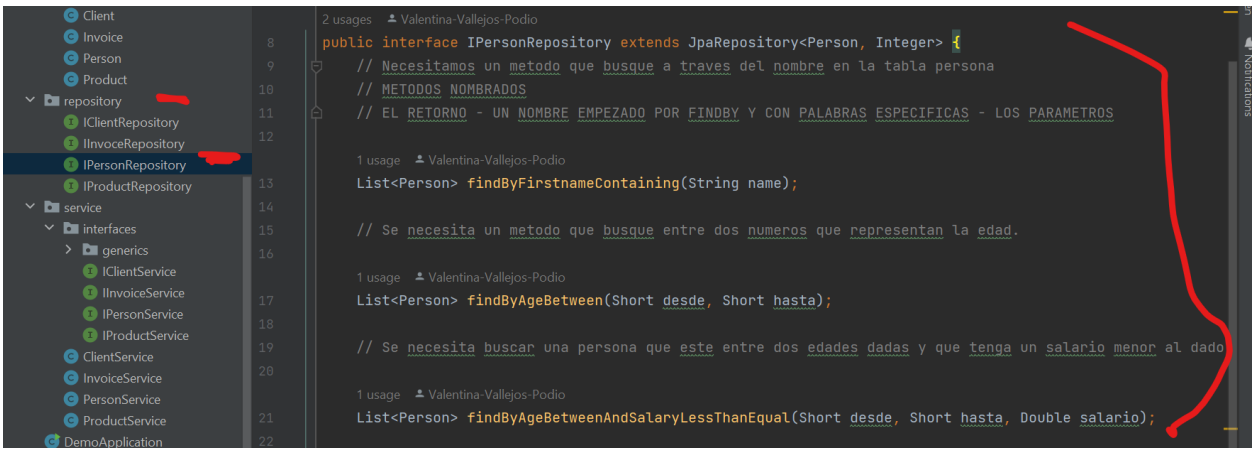
Luego podemos correr el servidor y verificar que este agregada la informacion:

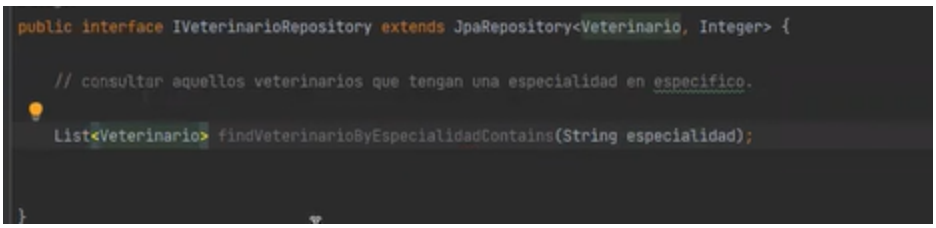


Coincide ya que hay 3 dueños insertados en data.sql

21- Consultas en el repository:

Consultas nombradas:





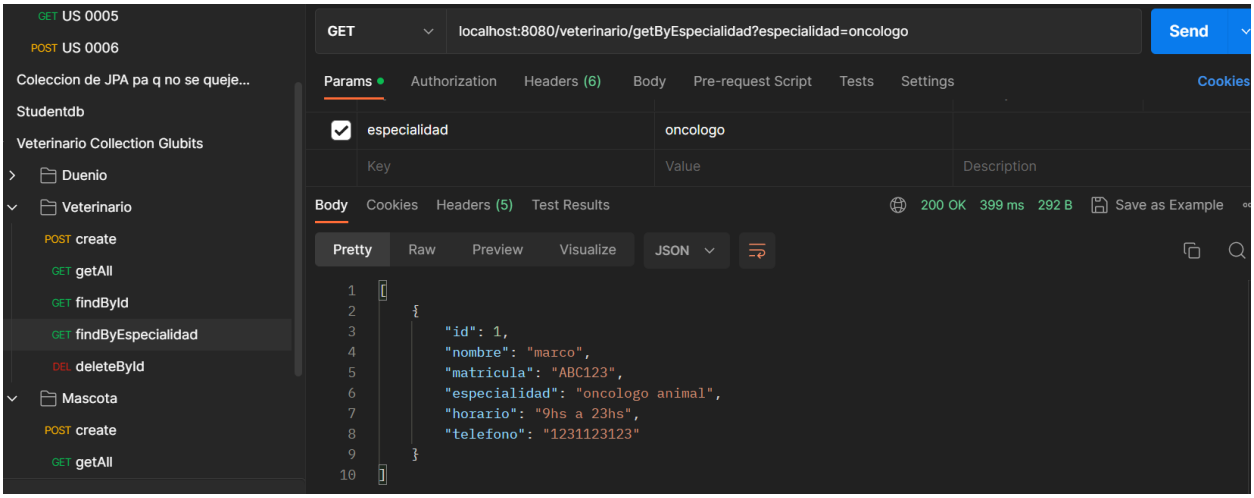
Para poder implementarlas debo crear en el services:

```java
public List<VeterinarioDTO> findByEspecialidad(String especialidad){

    var list :List<Veterinario> = veterinarioRepository.findVeterinarioByEspecialidadContains(especialidad);

    return list.stream().map(
            veterinario -> mapper.map(veterinario, VeterinarioDTO.class)
    )
            .collect(Collectors.toList());
}
```

Y crear el controller:

```java
@GetMapping("/getByEspecialidad")
public ResponseEntity<List<VeterinarioDTO>> getByEspecialidad(@RequestParam String especialidad){
    return ResponseEntity.ok(
            veterinarioService.findByEspecialidad(especialidad)
    );
}
```

Y lo verificamos en postman:



Consultas con HQL:

En el repository:

```java
// IMascotaRepository.java
package com.glubits.repasoJPA.repository;

import ...

2 usages
public interface IMascotaRepository extends JpaRepository<Mascota, Integer> {

    // quiero pedir todas las mascotas que tienen un cierto id de duenio

    1 usage
    @Query("FROM Mascota m WHERE m.duenio.id = :id")
    List<Mascota> findMascotaByDuenioIdEquals(@Param("id") Integer id);
```

En el service

```java
1 usage
public List<MascotaDTO> findByDuenioId(Integer id){
    var list = mascotaRepository.findMascotaByDuenioIdEquals(id);

    return list.stream().map(
                    mascota -> mapper.map(mascota, MascotaDTO.class)
            )
            .collect(Collectors.toList());
}
```

En el controller:

```java
no usages
@GetMapping("/getByDuenioId")
public ResponseEntity<List<MascotaDTO>> getByDuenioId(@RequestParam Integer id){
    return ResponseEntity.ok(
            mascotaService.findByDuenioId(id)
    );
}
```

Prueba en postman: