

Diseño y Desarrollo de Sistemas de Información

Mapeo Objeto Relacional
(JPA - *Hibernate*)

Objetivos

- Aprender a utilizar una implementación de una herramienta de Mapeo Objeto Relacional (ORM)
- Aprender a implementar operaciones CRUD en Java utilizando una herramienta de ORM

Mapeo Objeto Relacional. Conceptos y definiciones

- Un ORM es un modelo de programación que permite "mapear" las estructuras de una **base de datos relacional** (Oracle, MariaDB, etc.) en **entidades lógicas** para simplificar y acelerar el desarrollo de las aplicaciones
- Las estructuras de la base de datos relacional quedan vinculadas con las entidades lógicas o **base de datos virtual** definida en el ORM, de tal modo que las operaciones **CRUD** (*Create, Read, Update, Delete*) se realizan, de forma indirecta, a través del ORM

- Las herramientas de ORM, además de generar código de forma automática, suelen utilizar un lenguaje propio para realizar las consultas y gestionar la persistencia de los datos
- Los objetos o entidades de la base de datos virtual creada por el ORM son gestionados con lenguajes de propósito general
- La interacción con el SGBD se realiza mediante los métodos propios del ORM
- Interactuar directamente con las entidades de la base de datos virtual sin necesidad de escribir código SQL puede generar importantes ventajas a la hora de acelerar el desarrollo o implementación de las aplicaciones

Ventajas e inconvenientes de los ORM

■ Ventajas

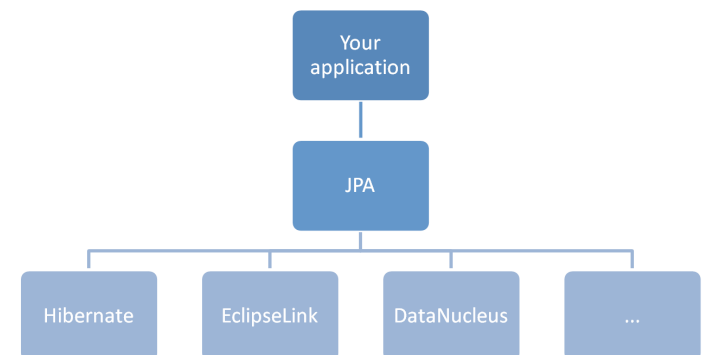
- No es necesario escribir "mucho" código SQL. Algunos programadores no lo dominan y, a veces, puede resultar complejo y propenso a errores
- Permite aumentar la reutilización del código y mejorar el mantenimiento del mismo
- Reduce el tiempo de desarrollo
- Mayor seguridad, evitando posibles ataques de inyección SQL y similares
- Están bien optimizados para las operaciones CRUD *insert*, *delete* y *update*, aunque no algo menos para las operaciones de recuperación (*select*)

■ Inconvenientes

- En entornos con gran carga puede reducir el rendimiento, ya que se está agregando una capa extra al sistema
- Su aprendizaje puede llegar a ser complejo
- Si las consultas son complejas, el ORM no garantiza una buena optimización, tal y como lo realizaría un desarrollador de forma nativa. De ahí que las implementaciones de ORM ofrezcan extensiones para escribir consultas en SQL nativo

¿Qué es JPA? ¿Qué es Hibernate?

- **JPA** (*Java Persistence API*) es la especificación que define el funcionamiento de la persistencia de objetos en Java. Por ejemplo, qué anotaciones se deben usar, cómo se deben persistir los objetos, etc.
- JPA es un documento, no una implementación. Para poder trabajar con esta API se necesita un *framework* que implemente la especificación
- **Hibernate** es uno de los *frameworks* que implementa la especificación JPA. Hay otros como *EclipseLink*, *DataNucleus*, *TopLink*, etc.



Diferencia entre JDBC e Hibernate

```
...  
ps = conexion.getConnection().prepareStatement("INSERT INTO MONITOR VALUES (?, ?, ?, ?, ?, ?, ?)");  
ps.setString(1, monitor.getCodMonitor());  
ps.setString(2, monitor.getNombre());  
ps.setString(3, monitor.getDni());  
ps.setString(4, monitor.getTelefono());  
ps.setString(5, monitor.getCorreo());  
ps.setString(6, monitor.getFechaEntrada());  
ps.setString(7, monitor.getNick());  
  
ps.executeUpdate();  
ps.close();  
...
```

```
...  
Session sesion = HibernateUtil.getSessionFactory().openSession();  
Transaction transaction = sesion.beginTransaction();  
  
sesion.save(monitor);  
  
transaction.commit();  
  
sesion.close();  
...
```

Dependencias MAVEN

- Añadiremos al proyecto (pom.xml) estas 2 dependencias

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.6.15.Final</version>
</dependency>
```

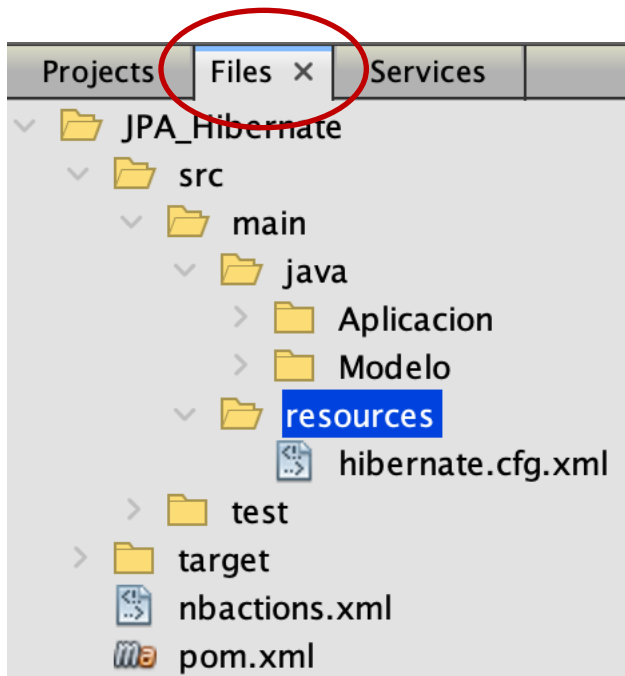
→ Dependencia para trabajar con Hibernate

```
<dependency>
  <groupId>javax.persistence</groupId>
  <artifactId>javax.persistence-api</artifactId>
  <version>2.2</version>
</dependency>
```

Dependencia para trabajar con la persistencia de Java

Fichero de configuración de Hibernate

- Hibernate utiliza un fichero de configuración, denominado **hibernate.cfg.xml**
- Este fichero debe situarse en la carpeta raíz del proyecto. Concretamente en **src/main/resources**



- Si no existe la carpeta *resources*, deberá crearse
- Una vez en la carpeta, con el botón derecho, seleccionar "Nuevo" + "Fichero xml"

Fichero de configuración de Hibernate

Oracle

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">oracle.jdbc.OracleDriver</property>
    <property name="hibernate.connection.url">jdbc:oracle:thin:@172.17.20.39:1521:etsi</property>
    <property name="hibernate.connection.username">usuario</property>
    <property name="hibernate.connection.password">contraseña</property>
    <property name="hibernate.show_sql">>true</property>
    <mapping class="Modelo.Monitor"/>
    <mapping class="Modelo.Actividad"/>
    <mapping class="Modelo.Socio"/>
  </session-factory>
</hibernate-configuration>
```

- Las etiquetas <mapping class> definen las clases que se mapean con las tablas de la base de datos. Se generarán automáticamente mediante una herramienta de NetBeans
- Las tablas que surgen de las relaciones "muchos a muchos" no se mapean en la aplicación (su funcionamiento se verá más adelante)

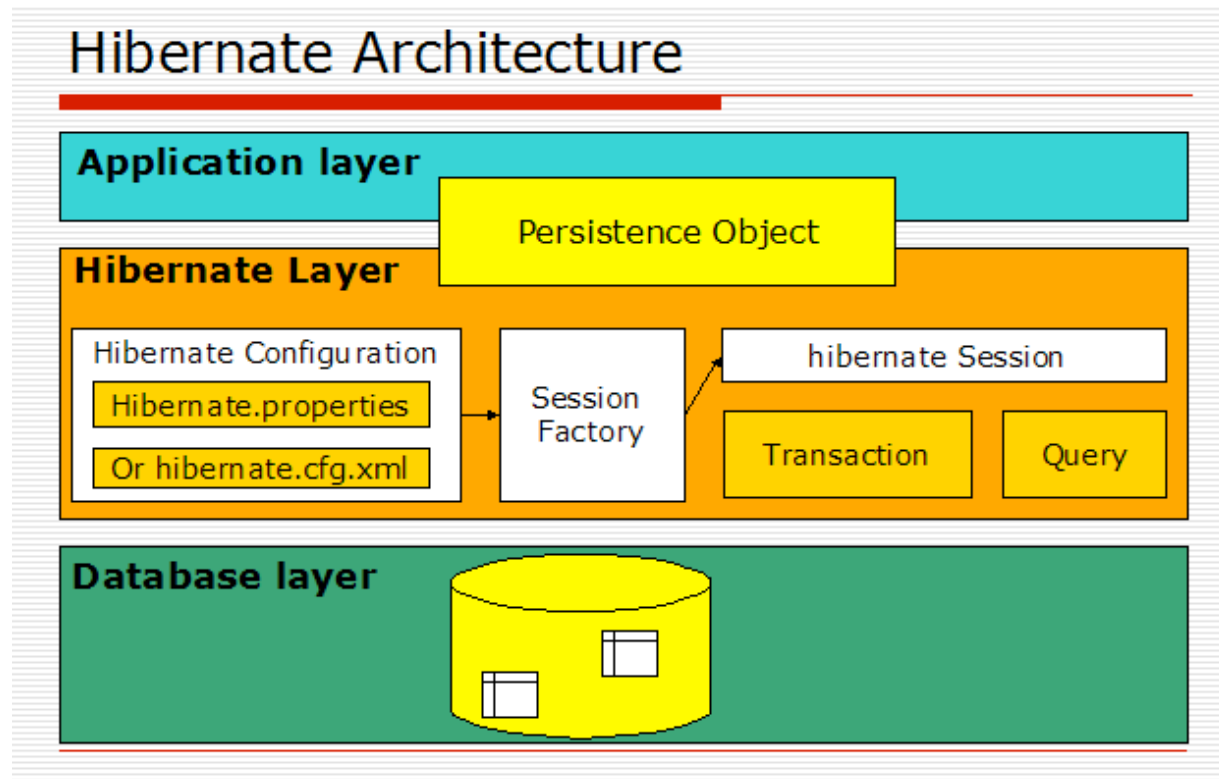
Fichero de configuración de Hibernate

MariaDB

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">org.mariadb.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mariadb://172.18.1.241:3306/DDSI_001</property>
    <property name="hibernate.connection.username">usuario</property>
    <property name="hibernate.connection.password">contraseña</property>
    <property name="hibernate.show_sql">true</property>
    <mapping class="Modelo.Monitor"/>
    <mapping class="Modelo.Actividad"/>
    <mapping class="Modelo.Socio"/>
  </session-factory>
</hibernate-configuration>
```

Conexión y comunicación entre la aplicación y la base de datos

- Las clases que establecen la conexión y la comunicación entre la base de datos y la aplicación son *SessionFactory* y *Session*
- La clase *SessionFactory*, junto con sus métodos principales, suele crearse en una clase llamada *HibernateUtil.java*, que situaremos en un paquete del proyecto (por ejemplo, *Config*)

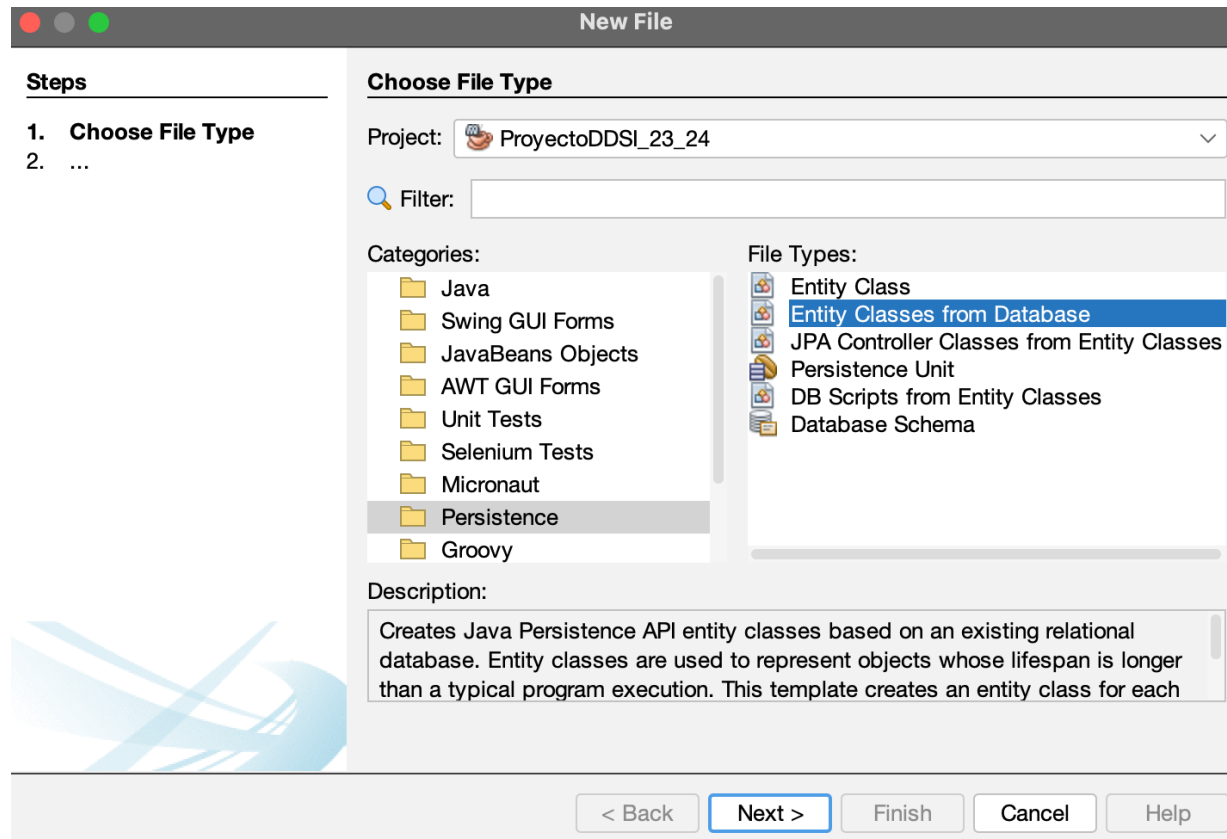


■ Ejemplo de *HibernateUtil.java*

```
public class HibernateUtilOracle {  
  
    private static final SessionFactory sessionFactory = buildSessionFactory();  
  
    private static SessionFactory buildSessionFactory() {  
        try {  
            ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()  
                .configure("hibernateOracle.cfg.xml").build();  
  
            Metadata metadata = new MetadataSources(serviceRegistry).getMetadataBuilder().build();  
  
            return metadata.getSessionFactoryBuilder().build();  
  
        } catch (Throwable ex) {  
            System.err.println ("Build SeesionFactory failed : " + ex);  
            throw new ExceptionInInitializerError(ex);  
        }  
    }  
  
    public static SessionFactory getSessionFactory() { return sessionFactory; }  
  
    public static void close() {  
        if ((sessionFactory!=null) && (sessionFactory.isClosed()==false)) {  
            sessionFactory.close();  
        }  
    }  
}
```

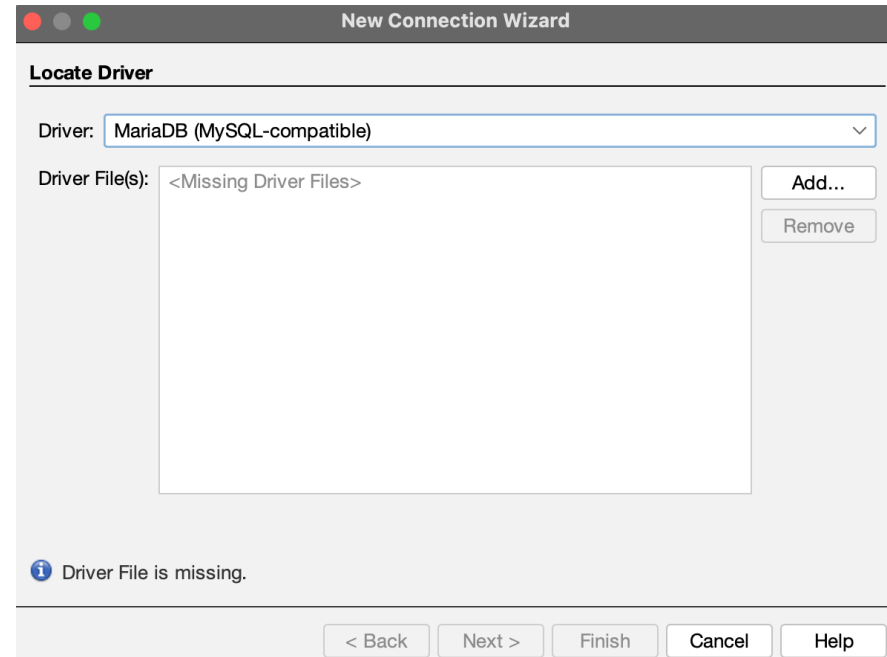
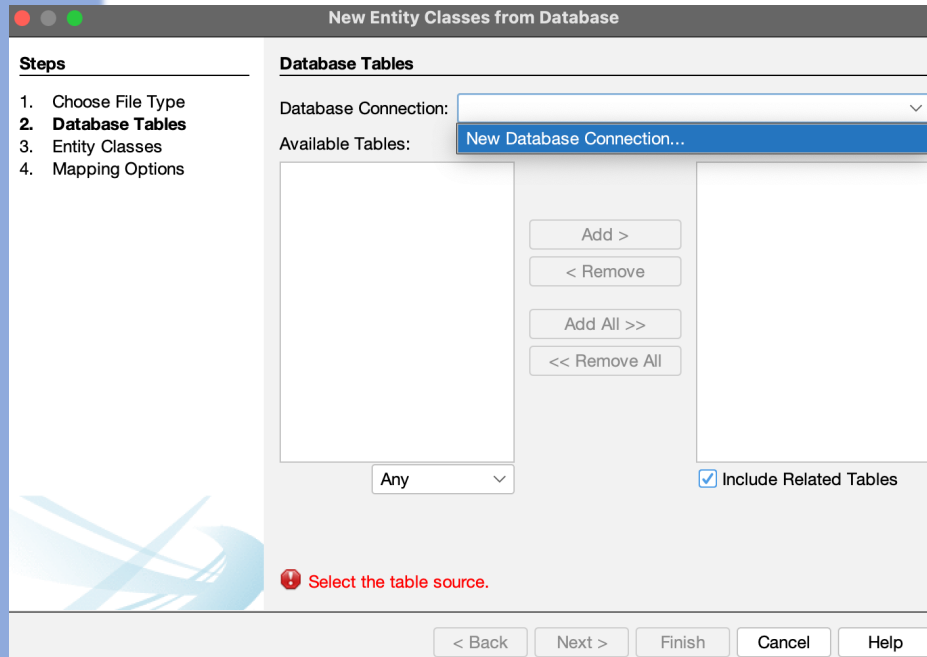
Creación de las clases derivadas de las tablas de la BD

- El siguiente paso consiste en crear las clases sobre las que mapearemos las tablas de la base de datos
- Como casi siempre, se puede hacer de forma manual o con alguna utilidad existente. En este caso, nos apoyaremos en una funcionalidad de **NetBeans**
- La funcionalidad se llama "**Entity classes from databases**", que se encuentra dentro del módulo "**Persistence**"

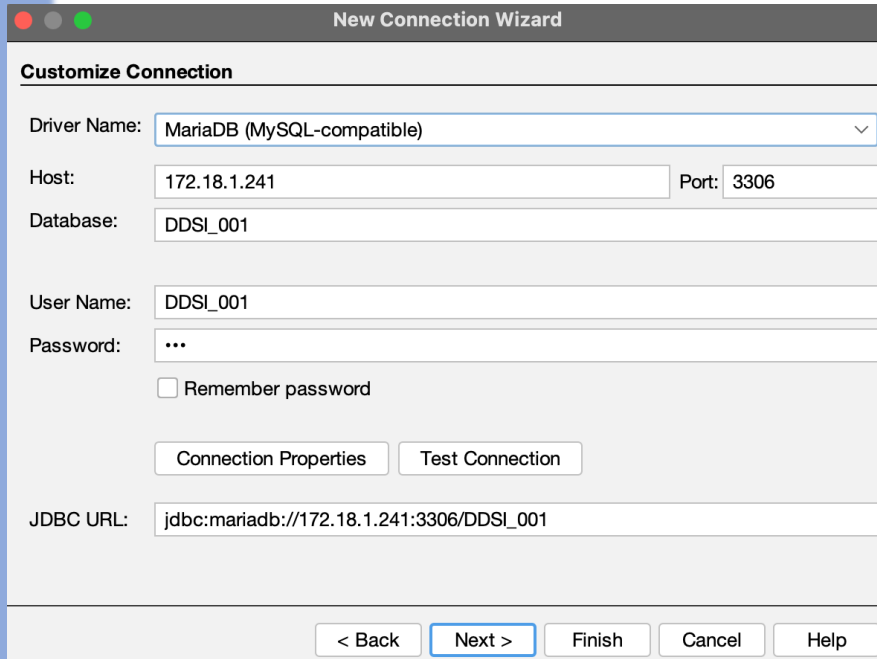


Creación de las clases derivadas de las tablas de la BD

- Para mapear las tablas debemos elegir la conexión a uno de los dos servidores de base de datos
- Si en *Netbeans* aún no hemos creado ninguna conexión, se puede crear en este paso. Por ejemplo, para MariaDB



Creación de las clases derivadas de las tablas de la BD



New Connection Wizard

Customize Connection

Driver Name:

Host: Port:

Database:

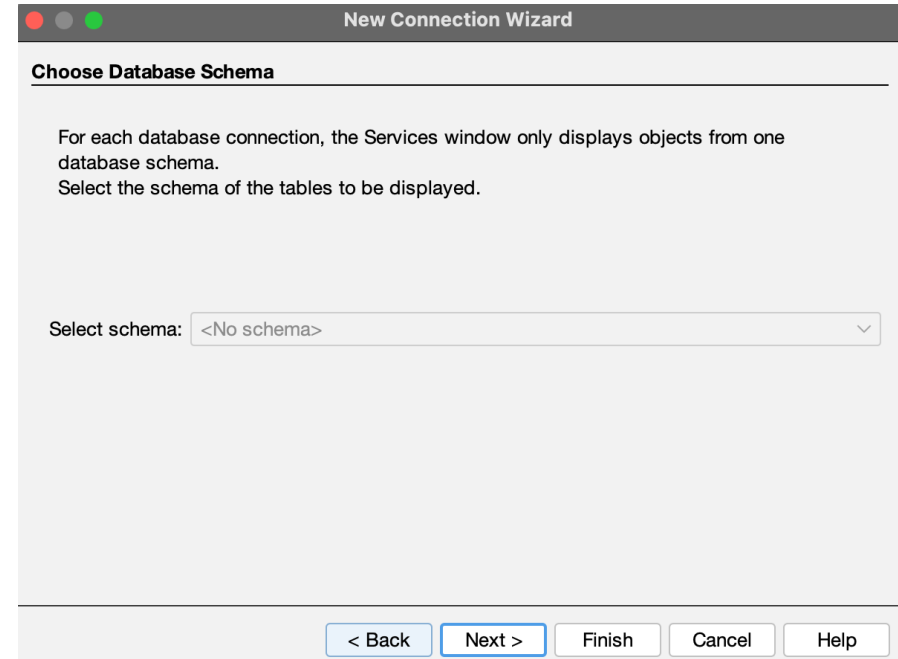
User Name:

Password:

☐ Remember password

JDBC URL:

< Back **Next >** Finish Cancel Help



New Connection Wizard

Choose Database Schema

For each database connection, the Services window only displays objects from one database schema.
Select the schema of the tables to be displayed.

Select schema:

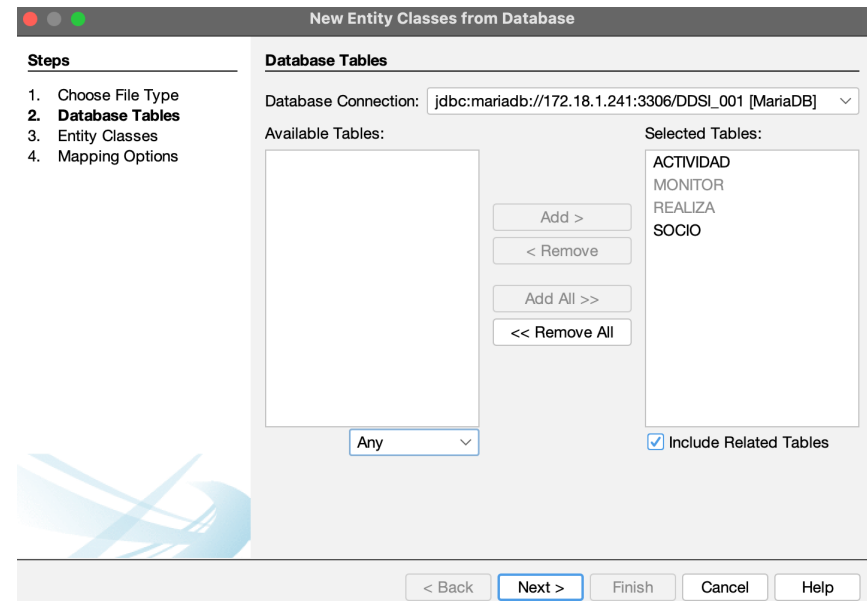
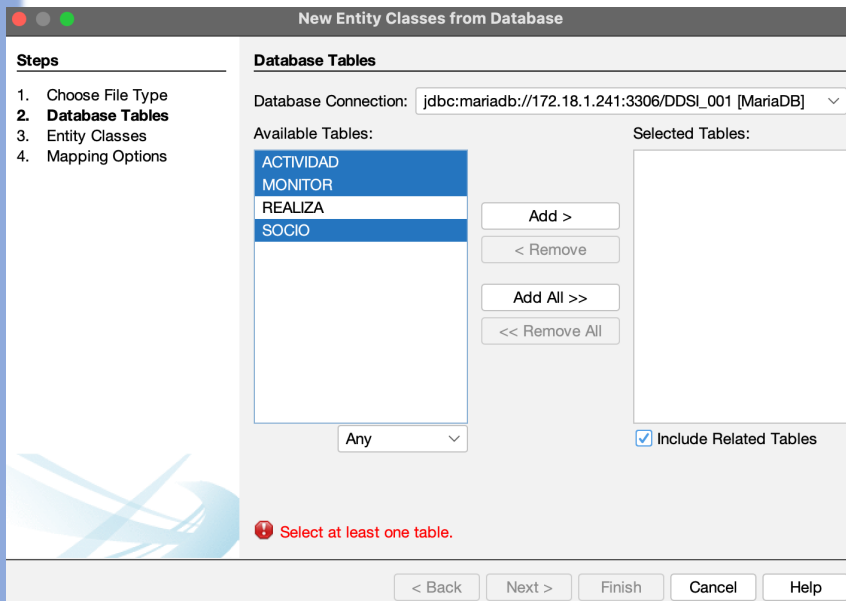
< Back **Next >** Finish Cancel Help

En el caso de MariaDB no hay que elegir "schema" puesto que tomará el del usuario/password seleccionado en el paso anterior

Creación de las clases derivadas de las tablas de la BD

IMPORTANTE

- En JPA/Hibernate **no se mapean las tablas "intermedias"** que surgen de las relaciones "muchos a muchos"



Creación de las clases derivadas de las tablas de la BD

Steps

1. Choose File Type
2. Database Tables
- 3. Entity Classes**
4. Mapping Options

Entity Classes

Specify the names and the location of the entity classes.

Class Names:

Database Table	Class Name	Generation Type
ACTIVIDAD	Actividad	New
MONITOR	Monitor	New
REALIZA	join table	New

Project: ProyectoDDSI_23_24

Location: Source Packages

Package: **Modelo**

☒ Generate Named Query Annotations for Persistent Fields

☐ Generate JAXB Annotations

☐ Generate MappedSuperclasses instead of Entities

☐ **Create Persistence Unit**

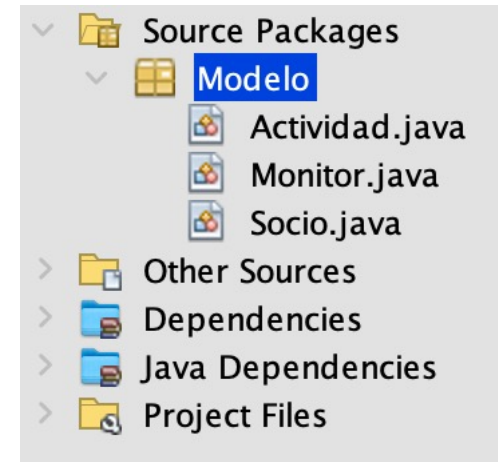
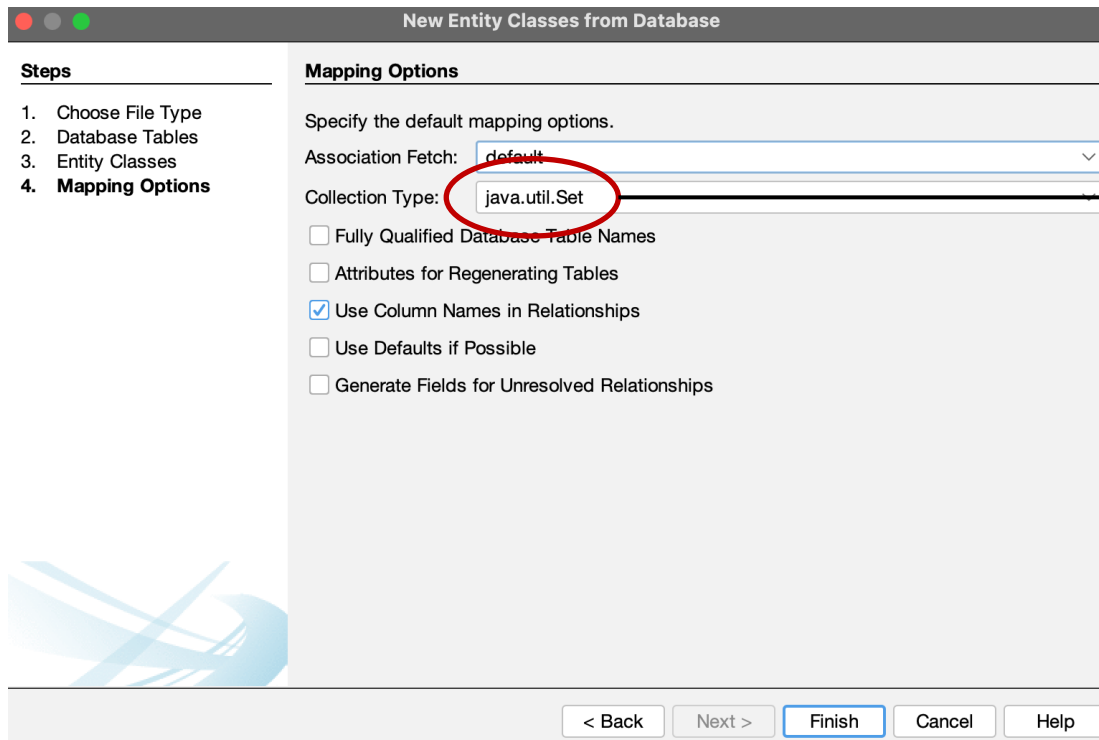
⚠ The project does not have a persistence unit. You need a persistence unit to persist entit...

< Back Next > Finish Cancel Help

Seleccionar la capa Modelo

Deseleccionar "Create Persistence Util"

Creación de las clases derivadas de las tablas de la BD



Seleccionar Set como estructura de datos para almacenar las colecciones de los atributos

Anotaciones

- **@Entity**. Especifica que la clase es persistente
- **@Table**. Especifica el nombre de la tabla de la base de datos asociada a la clase de entidad
- **@Colum**. Se utiliza para indicar los detalles del atributo al que se debe asignar un campo de una entidad. Incluye el nombre, tipo y cualquier restricción
- **@Id**. Indica que el atributo corresponde a la clave principal en la tabla
- **@Basic**. Se utiliza para indicar que un atributo es obligatorio
- **@NamedQuery**. Se utiliza para definir una consulta con nombre a la que se puede hacer referencia en el código. Las consultas con nombre se escriben en HQL
- **@OneToMany**. Indica una relación de uno a muchos entre dos entidades
- **@ManyToMany**. Especifica una relación de muchos a muchos entre dos entidades

■ Clase Monitor (1/2)

```
@Entity
@Table(name = "MONITOR")
@NamedQueries({
    @NamedQuery(name = "Monitor.findAll", query = "SELECT m FROM Monitor m"), ... })
```

```
public class Monitor implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @Column(name = "codMonitor")
    private String codMonitor;
    @Basic(optional = false)
    @Column(name = "nombre")
    private String nombre;
    @Basic(optional = false)
    @Column(name = "dni")
    private String dni;
    @Column(name = "telefono")
    private String telefono;
    @Column(name = "correo")
    private String correo;
    @Column(name = "fechaEntrada")
    private String fechaEntrada;
    @Column(name = "nick")
    private String nick;
    @OneToOne(mappedBy = "monitorResponsable")
    private Set<Actividad> actividadSet;
```

Es necesario inicializar esta estructura y, para mayor claridad, ponerle un nombre más significativo

```
@OneToOne(mappedBy = "monitorResponsable")
private Set<Actividad> actividadesResponsable = new HashSet<Actividad>();
```

■ Clase Monitor (2/2)

```
public Monitor() { }

public Monitor(String codMonitor) { this.codMonitor = codMonitor; }

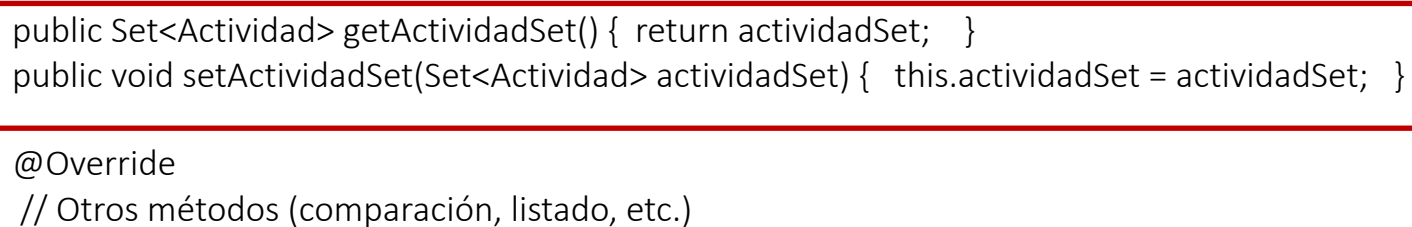
public Monitor(String codMonitor, String nombre, String dni, String fechaEntrada) {
    this.codMonitor = codMonitor;
    this.nombre = nombre;
    this.dni = dni;
    this.fechaEntrada = fechaEntrada;
}

public String getCodMonitor() { return codMonitor; }
public void setCodMonitor(String codMonitor) { this.codMonitor = codMonitor; }

public String getNombre() { return nombre; }
public void setNombre(String nombre) { this.nombre = nombre; }

...
public Set<Actividad> getActividadSet() { return actividadSet; }
public void setActividadSet(Set<Actividad> actividadSet) { this.actividadSet = actividadSet; }

@Override
// Otros métodos (comparación, listado, etc.)
```



```
public Set<Actividad> getActividadesResponsable() { return actividadesResponsable; }
public void setActividadesResponsable(Set<Actividad> actividadesResponsable) { this.actividadesResponsable = actividadesResponsable; }
```

- Se puede comprobar que la herramienta ha generado, de forma automática, 3 constructores para la clase **Monitor**:

```
public Monitor ()  
public Monitor (String codmonitor)  
public Monitor (String codmonitor, String nombre, String dni, String fechaentrada)
```

- Sin embargo, no ha generado el constructor con todos los atributos (ya que algunos no son obligatorios), por lo que tendremos que añadirlo manualmente:

```
public Monitor (String codmonitor, String nombre, String dni, String telefono, String correo,  
String fechaentrada, String nick)
```

■ Clase Actividad (1/2)

@Entity

@Table(name = "ACTIVIDAD")

@NamedQueries({

 @NamedQuery(name = "Actividad.findAll", query = "SELECT a FROM Actividad a"), ... })

public class Actividad implements Serializable {

 private static final long serialVersionUID = 1L;

 @Id

 @Basic(optional = false)

 @Column(name = "IdActividad")

 private String idActividad;

 @Basic(optional = false)

 @Column(name = "nombre")

 private String nombre;

 @Column(name = "descripcion")

 private String descripcion

 @Column(name = "precioBaseMes")

 private int precioBaseMes;

 @JoinTable (name = "REALIZA",

 joinColumns = {@JoinColumn (name = "idActividad", referencedColumnName = "idActividad ")},

 inverseJoinColumns = {@JoinColumn (name = "numeroSocio", referencedColumnName = "numeroSocio")})

 @ManyToMany

 private Set<Socio> socioSet;

 @JoinColumn (name = "monitorResponsable", referencedColumnName = "codMonitor")

 @ManyToOne

 private Monitor monitorResponsable;

Es necesario inicializar esta estructura y, para mayor claridad, ponerle un nombre más significativo

@ManyToMany

private Set<Socio> socios = new HashSet<Socio>();



■ Clase Actividad (2/2)

```
public Actividad() { }

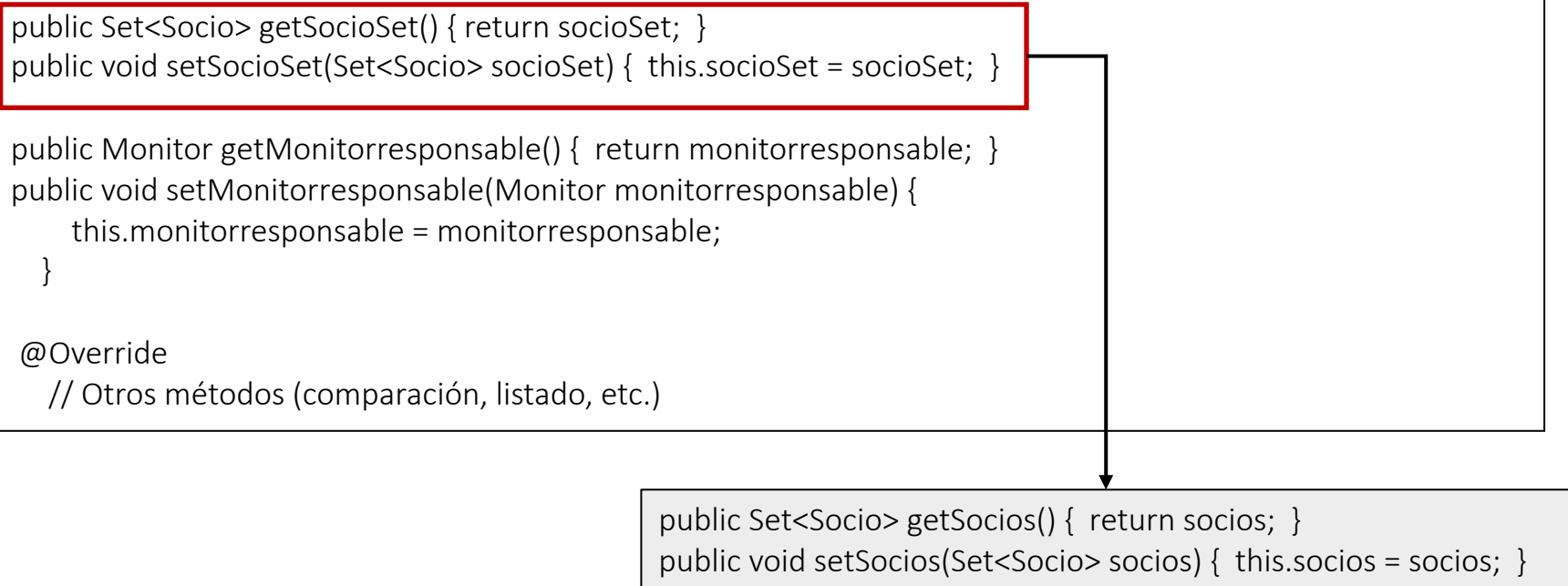
public Actividad(String idActividad) { this.idactividad = idActividad; }

public Actividad(String idActividad, String nombre, int precioBaseMes) {
    this.idActividad = idActividad;
    this.nombre = nombre;
    this.precioBaseMes = precioBaseMes;
}

public String getIdActividad() { return idActividad; }
public void setIdActividad(String idActividad) { this.idActividad = idActividad; }
...
public Set<Socio> getSocioSet() { return socioSet; }
public void setSocioSet(Set<Socio> socioSet) { this.socioSet = socioSet; }

public Monitor getMonitorresponsable() { return monitorresponsable; }
public void setMonitorresponsable(Monitor monitorresponsable) {
    this.monitorresponsable = monitorresponsable;
}

@Override
// Otros métodos (comparación, listado, etc.)
```



```
public Set<Socio> getSocios() { return socios; }
public void setSocios(Set<Socio> socios) { this.socios = socios; }
```

■ Clase Socio (1/2)

@Entity

@Table(name = "SOCIO")

@NamedQueries({

@NamedQuery(name = "Socio.findAll", query = "SELECT s FROM Socio s"), ... })

public class Socio implements Serializable {

private static final long serialVersionUID = 1L;

@Id

@Basic(optional = false)

@Column(name = "numeroSocio")

private String numeroSocio;

@Basic(optional = false)

@Column(name = "nombre")

private String nombre;

@Basic(optional = false)

@Column(name = "dni")

private String dni;

@Column(name = "fechaNacimiento")

private String fechaNacimiento;

@Column(name = "telefono")

private String telefono;

@Column(name = "correo")

private String correo;

@Column(name = "fechaEntrada")

private String fechaEntrada;

@Basic(optional = false)

@Column(name = "categoria")

private String categoria;

@ManyToMany(mappedBy = "socioSet")

private Set<Actividad> actividadSet;

Es necesario inicializar esta estructura y, para mayor claridad, ponerle un nombre más significativo

@ManyToMany(mappedBy = "socios")

private Set<Actividad> actividades = new HashSet<Actividad>();

■ Clase Socio (2/2)

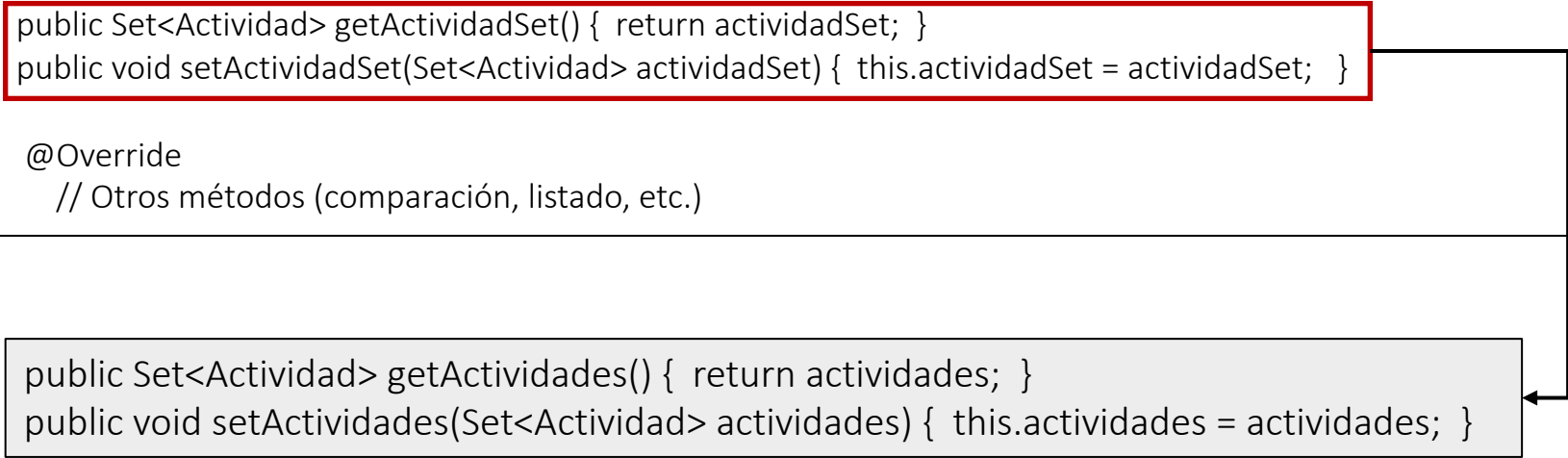
```
public Socio() { }

public Socio(String numeroSocio) { this.numeroSocio = numeroSocio; }

public Socio(String numeroSocio, String nombre, String dni, String fechaEntrada, Character categoria) {
    this.numeroSocio = numeroSocio;
    this.nombre = nombre;
    this.dni = dni;
    this.fechaEntrada = fechaEntrada;
    this.categoria = categoria;
}

public String getNumeroSocio() { return numeroSocio; }
public void setNumeroSocio(String numeroSocio) { this.numeroSocio = numeroSocio; }
...
public Set<Actividad> getActividadSet() { return actividadSet; }
public void setActividadSet(Set<Actividad> actividadSet) { this.actividadSet = actividadSet; }

@Override
// Otros métodos (comparación, listado, etc.)
```



```
public Set<Actividad> getActividades() { return actividades; }
public void setActividades(Set<Actividad> actividades) { this.actividades = actividades; }
```

Conexión a los servidores de BD con Hibernate

Controlador Login

- Tendrá un atributo de clase **SessionFactory**, que será el encargado de establecer la conexión con la base de datos a través de Hibernate
- Reprogramamos la función **conectarBD()**. En este caso no es necesario pedir la información de la conexión puesto que esos datos están almacenados en los ficheros **hibernate.cfg.xml** correspondientes

```
private void conectarBD() {  
    try {  
        if (condición de elección de Oracle) {  
            sessionFactory = HibernateUtilOracle.getSessionFactory();  
        } else if (condición de elección de MariaDB) {  
            sessionFactory = HibernateUtilMariaDB.getSessionFactory();  
        }  
        vMensaje.mensajeConsola("Conexión Correcta con Hibernate");  
  
    } catch (ExceptionInInitializerError e) {  
        Throwable cause = e.getCause();  
        System.out.println("Error en la conexión. Revise el fichero .cfg.xml: " + cause.getMessage());  
    }  
}
```

Conexión a los servidores de BD con Hibernate

Controlador Login

- Si la conexión es correcta, se crea un objeto de tipo Controlador y se le pasa por parámetro la conexión (que ahora es un objeto de tipo [SessionFactory](#))

```
public ControladorLogin() {  
    conectarBD();  
  
    ControladorPrincipal controladorP = new ControladorPrincipal(sessionFactory);  
}
```

Conexión a los servidores de BD con Hibernate

Controlador Principal

■ Esquema general de funcionamiento para realizar una operación a la BD

```
Session sesion = sessionFactory.openSession();
tr = sesion.beginTransaction();

try {
    ArrayList<Socio> ISocios = socioDAO.listaSociosHQL(sesion);
    vSocios.muestraSocios(ISocios);
    tr.commit();
} catch (Exception e) {
    tr.rollback();
    vMensajes.mensajeConsola("Error en la petición de socios",
        e.getMessage());
} finally {
    if (sesion != null && sesion.isOpen()) {
        sesion.close();
    }
}
```

Se abre una sesión (objeto de tipo Session) con la conexión recibida desde el Controlador Login

Se comienza una transacción. El controlador tendrá un atributo de tipo **Transaction**. En este ejemplo es "tr"

Se realiza la operación CRUD a través de un método DAO, pasándole como parámetro la sesión abierta

Se realiza el tratamiento con el resultado. En este caso, se envía a la vista vSocios

Si la operación tiene éxito, se confirma (commit)

Si la operación falla, se deshace (rollback)

Finalmente, en cualquier caso, se cierra la sesión

Cuando se programan aplicaciones que gestionan una base de datos, las operaciones deben realizarse en el entorno de una transacción para garantizar la consistencia de la información. Aunque para las operaciones de consultas (SELECT) no es necesario, se muestra la estructura general que tendrán las llamadas a las operaciones CRUD desde un controlador

Operaciones CRUD con Hibernate

Consultas

- *Hibernate* permite realizar consultas mediante su propio lenguaje denominado **HQL** (*Hibernate Query Language*) o realizarlas directamente en lenguaje SQL ("**nativas**")
- Existen diversas formas de ejecutar consultas. Los métodos más habituales y que usaremos en estas prácticas son **createQuery()**, **createNamedQuery()** y **createNativeQuery()**
- Todos los métodos devuelven un objeto de tipo **Query**

Operaciones CRUD con Hibernate

Consultas

```
public ArrayList<Monitor> listaMonitoresConHQL(Session sesion) throws Exception {  
    Query consulta = sesion.createQuery("SELECT m FROM Monitor m", Monitor.class);  
    ArrayList<Monitor> monitores = (ArrayList<Monitor>) consulta.list();  
  
    return monitores;  
}
```

Consulta HQL para recuperar toda la información de un "monitor". El segundo parámetro indica que se está mapeando toda la tabla **MONITOR** en la clase **Monitor**.

Se utiliza el método **list()** para obtener una lista de resultados. Podemos usar una estructura específica para almacenar la lista de los resultados. En este ejemplo, se ha usado un **ArrayList** de objetos de tipo **Monitor**

Operaciones CRUD con Hibernate

Consultas

La misma consulta con HQL sin usar la cláusula SELECT

```
Query consulta = sesion.createQuery ("FROM Monitor m", Monitor.class);
```

La misma consulta con HQL usando una consulta "nombrada"

```
@NamedQuery(name = "Monitor.findAll", query = "SELECT m FROM Monitor m")
```

```
Query consulta = sesion.createNamedQuery ("Monitor.findAll", Monitor.class);
```

La misma consulta con SQL nativo

```
Query consulta = sesion.createNativeQuery ("SELECT * FROM MONITOR M", Monitor.class);
```

¡¡ Cuidado !! MariaDB es sensible a mayúsculas y minúsculas. Este nombre debe coincidir con el de la etiqueta **@Table** del POJO correspondiente

La misma consulta con SQL nativo usando una consulta "nombrada"

```
@NamedNativeQueries({  
    @NamedNativeQuery(name = "Monitor.TodosNativos", query = "SELECT * FROM MONITOR M",  
        resultClass = Monitor.class) })
```

```
Query consulta = sesion.createNamedQuery ("Monitor.TodosNativos", Monitor.class);
```

Operaciones CRUD con Hibernate

Consultas

- Por supuesto, también es posible realizar consultas que devuelvan campos específicos

Consulta que devuelve dos campos de la tabla **MONITOR**

```
public ArrayList<Object[]> listaNombreDNIMonitores(Session sesion) throws Exception {  
    Query consulta = sesion.createQuery("SELECT m.nombre, m.dni FROM Monitor m");  
    ArrayList<Object[]> monitores = (ArrayList<Object[]>) consulta.list();  
  
    return monitores;  
}
```

En este caso, la versión en SQL nativo tiene la misma sintaxis

```
Query consulta = sesion.createNativeQuery("SELECT m.nombre, m.dni FROM MONITOR m");
```

La consulta devuelve un array de objetos "genérico". Por tanto, para recorrerlo, visualizarlo, etc., habrá que usar un código adecuado

```
public void muestraMonitores_Nombre_DNI(ArrayList<Object[]> lMonitores) {  
    System.out.println("Nombre" + "\t" + "DNI" + "\t");  
    for (Object[] m : lMonitores) {  
        System.out.println(m[0] + "\t" + m[1]);  
    }  
}
```

Ejemplo de un método que muestra el contenido de un ArrayList de objetos con 2 campos

Operaciones CRUD con Hibernate

Consultas

Consulta que devuelve un único campo de la tabla **MONITOR**

```
public ArrayList<String> listaCorreoMonitores(Session sesion) throws Exception {  
    Query consulta = sesion.createNativeQuery("SELECT m.correo FROM MONITOR m");  
    ArrayList<String> correoMonitores = (ArrayList<String>) consulta.list();  
  
    return correoMonitores;  
}
```

Consulta que devuelve un valor único

```
public String ultimoMonitor(Session sesion) throws Exception {  
    Query consulta = sesion.createQuery("SELECT MAX(m.codMonitor) FROM Monitor m");  
    String codUltimoMonitor = consulta.getSingleResult().toString();  
  
    return codUltimoMonitor;  
}
```

Operaciones CRUD con Hibernate

Consultas

- Y consultas parametrizadas

Consulta HQL parametrizada

```
public Monitor devuelveMonitorDNI(Session sesion, String pDNI) throws Exception {  
    Query consulta = sesion.createQuery("SELECT m FROM Monitor m WHERE m.dni = :dni",  
        Monitor.class);  
    consulta.setParameter("dni", pDNI);  
  
    Monitor monitor = (Monitor) consulta.getSingleResult();  
    return monitor;  
}
```

Una forma más "elegante"

```
Query consulta = sesion.createQuery("SELECT m FROM Monitor m WHERE m.dni = :dni",  
    Monitor.class).setParameter("dni", pDNI)
```

Operaciones CRUD con Hibernate

Consultas

- Y todavía más... consultas con JOIN

SQL nativo

```
public ArrayList<Object[]> devuelveResponsables(Session sesion) throws Exception {  
    Query consulta = sesion.createNativeQuery("SELECT a.nombre as nombreActividad,  
        m.nombre as nombreMonitor FROM ACTIVIDAD a INNER JOIN MONITOR m ON  
        a.monitorResponsable = m.codMonitor");  
  
    ArrayList<Object[]> responsables = (ArrayList<Object[]>) consulta.list();  
  
    return responsables;  
}
```

HQL

```
Query consulta = sesion.createQuery("SELECT a.nombre as nombreActividad, m.nombre as  
    nombreMonitor FROM Actividad a JOIN a.monitorResponsable m");
```