Mapeo Objeto Relacional

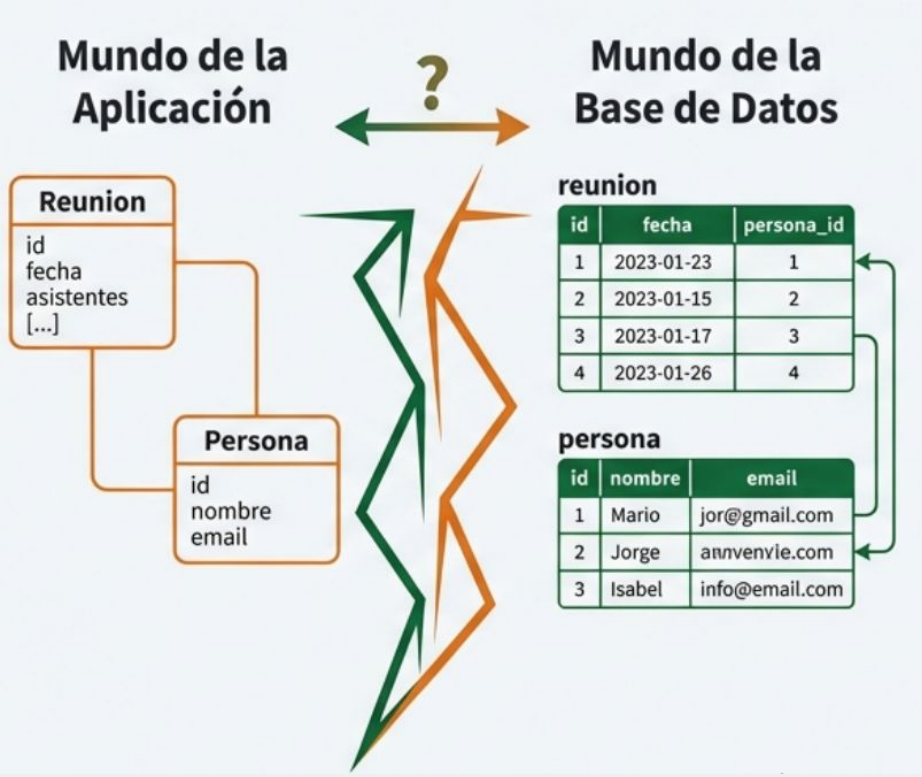
Acceso a Datos

Objetivos

- Comprender el propósito y los fundamentos del Mapeo Objeto - Relacional (ORM)
- Mapeo a través de Hibernate: correspondencia hbm y relaciones
- Crear POJO (clases Java) y su correspondencia para ORM Hibernate.
- Analizar el ciclo de vida de objetos persistentes

Object Relational Mapping (ORM)

En nuestras aplicaciones, trabajamos con un mundo de objetos: clases, herencia, colecciones y relaciones complejas. Sin embargo, las bases de datos relacionales almacenan datos en un formato de tablas con campos con tipos primitivos y relaciones entre tablas (FK)



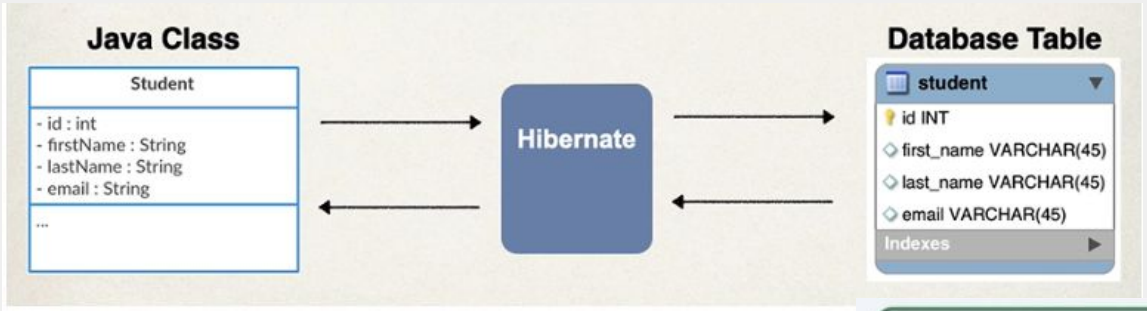
Object Relational Mapping (ORM)

El reto: ¿Cómo traducimos de manera eficiente y mantenible entre estos dos paradigmas?

El **Mapeo Objeto-Relacional (ORM)** es la técnica que nos permite superar este desafío. Actúa como una capa de traducción automática entre el modelo de objetos de su aplicación y la base de datos relacional.

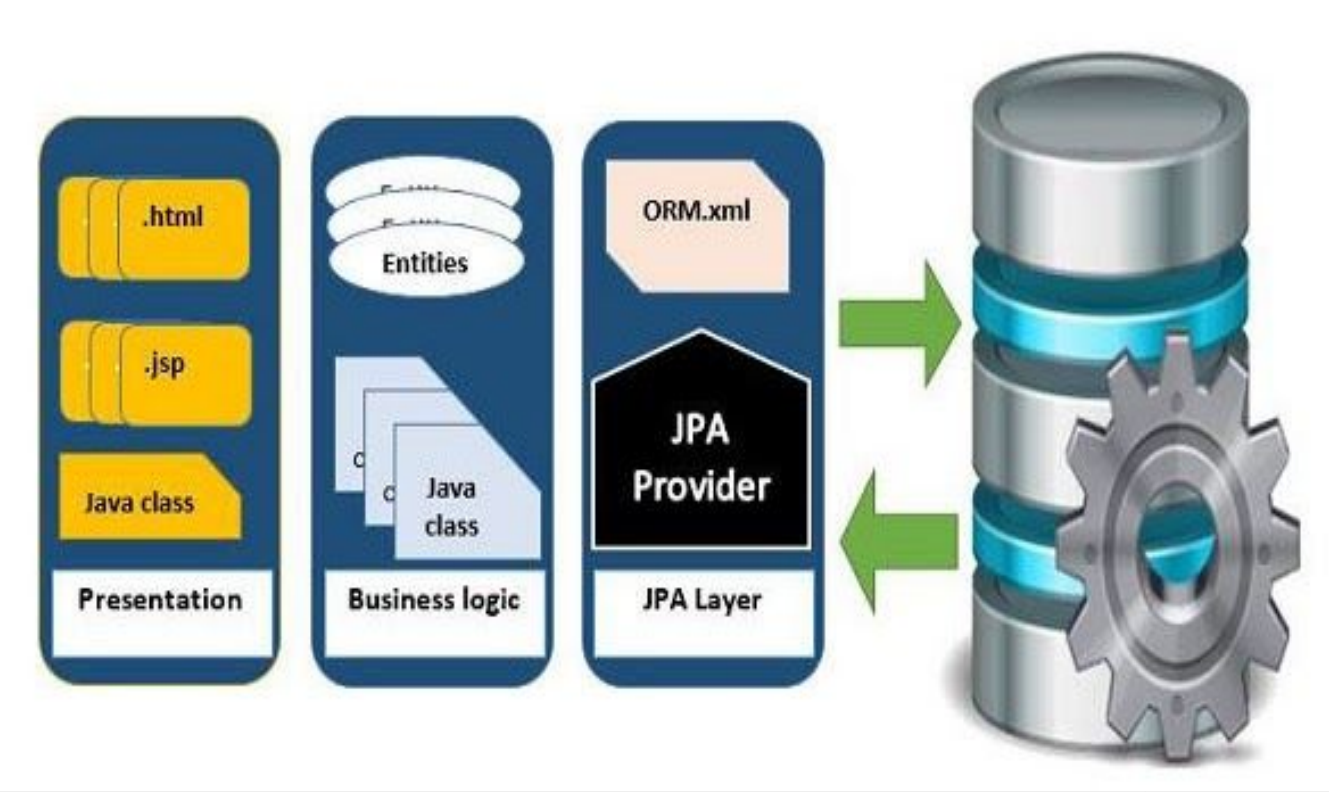
Object Relational Mapping (ORM)

ORM nos ayudará con esta correspondencia objeto-relacional. Consisten en las técnicas y herramientas que hacen posible la persistencia de **objetos** en bases de datos **relacionales (tablas)**, mediante el mapeo o la correspondencia de los objetos y las tablas.



"En lugar de escribir consultas SQL para interactuar con la base de datos, se utilizan objetos de programación para guardar, actualizar y recuperar datos."

Object Relational Mapping (ORM)



En la práctica tendremos una base de datos física que se mapeara con una base de datos virtual

Herramientas ORM

- Permiten crear una capa intermedia para el acceso y persistencia de los datos.
- Estas herramientas aportan un lenguaje de consultas orientadas a objetos propio e **independiente** de la BD
- Algunas herramientas ORM:
  - Para Java y .Net: Hibernate y Nhibernate respectivamente
  - Para PHP: Doctrine, Propel, ADOdb
  - Para Visual, .Net y C#: LINQ

ORM: Hibernate

Hibernate es la tecnología Java más extendida para ORM.

Ventajas:

- **Productividad**. Permite centrarnos en la lógica de negocio sin tener que centrarnos en transformaciones entre objetos y base de datos
- **Mantenibilidad**. Se reduce el número de líneas, reduciendo el número de errores.
- **Rendimiento**. A pesar de añadir una capa de abstracción, Hibernate aporta optimizaciones como puede ser el mecanismo de caché.
- **Independencia**. Nos aisla del Sistema Gestor de la Base de Datos.

ORM: Hibernate

Además de un ORM, Hibernate ofrece diferentes módulos como:

- **EntityManager** Implementación del estándar JPA
- **Envers**. Auditoría de los cambios que se van realizando. Control de versiones.
- **OGM** Para Bases de Datos NO SQL Mapeador Objeto- GRID

ORM: Hibernate vs JDBC

Ventajas	Desventajas
• Mayor rendimiento • Funcionalidades concretas	• Mayor tiempo desarrollo. • Mantenimiento costoso. • Errores durante el desarrollo

• Desarrollo más veloz. • Uso de entidades en lugar de *queries*. • Eliminación errores ejecución.	• Curva aprendizaje más pronunciada. • Posibilidad de menor rendimiento, si bien existe posibilidad de *caché*.

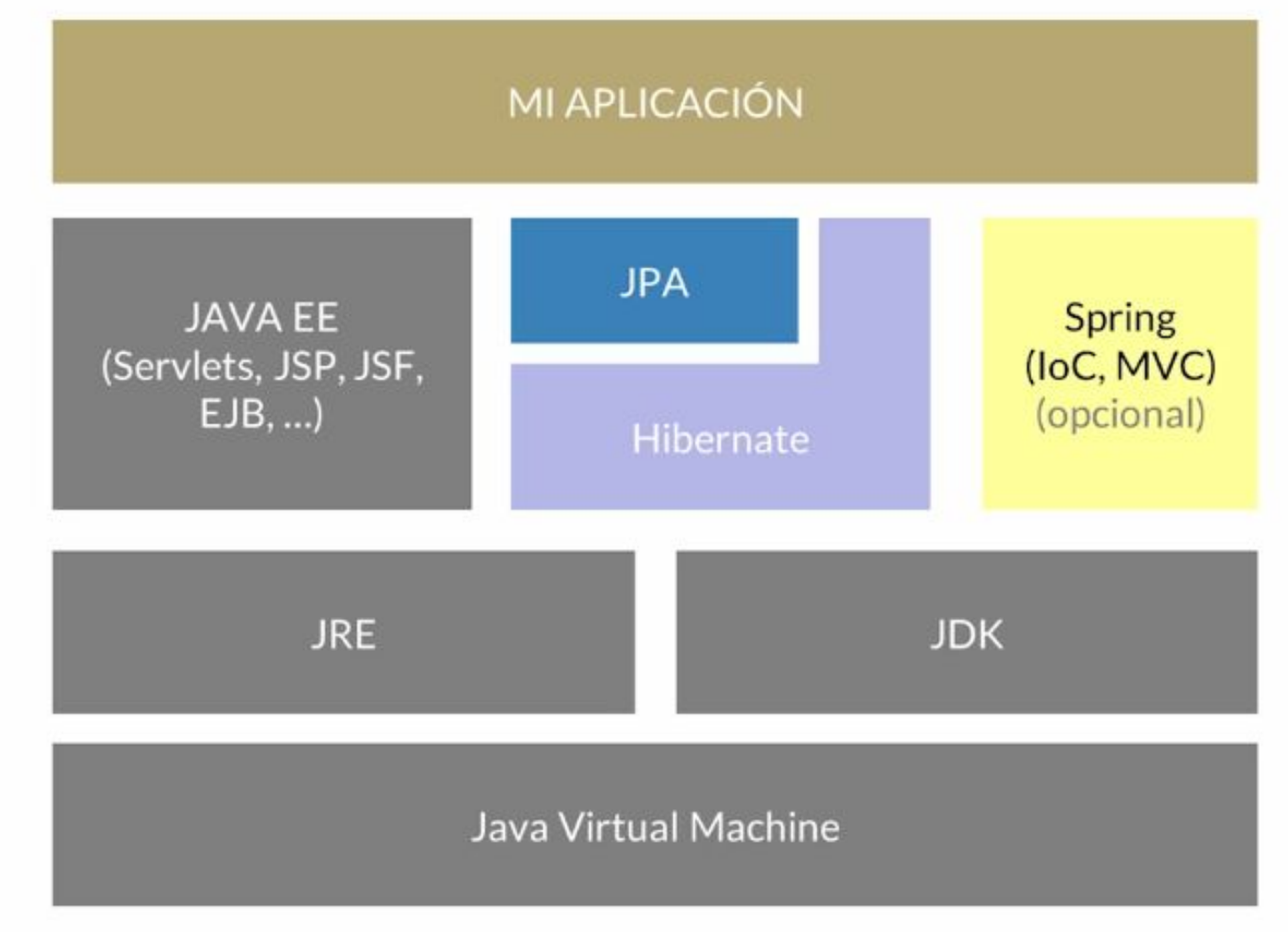
ORM: Hibernate vs JPA

JPA es la especificación para ORM. Este estándar consiste en:

- Especificación de cómo los objetos se relacionan con la BBDD
- API sencilla para realizar CRUD
- Lenguaje de consultas y API para realizar consultas sobre datos
- Elementos de optimización.

Hibernate es una implementación del estándar JPA

ORM: Arquitectura Tipo



ORM: Hibernate

- Para hacer la persistencia de nuestros objetos Java a la base de datos hay que indicar cómo se debe realizar dicha persistencia. Para ello hay dos métodos:
  - Ficheros XML
  - Anotaciones en el código: **JPA**
- **JPA** es una especificación mientras que Hibernate es la implementación de esta especificación.
- Con Hibernate no utilizaremos SQL habitualmente, sino que el propio motor de Hibernate construirá las consultas.
- Usaremos el lenguaje **HQL** (Hibernate Query Language) para acceder a los datos. Mientras SQL opera sobre filas y columnas de tablas, HQL opera sobre conjuntos de objetos persistentes **JPQL**.

ORM: Arquitectura Hibernate

Entre los objetos debemos diferenciar:

- Objetos **transitorios**
- Objetos **persistentes**. Los objetos del modelo que tienen una correspondencia mediante Hibernate.

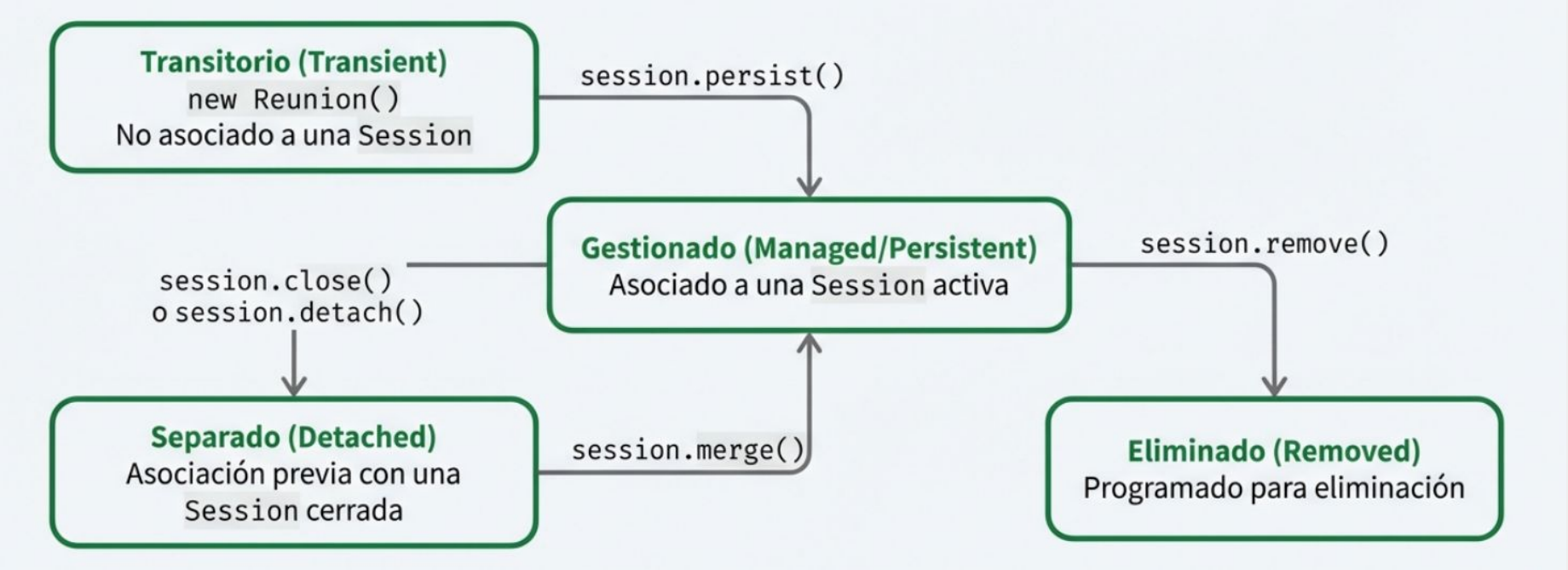
La aplicación creará objetos transitorios y sólo algunos serán persistidos en la base de datos.

Los objetos se persisten en una **Session**. Una sesión se construye sobre la conexión de la base de datos.

El gestor de entidades (EntityManager) asociado junto con la sesión constituye el contexto de la persistencia.

ORM: Arquitectura Hibernate

Un objeto en el mundo de Hibernate no es simplemente 'guardado'. Pasa por distintos estados que definen su relación con el contexto de persistencia (la Session).

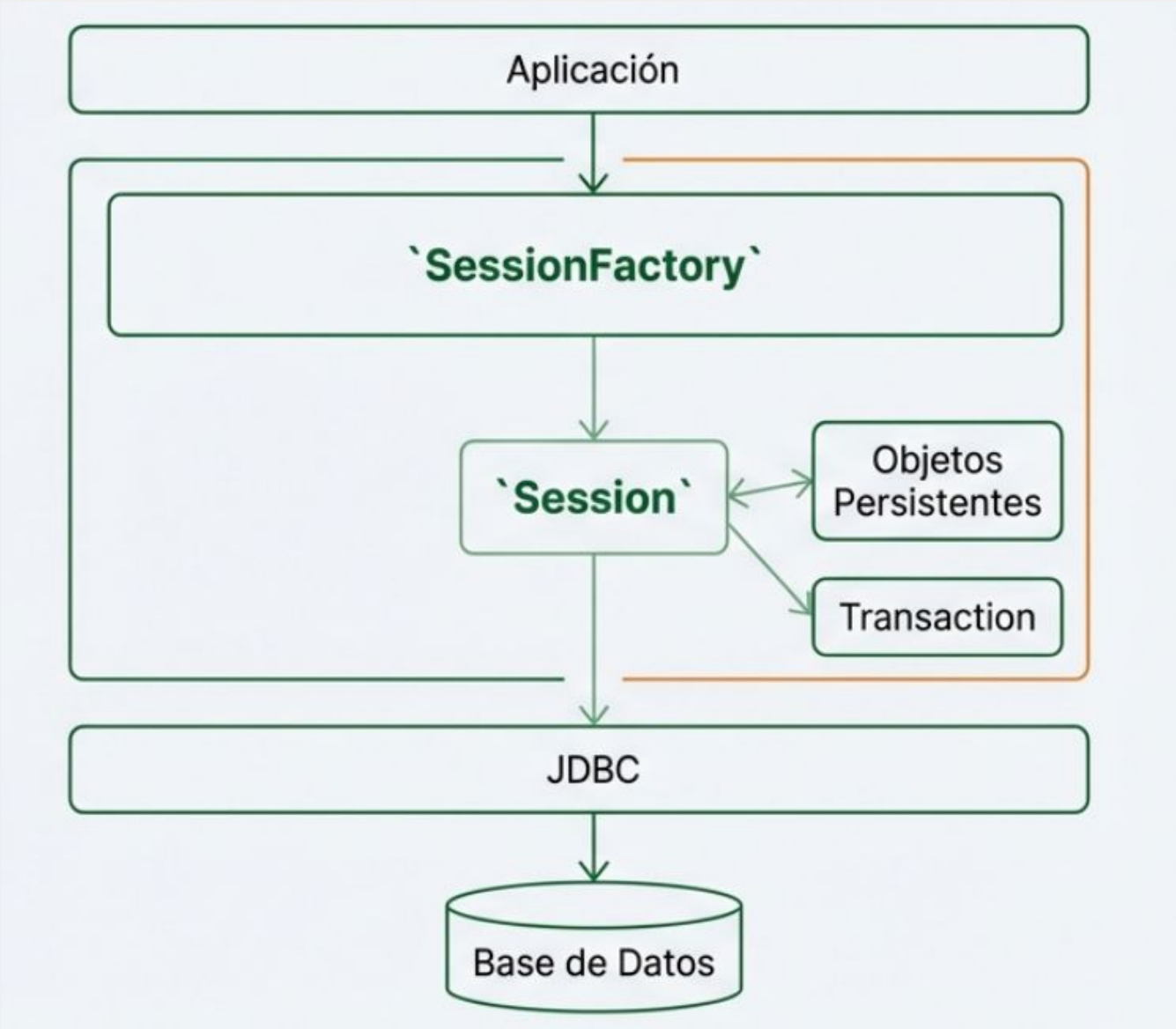


ORM: Arquitectura Hibernate

Estados de la persistencia

1. **Transitorio** (transient) El objeto no está asociado con ningún contexto de persistencia. No tiene porqué tener un id único
2. **Gestionado** (managed) o persistente (persistent). El objeto está asociado a un contexto de persistencia. Cualquier cambio que se realice en el objeto, se reflejará en la base de datos.
3. **Separado** (detached). El objeto tiene un id asociado pero no está asociado a un contexto, ya sea porque el contexto se ha cerrado o porque se desvinculó.
4. **Eliminado** (removed) El objeto tiene id asociado y está asociado a un contexto pero está pendiente de ser eliminado en la BBDD.

ORM: Hibernate Componentes Clave



ORM: Arquitectura Hibernate

La **Session** es el principal interfaz de Hibernate para realizar operaciones CRUD con la base de datos.

- Proporciona métodos para almacenar, actualizar, eliminar y recuperar objetos persistentes.
- Cada sesión es una conexión ligera y corta con la BBDD.
- No es segura para múltiples hilos (thread-safe), por lo que se recomienda abrir una nueva sesión por cada **transacción** o unidad de trabajo.
- Abriremos una sesión a partir de la clase **SessionFactory**.

ORM: Arquitectura Hibernate

La **SessionFactory** es un objeto pesado y es responsable de crear instancias de Session.

Es un recurso costoso de crear, por lo que se crea solo una vez por aplicación (en general, se utiliza un **Singleton**).

La configuración de la SessionFactory contiene toda la información de conexión a la base de datos, configuraciones de Hibernate y mapeos de las entidades.

SessionFactory se crea a partir de **Metadata**

ORM: Arquitectura Hibernate

Metadata es una clase en Hibernate que encapsula toda la información de configuración y mapeo que Hibernate necesita para inicializar la **SessionFactory**.

Hibernate lee y agrupa en Metadata toda la información que necesita de las configuraciones de mapeo (anotaciones o archivos XML) y de los archivos de configuración (hibernate.cfg.xml, por ejemplo).

Para obtener una instancia de Metadata, se usa la clase **MetadataSources**, que toma la configuración de **StandardServiceRegistry** y genera los metadatos.

ORM: Arquitectura Hibernate

StandardServiceRegistry es responsable de gestionar todos los servicios que Hibernate necesita en su ejecución, como la conexión a la base de datos, el manejo de transacciones, la configuración de caché, entre otros.

Es el punto de entrada inicial para configurar Hibernate, y permite agrupar todas las configuraciones y propiedades en un solo lugar.

StandardServiceRegistryBuilder es una clase de apoyo que permite construir y configurar un StandardServiceRegistry de manera programática, y personalizar los servicios mediante configuraciones en código o a través de archivos de propiedades (como **hibernate.properties o hibernate.cfg.xml**).

ORM: Arquitectura Hibernate

Las operaciones de recuperación o almacenamiento de los objetos persistentes se agrupan en transacciones (**Transaction**) que están siempre asociadas a una sesión.

Permite agrupar operaciones en transacciones, facilitando la gestión de transacciones **ACID** (Atomicidad, Consistencia, Aislamiento y Durabilidad).

Se puede gestionar de manera **programática o declarativa**, y es compatible con transacciones JTA (Java Transaction API).

ORM: Arquitectura Hibernate

Utilizará sentencias **HQL** mediante la clase **Query**

Hibernate interactúa con la BBDD mediante **JDBC.** Internamente puede utilizar APi de Java como **JNDI** (java Naming and Directory Interface) para fuente de datos (**Datasource**) y **JTA** (Java Transformation Arquitecture)

ORM: Hibernate primeros pasos: base datos

1. Arrancar contenedor con mysql en Docker
2. Configuramos conexión en MySQLWorkbench.
3. Configuraremos las dependencias en el pom.xml
4. Añadir la clase PruebaConexion para probar nuestras dependencias:

```
public class PruebaConexion {
public static void main(String[] args) throws Exception {
        Connection c = DriverManager.getConnection(
            "jdbc:mysql://localhost:3307/accesoDatosDAM",
            "usuario2",
            "usuario"
        );
        System.out.println("Conectado OK");
        c.close();
    }
}
```

ORM: Hibernate primeros pasos: dependencias

Añadir **dependencias** para el driver de la BBDD y para la propia librería de Hibernate. En nuestro caso agregaremos:

```
<dependency> <!-- Hibernate ORM -->
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.4.4.Final</version>
</dependency>
<dependency> <!-- MySQL Connector -->
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <version>8.4.0</version>
</dependency>
<dependency> <!-- JPA API (Jakarta) -->
    <groupId>jakarta.persistence</groupId>
    <artifactId>jakarta.persistence-api</artifactId>
    <version>3.1.0</version>
</dependency>
```

ORM: Hibernate primeros pasos: dependencias

Además agregaremos la dependencia de **JUnit 5** para poder hacer pruebas unitarias

```
<!-- JUnit -->
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.10.1</version>
    <scope>test</scope>
</dependency>
```

ORM: Hibernate primeros pasos: base datos

3. Creamos también un usuario (con la clave igual que el nombre del usuario creado)

   Para crearlo y darle a este usuario todos los privilegios sobre la BD en MySQLWorkbench:

   -- Conectar como root

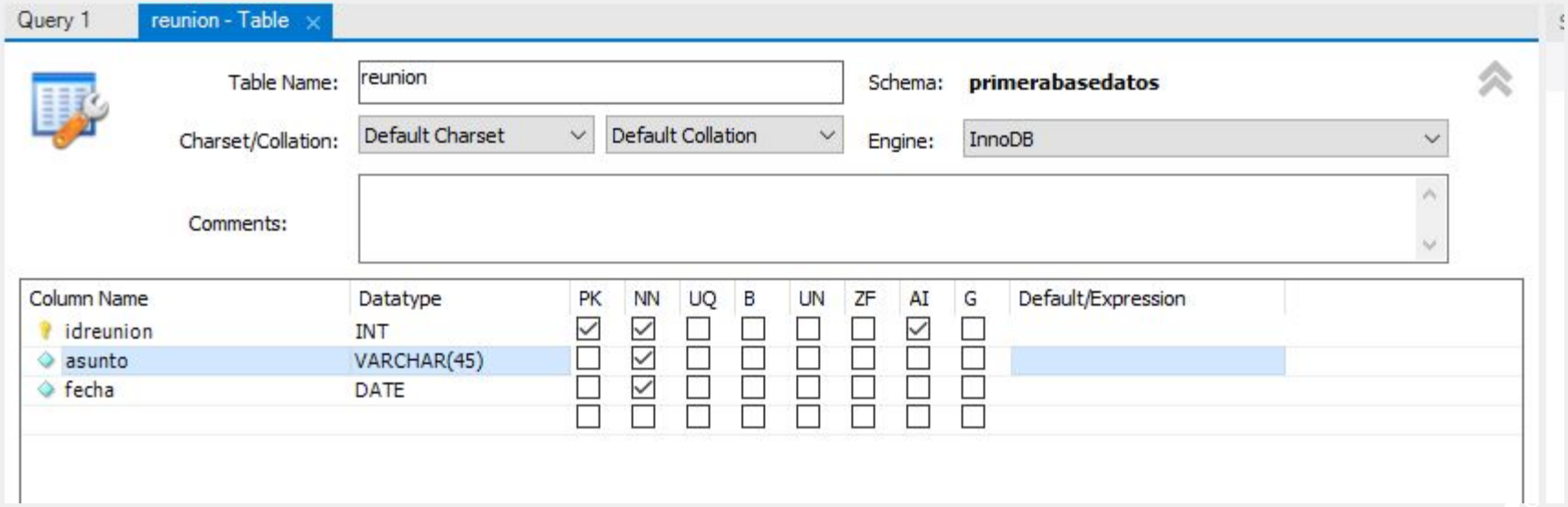
   CREATE USER 'usuario2'@'localhost' IDENTIFIED BY 'usuario';

   GRANT ALL PRIVILEGES ON accesoDatosDAM*.* TO 'usuario2'@'localhost';*

   FLUSH PRIVILEGES;

ORM: Hibernate primeros pasos: base datos

5. Creamos una primera tabla llamada reunion:
    - idReunion: INT, Primary Key y Not NULL y, además, marcaremos AI (AUTO_INCREMENTO)
    - fecha: DATETIME, Not NULL
    - asunto: VARCHAR de 45, Not NULL

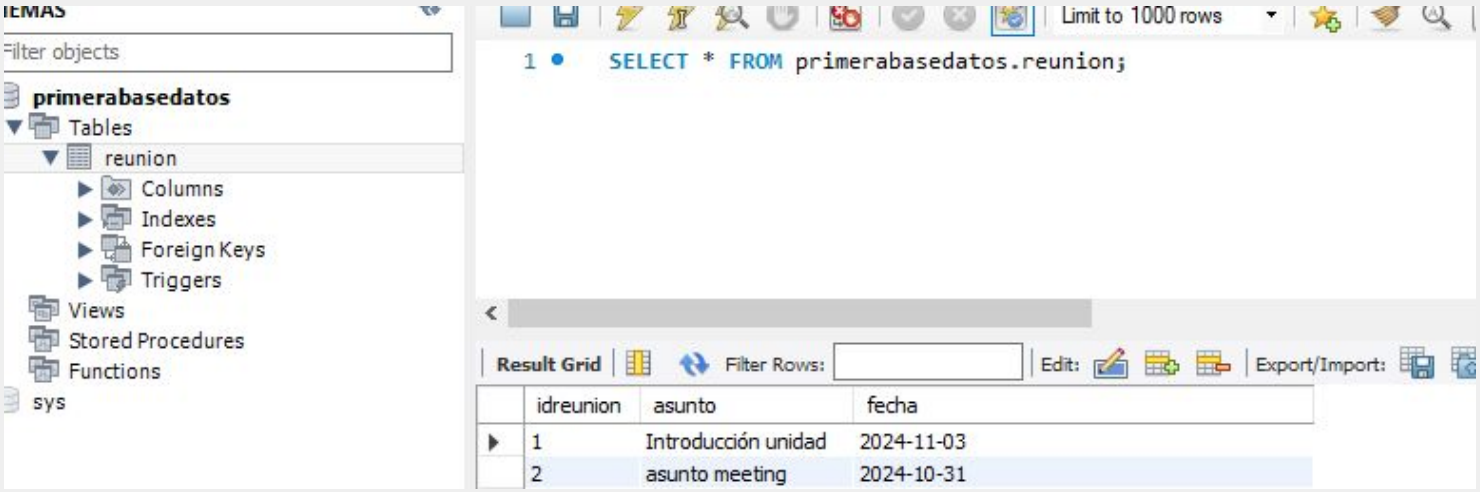


ORM: Hibernate primeros pasos: base datos

Creamos una primera tabla llamada reunion:

```
CREATE TABLE `primerabasedatos`.`reunion`
(  `idreunion` INT NOT NULL AUTO_INCREMENT,
    `asunto` VARCHAR(45) NOT NULL,
    `fecha` DATE NOT NULL,
    PRIMARY KEY (`idreunion`));
```

6. Insertamos dos reuniones en nuestra tabla

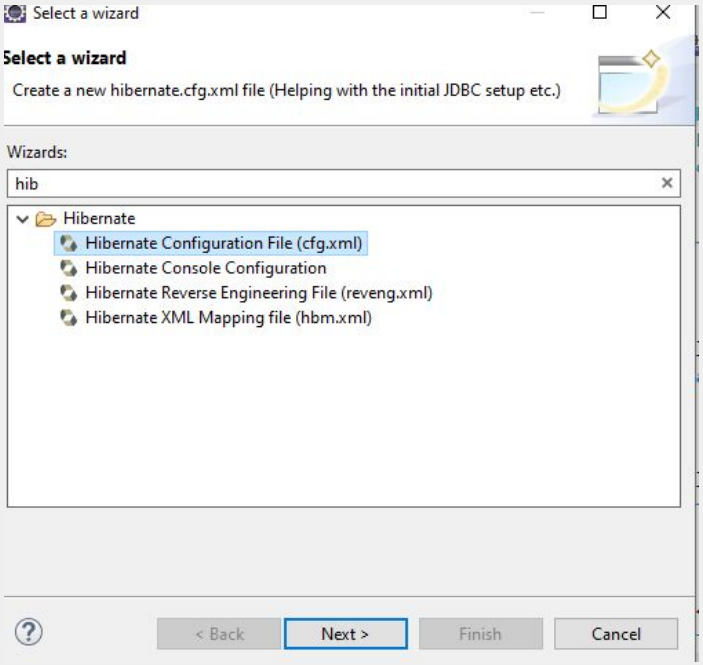


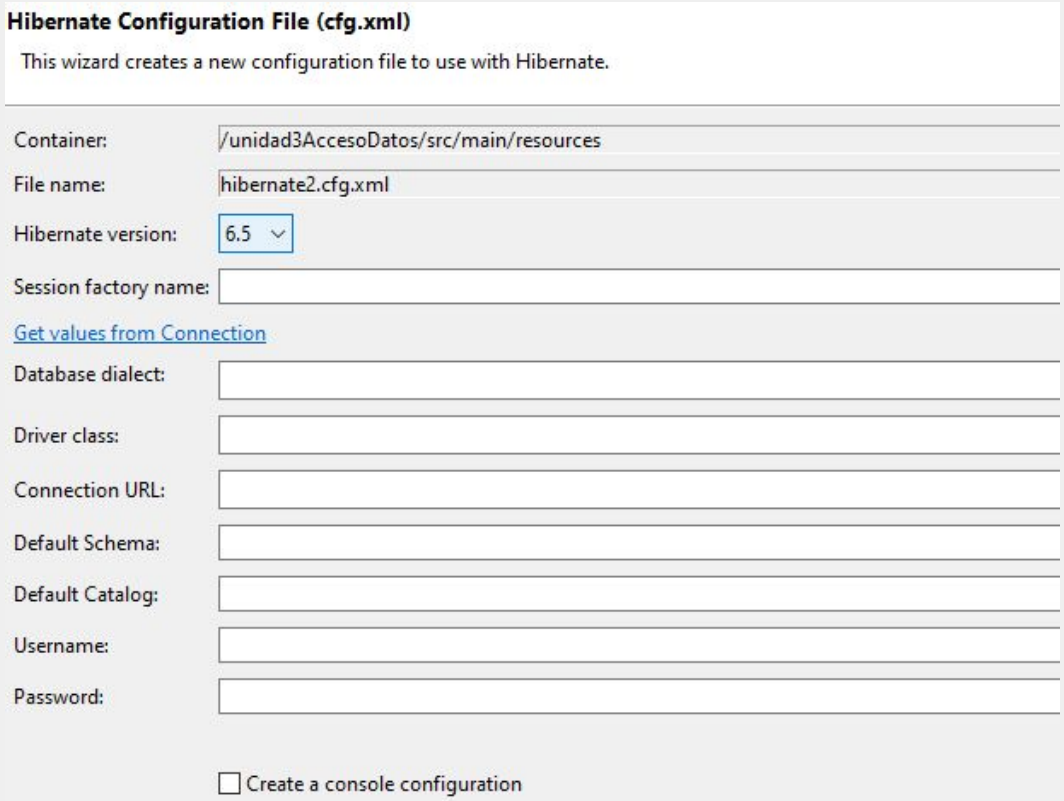
ORM: Hibernate primeros pasos: hibernate.cfg.xml

Se recomienda instalar del Eclipse Marketplace: JBoss Tool

Este plugin nos facilitará trabajar con ficheros de configuración de Hibernate.

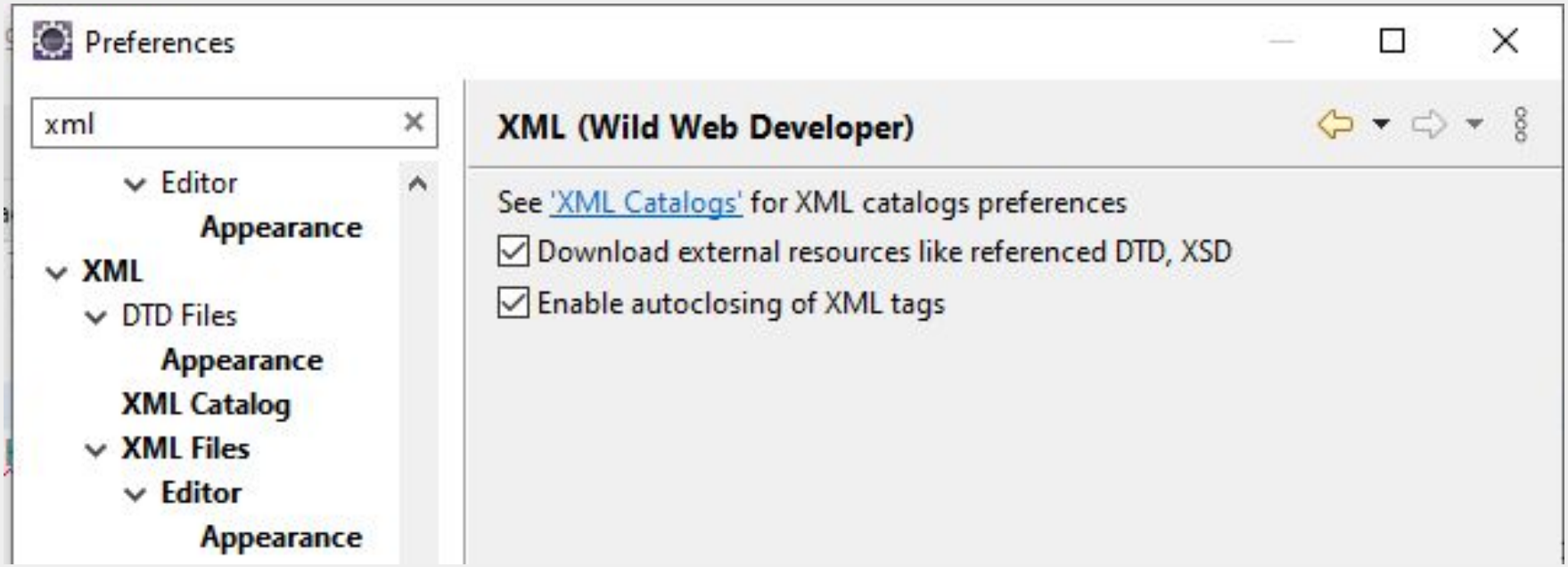
New/Others





ORM: Hibernate primeros pasos: hibernate.cfg.xml

En Eclipse, habilitar:



ORM: Hibernate primeros pasos: hibernate.cfg.xml

Este fichero contiene la configuración con la base de datos.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
      "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
      "https://hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
        <property name="hibernate.connection.driver_class"> com.mysql.cj.jdbc.Driver      </property>
      <property name="hibernate.connection.url">
         jdbc:mysql://localhost:3307/accesoDatosDAM?useSSL=false&amp;serverTimezone=UTC
      </property>
      <property name="hibernate.connection.username">usuario2</property>
      <property name="hibernate.connection.password">usuario</property>
      <!-- Dialect -->
      <property name="hibernate.dialect"> org.hibernate.dialect.MySQLDialect       </property>
      <property name="hibernate.hbm2ddl.auto">update</property>
      <property name="hibernate.show_sql">true</property>
      <property name="hibernate.format_sql">true</property>
      <!-- Entidades -->
      <mapping class="modelo.hibernate.Reunion"/>
  </session-factory>
</hibernate-configuration>
```

Clave para la conexión

Registra tu clase Java

Hibernate primeros pasos: Clases Modelo Java

Notaciones en la clase Modelo:

- **@Entity**: Indica que la clase es una entidad y será mapeada a una tabla en la base de datos.
- **@Table(name = "nombre_tabla")**: (Opcional) Define el nombre de la tabla en la base de datos a la que se mapea la entidad. Si se omite, Hibernate usará el nombre de la clase como nombre de la tabla.
- **@Id**: Especifica el campo que será usado como clave primaria en la tabla.
- **@GeneratedValue**(strategy = GenerationType.IDENTITY): Define cómo se generará el valor de la clave primaria.

Hibernate primeros pasos: Clases Modelo Java

- **@Column(**name = "nombre_columna", nullable = false, length = 100): Mapea un campo a una columna específica en la tabla y define propiedades de la columna, como el nombre, si puede ser null, la longitud máxima, etc.

  Es obligatorio usar la notación @Column(name = "nombreCampo"), si las columnas de la tablas se llaman diferentes que los atributos

- **@Transient**: Indica que el atributo no debe ser persistido en la base de datos.

ORM: Hibernate primeros pasos: Clases Java

```java
//Esta clase representa una entidad
@Entity
@Table(name = "reunion")
public class Reunion {
    // Este campo es la clave primaria
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int idReunion;
    //Es obligatorio usar la notación @Column(name="nombreCampo")
    // Si las columnas de la tablas se llaman diferentes que los atributos
    // @Column(name="fecha")
    private LocalDateTime fecha;
    // @Column(name="asunto")
    private String asunto;
    // Generamos el constructor sin parámetros y los métodos get/set
}
```

Clase -> Tabla

Atributo -> Columna

Atributo -> Columna

imports del paquete jakarta

ORM: Hibernate primeros pasos: Clase HibernateUtil

Para poder utilizar Hibernate necesitamos una sesión:

```
public class HibernateUtil {
    private static StandardServiceRegistry registro;
    private static SessionFactory factoriaSession;

    public static SessionFactory getFactoriaSession() {
        if(factoriaSession == null)        {
            registro = new StandardServiceRegistryBuilder().configure().build();
            MetadataSources sources = new MetadataSources(registro);
            Metadata metadatos = sources.getMetadataBuilder().build();
            factoriaSession = metadatos.buildSessionFactory();
        }
        return factoriaSession;
    }

    public static void shutdown()
    {
        if(registro != null)        {
            StandardServiceRegistryBuilder.destroy(registro);   }
    }}
```

ORM: Hibernate primeros pasos: Test JUnit

Utilizaremos una Test Case de JUnit para probar lo configurado hasta ahora.

Para ello, genera en la carpeta test, el paquete utiles y crea una test case de JUnit 5.

En este test añadiremos un test para probar cada uno de los métodos del CRUD

- Create
- Retrieve
- Update
- Delete

ORM: Hibernate primeros pasos: Test JUnit

****Obteniendo una Sesión**:**

```
// Simplified session opening
Session session = HibernateUtil.getSessionFactory().openSession();
Transaction tx = null;
```

1. Crear (Create/Persist)

```
tx = session.beginTransaction();
Reunion nueva = new Reunion("Reunión de equipo", Lo…
session.persist(nueva);
tx.commit();
```

2. Leer (Retrieve/Find)

```
Reunion r = session.find(Reunion.class, 1);
System.out.println("Asunto: " + r.getAsunto());
```

3. Actualizar (Update)

Los objetos gestionados se actualizan automáticamente.

```
tx = session.beginTransaction();
Reunion r = session.find(Reunion.class, 1);
r.setAsunto("Nuevo Asunto --"); // ¡Listo!
tx.commit();
```

4. Borrar (Delete/Remove)

```
tx = session.beginTransaction();
Reunion r = session.find(Reunion.class, 2);
session.remove(r);
tx.commit();
```

ORM: Hibernate primeros pasos: Test JUnit

```
@Test
void testCreateReunion() {
    Session sesion = HibernateUtil.getFactoriaSession().openSession();
    //Registramos una transacción
    sesion.beginTransaction();
    Reunion reunion = new Reunion();
    reunion.setAsunto("mi reunion de hoy");
    reunion.setFecha(LocalDateTime.now());
    sesion.persist(reunion);
    sesion.getTransaction().commit();
    sesion.close();
}
```

ORM: Hibernate primeros pasos: Test JUnit

```
@Test
void testRetrieveReunion() {
    Session sesion = HibernateUtil.getFactoriaSession().openSession();
    Reunion r = sesion.find(Reunion.class, 1);
    logger.debug("El asunto es:"+r.getAsunto());
    sesion.close();
}
```

ORM: Hibernate primeros pasos: Test JUnit

```
@Test
void testUpdateReunion() {
    Session sesion = HibernateUtil.getFactoriaSession().openSession();
    Reunion r = sesion.find(Reunion.class, 1);
    sesion.beginTransaction();
    r.setAsunto("Nuevo Asunto --");
    sesion.getTransaction().commit();
    sesion.close();
}
```

ORM: Hibernate primeros pasos: Test JUnit

```
@Test
void testDeleteReunion() {
    Session sesion = HibernateUtil.getFactoriaSession().openSession();
    sesion.beginTransaction();
    sesion.remove(sesion.find(Reunion.class, 2));
    sesion.getTransaction().commit();
    sesion.close();
}
```

ORM: Hibernate

Es tu turno:

Agrega una clase Persona con los siguientes atributos/campos:

- **dni**: Campo de tipo String que actúa como clave primaria de la tabla. Este campo debe ser único y tiene un tamaño máximo de 15 caracteres
- **nombreApellido**: Campos de tipo String, obligatorios y con un tamaño máximo de 100 caracteres.
- **edad**: Campo de tipo Integer, que es opcional.
- **email**: Campo de tipo String, que es único y tiene un tamaño máximo de 150 caracteres.
- **fechaNacimiento**: Campo de tipo Date que almacena la fecha de nacimiento (sin hora).
- **telefono**: Campo de tipo String con un tamaño máximo de 15 caracteres, opcional.

Crea la tabla, la clase Java y las operaciones CRUD en tests de una Test Case .

Operaciones CRUD

A continuación seguimos creando nuestra arquitectura.

Vamos a crear una interfaz que contiene todas las operaciones del CRUD.

```
/**
* Esta interfaz contiene las operaciones del CRUD
* Create a element T
* Retrieve a element and all elements: get
* Update a element T
* Delete a element T
* @param <T>
*/
public interface IDao<T> {
	void create(T t);
	T get(int id);
	List<T> getAll();
	void update(T t);
	void delete(T t);
}
```

Operaciones CRUD: AbstractDao

Ahora vamos a crear una clase abstract y genérica que implemente la interfaz IDao: AbstractDao

```java
public abstract class AbstractDao<T> implements IDao<T> {
    private Class<T> clase;

    @Override
    public void create(T t) {
        executeInsideTransaction(t);
    }

    @Override
    public Optional<T> get(int id) {
        Session sesion = HibernateUtil.getFactoriaSession().openSession();
        return Optional.ofNullable(sesion.find(clase, id));
    }
    @Override
    public List<T> getAll() {
        Session sesion = HibernateUtil.getFactoriaSession().openSession();
        String queryString = "FROM " + clase.getName();
        TypedQuery<T> query = sesion.createQuery(queryString, clase);
        List<T> resultados = query.getResultList();
        return resultados;
    }
    @Override
    public void update(T t) {
        Session sesion = HibernateUtil.getFactoriaSession().openSession();
        executeInsideTransaction(sesion, sesion.merge(t));
    }

    @Override
    public void delete(T t) {
        Session sesion = HibernateUtil.getFactoriaSession().openSession();
        //Registramos una transacción
        Transaction tx = sesion.beginTransaction();
        try {
```

Clases ModeloDao

Para evitar tener que implementar las operaciones del DAO por cada modelo, lo que haremos que crearemos un paquete dao, y dentro, una clase dao por cada clase mapeada. Esta clase extenderá AbstractDao. Por ejemplo, para la clase Reunion:

```
public class ReunionDao extends AbstractDao<Reunion>{
    public ReunionDao() {
        setClase(Reunion.class);
    }
 // En esta clase añadiremos el resto de operaciones que queramos //
realizar sobre la colección de reuniones.
}
```

Asociaciones entre entidades: Relaciones

Relaciones Unidireccionales vs Bidireccionales

Decidir si mapear una relación como unidireccional o bidireccional depende del comportamiento que esperas de la relación en tu modelo de datos y cómo necesitas acceder a ella en tu aplicación.

- **Relación unidireccional.** Solo una de las entidades conoce la relación y puede acceder a la otra. Cuándo usarla:
  - Cuando solo necesitas acceso en una dirección.
  - Para simplificar el modelo y reducir la complejidad si la otra dirección no es necesaria.

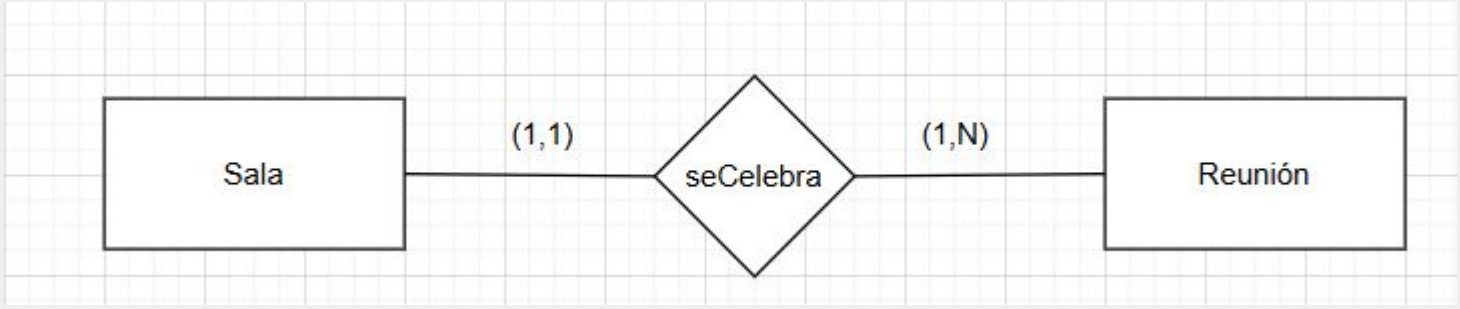
Ejemplo: Un Pedido que tiene una referencia a un Producto. El pedido sabe qué producto lleva, pero el producto no necesita una lista de todos los pedidos donde ha sido incluido.

Asociaciones entre entidades: Relaciones

Relaciones Unidireccionales vs Bidireccionales

- **Relación bidireccional.** Ambas entidades conocen y pueden acceder a la otra. Cuándo usarla:
  - Cuando necesitas acceder a los datos desde ambas direcciones.
  -

Veamos un ejemplo:



Una reunión ocurre en una sala, pero también me puede hacer falta listar, a partir de una sala, las reuniones planificadas para una sala

Asociaciones entre entidades: Relaciones

Relaciones Unidireccionales vs Bidireccionales

- **Relación bidireccional. El Concepto de "Owner" (Propietario)**
  Cuando usas mapeo bidireccional, los ORM necesitan saber qué lado es el "dueño" de la relación para evitar confusiones al generar el SQL.
  - **El Owner:** Es la entidad que "manda" sobre la columna de la clave foránea (FK) en la base de datos. Normalmente es el lado de los "Muchos" (N).
  - **El lado Inverso:** Es simplemente una vista o referencia. Se define usando atributos como mappedBy (en Java/JPA) para indicarle al ORM: "No crees otra columna aquí, usa la que ya definió la otra clase".

Asociaciones entre entidades: Relaciones

Relación bidireccional: puedes navegar la relación en ambos sentidos dentro del modelo de objetos.

Desde Sala → obtener sus Reuniones.

Desde Reunión → acceder a la Sala.

El mapeo bidireccionales un **concepto a nivel de objetos**:

- Sin embargo, ORM necesita saber cómo navegar la relación en memoria y para eso le hace falta las notaciones
- En la base de datos, no existen dos relaciones: solo hay una clave foránea (sala_id en la tabla reunion).
- ORM traduce a SQL. El ORM construye la segunda dirección (Sala → Reuniones) virtualmente, usando esa clave foránea.

Asociaciones entre entidades: Relaciones

Relación bidireccional:¿Para qué sirven las notaciones?

Las anotaciones (@OneToMany, @ManyToOne, mappedBy, etc.) sirven para que el ORM:

- Sepa cómo mapear la relación en SQL (qué tabla tiene la FK, si hay tabla intermedia, etc.).
- Gestione la persistencia automática: al guardar una Sala, puede guardar también sus Reuniones si hay cascade.

Sin notación: `@OneToMany(mappedBy = "sala")`:

- Para el ORM, esa lista es solo un atributo en memoria, no una relación persistente.
- No se generará ningún join ni se cargará desde la BD.
- Hibernate puede incluso crear una tabla intermedia para mapear esa colección, porque no sabe que la FK está en Reunion.

Asociaciones entre entidades: Relaciones

Relación bidireccional:¿Cómo forzar que se cree la tabla intermedia?

Usando la anotación @JoinTable

```
@Entity
public class Sala {
   …
   @OneToMany
   @JoinTable(
      name = "sala_reunion",
      joinColumns = @JoinColumn(name = "sala_id"),
      inverseJoinColumns = @JoinColumn(name = "reunion_id")
   )
   private List<Reunion> reuniones;
}
```

Esto genera una tabla sala_reunion con dos columnas (sala_id, reunion_id) en lugar de poner la FK directamente en Reunion.

Sólo mapeamos así, si queremos guardar algún atributo en la relación. Aunque lo ideal, sería en lugar de esto, modelar la propia relación como una clase del modelo y mapear la relación con las entidades.

Bidireccionalidad y sincronización de objetos

Java no sincroniza los objetos automáticamente

En una relación bidireccional (ej. Departamento <-> Empleado):

1. **Hibernate** solo mira el lado "propietario" (el que tiene @ManyToOne) para guardar en la DB.
2. **Tu Código** necesita ver ambos lados para que la lógica sea coherente en memoria antes de hacer el save.

Asociaciones entre entidades: Relaciones 1,N

Relaciones Unidireccionales vs Bidireccionales

- **Mapeo unidireccional:**
  La relación sólo se mapeará añadiendo un atributo Sala en la reunión. De manera que sabremos en qué sala ocurre la reunión pero no la lista de reuniones que ocurren en una sala. Para ello añadiremos el atributo Sala en la clase **Reunion**:

```
@ManyToOne(cascade = CascadeType.PERSIST)
@JoinColumn(name="idSala")
private Sala sala;
```

  y también lo añadiremos en el constructor con parámetros y sus métodos setter and getter

Asociaciones entre entidades: Relaciones 1,N

Relaciones Unidireccionales vs Bidireccionales

- **Mapeo bidireccional:**
  Si por el contrario en nuestra aplicaciones queremos tener la información en los **dos sentidos.** En el lado propietario (N) usamos @ManyToOne con @JoinColumn y en el inverso @OneToMany(mappedBy=...)
  En nuestro ejemplo, tendremos que mapear, **no sólo en el objeto Reunion** sino que también tendremos que añadir una lista de Reuniones en la clase **Sala**.

```
@OneToMany(mappedBy = "sala")
   private List<Reunion> reuniones;
```

  y también lo añadiremos en el constructor con parámetros y sus métodos setter and getter

Asociaciones entre entidades: Relaciones 1,N

Relaciones Unidireccionales vs Bidireccionales

- **Mapeo bidireccional:**
  En una relación bidireccional, debes asegurarte de que ambos lados de la relación estén sincronizados. Para ello deberíamos:
  1. En el constructor generar la lista vacía.
  2. Y agrego un método similar a este en la clase **Sala**:

```
public void addReunion(Reunion r)
{
        this.reuniones.add(r);
        r.setSala(this);
}
```

```
@OneToMany(mappedBy = "sala", cascade = CascadeType.ALL)
   private List<Reunion> reuniones;
```

Asociaciones entre entidades: Relaciones 1,N

Relaciones Unidireccionales vs Bidireccionales

Relación	Implementación correcta	Tablas generadas
1:N Unidireccional	Solo `@ManyToOne` en "muchos"	`sala` y `reunion` con clave foránea en `reunion`
1:N Bidireccional	`@OneToMany(mappedBy = ...)` en "uno" + `@ManyToOne` en "muchos"	Misma estructura, pero permite navegación en ambos sentidos

Asociaciones entre entidades: Relaciones 1,N

Notaciones y atributos que pueden aparecer con @OneToMany:

- **mappedBy**: Define el lado que no es el propietario de la relación (el que **no contiene la clave foránea**). Especifica el nombre del atributo en la entidad de la otra tabla que mapea esta relación.
- **fetch**: Define cómo se carga la relación. Puede ser:
  - **FetchType.LAZY**: Los datos se cargan sólo cuando se accede a la colección. Esta es la opción predeterminada para @OneToMany y suele ser la más eficiente.
  - **FetchType.EAGER**: La colección se carga inmediatamente junto con la entidad principal.

Asociaciones entre entidades: Relaciones 1,N

Notaciones y atributos que pueden aparecer con @OneToMany:

- **@JoinColumn** Con qué campo de la otra tabla se une. Suele ser una FK. Tiene los atributos
  - **name** Nombre de la columna FK
  - **nullable** Si false, esta columna es obligatoria
  - **unique** Si la columna es única
  - **insertable** y **updatable** Si la columna se puede insertar o actualizar

- **atributo optional** Indica si la relación es opcional, pudiendo existir valores nulos . Cardinalidad (0, 1) @ManyToOne(optional = false) SI una reunión puede no tener sala

Asociaciones entre entidades Relaciones 1,N

Notaciones y atributos que pueden aparecer con @OneToMany:

- **cascade**. Operaciones en cascada. Puede ser:
  - **CascadeType.ALL**: Todas las operaciones (persist, merge, remove, refresh, detach).
  - **CascadeType.PERSIST**: Persistir la entidad padre también persiste los hijos.
  - **CascadeType.MERGE**: Al actualizar la entidad padre, también se actualizan los hijos.
  - **CascadeType.REMOVE**: Al eliminar la entidad padre, se eliminan los hijos.
  - **CascadeType.REFRESH**: Si se recarga la entidad padre desde la base de datos, también se recargan los hijos.
  - **CascadeType.DETACH**: Si se separa la entidad padre del contexto de persistencia, también se separan los hijos.

Asociaciones entre entidades: Relaciones 1,N

Notaciones y atributos que pueden aparecer con @OneToMany :

Importante:

La entidad que tiene el atributo cascade, será la que haga propagar a la otra entidad.

Si volvemos a nuestro ejemplo, al persistir una Sala → Se persistirán los cambios en reunión.

Pero no a la inversa.

```
Sala sala =  new Sala("sala azul");
Reunion nuevaReunion = new Reunion(LocalDateTime.now().plusDays(3), "Reunión futura1_", sala);
//daoReunion.create(nuevaReunion);
salaDao.create(sala);
```

Asociaciones entre entidades: : Relaciones 1,1

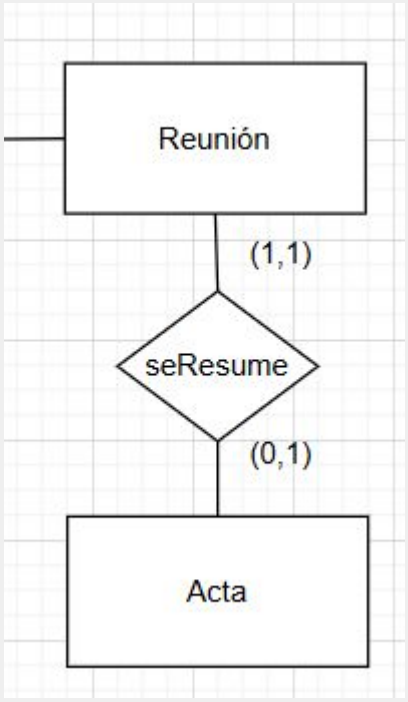
Unidireccional:

- Sólo una entidad conoce la otra.
- Se pone **@OneToOne** y **@JoinColumn(name =”nombre del campo en la tabla”)** en la entidad que tendrá la clave foránea.

Bidireccional:

- Existe navegabilidad en ambos sentidos
- Además de mapearlo como unidireccional, se añade en el lado no propietario @OneToOne **(mappedBy=”nombreAtributoCorrespondienteClasePropietaria”**

Asociaciones entre entidades: : Relaciones 1,1



1) Crea la Clase Acta con un id y un String de contenido. Agrega un atributo reunion
2) Agrega el mapeo en hibernate.cfg
3) Ahora mapeareamos la relación:
   a) En la clase Acta: **Unidireccional**

```
@OneToOne
@JoinColumn (name= "idReunion")
private Reunion reunion;
```

   b) En la clase Reunion, añadimos: Con **bidireccional**

```
@OneToOne ( optional=true)
private Acta acta;
```

La **unidireccional** sería añadiendo sólo a Reunión el atributo Acta

Asociaciones entre entidades Relaciones 1,1

Cascade:

- A pesar de que la relación sea bidireccional, Hibernate no maneja automáticamente la sincronización entre los dos lados de la relación.
- En cada relación existe un **lado propietario**. Es el lado que tiene la notación @JoinColumn. En ese caso, al actualizar el lado propietario, el cambio se propagará al otro lado.

Veamos un ejemplo:

```java
@Entity
@Table(name = "acta")
public class Acta {

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

    // Relación uno a uno con la entidad Reunion
    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "idReunion")
    private Reunion reunion;
```

```java
// Esta clase representa una entidad
@Entity
@Table(name = "reunion")
public class Reunion {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int idReunion;
    // Relación uno a uno con la entidad Acta
    @OneToOne(mappedBy = "reunion", optional=true)
    private Acta acta;
```

Asociaciones entre entidades Relaciones 1,1

Cuando tienes una relación bidireccional de tipo uno a uno, tienes dos entidades relacionadas, y en ambos lados puedes navegar por la relación.

Sin embargo, JPA solo coloca la clave foránea en uno de los lados de la relación. Este lado es considerado el **propietario de la relación.**

1. Lado propietario de la relación.Es la entidad que posee la clave ajena en la base de datos. Se define con el lado que utiliza la notación @JoinColumn y cascade
2. El lado no propietario solo tiene una referencia a la entidad propietaria a nivel de datos, pero **no almacena la clave foránea.** Considera que puede puede acceder a la otra entidad a través de una referencia pero no tendrá una columna de clave ajena.

Asociaciones entre entidades Relaciones 1,1

Cascade y entidad propietaria

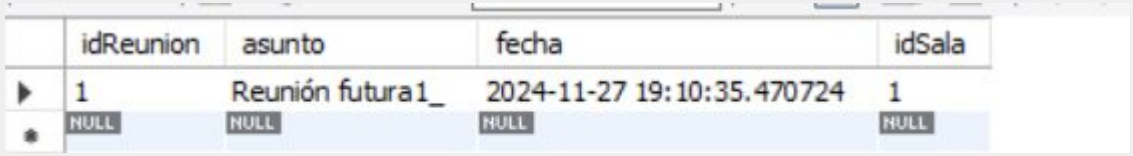
Consideremos que el lado Propietario es Acta. Al persistir un acta, se creará también la Persona (Cascade ALL)

En este caso, Hibernate creará el Acta con el id de Reunión y se creará la reunión.

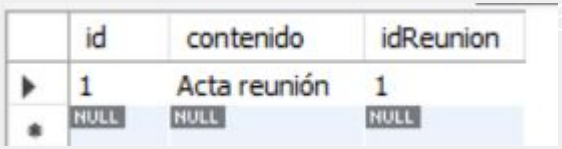
1) Persistimos el lado propietario:

```
Reunion r1 = new Reunion(LocalDateTime.now().plusDays(3), "Asunto", s1);
//reunionDao.create(r1);
// Acta
ActaDao actaDao = new ActaDao();
Acta a1 = new Acta("Acta reunión", r1);
actaDao.create(a1);
```

En este ejemplo, al crear un acta se creará la reunión y se le pondrá a la reunión el identificador de acta



idReunion	asunto	fecha	idSala
1	Reunión futura1_	2024-11-27 19:10:35.470724	1
NULL	NULL	NULL	NULL



id	contenido	idReunion
1	Acta reunión	1
NULL	NULL	NULL

Asociaciones entre entidades Relaciones 1,1

Cascade y entidad NO propietaria

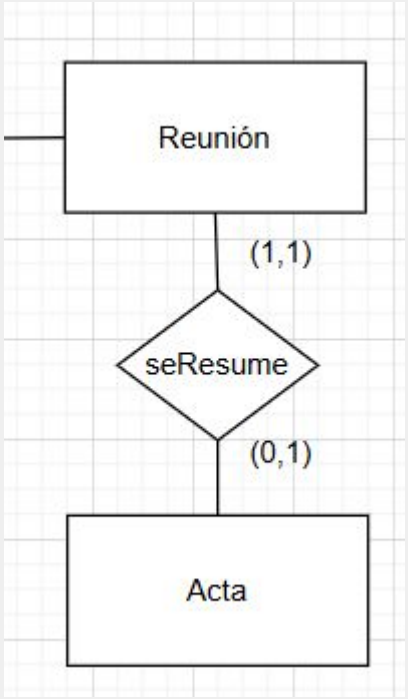
Si creo una Reunión y le asigno un acta, el acta debe ser persistida de manera explícita. No se propaga

2) Persistimos el lado NO propietario:

```
Reunion r1 = new Reunion(LocalDateTime.now().plusDays(3), "Asunto", s1);
// Acta
ActaDao actaDao = new ActaDao();
Acta a1 = new Acta("Acta Reunion ", r1);
r1.setActa(a1);
//actaDao.create(a1);
reunionDao.create(r1);
```

Al guardar la reunión te dice que hay cambios sin propagar (la sala), devolviendo una excepción

Asociaciones entre entidades: : Relaciones 1,1




```
public Acta(String contenido, Reunion reunion) {
        super();
        this.contenido = contenido;
        this.reunion = reunion;
        reunion.setActa(this);
    }
```

Además como la relación es bidireccional. Deberemos asegurar la consistencia de datos.

Importante: Con la relaciones bidireccionales evitar las llamadas reflexivas→ redundancia. A considerar la reflexión con los constructores, set y toString.

Además habría que contemplar que una reunión puede no tener acta.

Asociaciones entre entidades: : Relaciones 1,1

Importante: Con la relaciones bidireccionales evitar las llamadas reflexivas→ redundancia

A considerar la reflexión con los constructores, set y toString.

Ejemplo:

```
@Override
public String toString() {
    return "Reunion [idReunion=" + idReunion +
            ", fecha=" + fecha + ", asunto=" + asunto +
            ", sala=" + sala
            + ", acta=" + acta + "]";
}
```

```
@Override
public String toString() {
    return "Acta [idActa=" + idActa +
            ", contenido=" + contenido +
            ", reunion=" + reunion + "]";
}
```

Posible solución:

```
@Override
public String toString() {
    String cadena = "Reunion [idReunion="
            + idReunion + ","
            + " fecha=" + fecha + ", asunto="
            + asunto + ", sala=" + sala;
    if(this.acta != null)
        cadena= cadena+", acta= "
    + this.getActa().getIdActa();
    return cadena+"]";

}
```

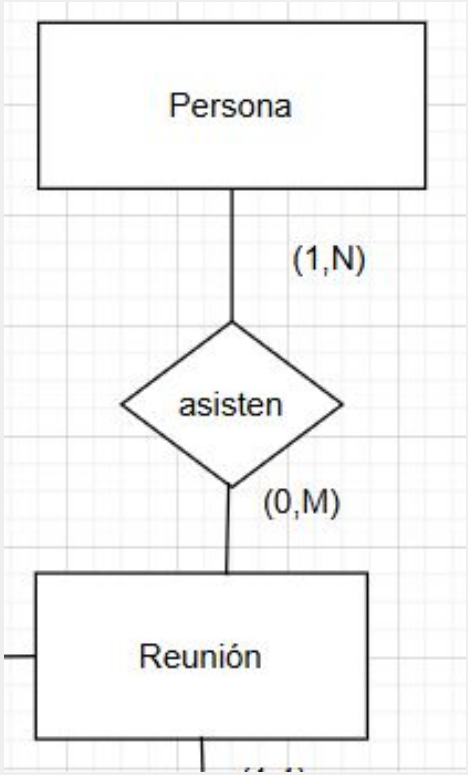
```
@Override
public String toString() {
    return "Acta [idActa=" + idActa
            + ", contenido=" + contenido
            + ", reunion="
            + reunion.getAsunto() + "]";
}
```

Asociaciones entre entidades: : Relaciones M,N

Ejemplo:

Una Persona puede asistir a varias, o ninguna reunión) y a una reunión asistirá una o varias personas.

Cuando se tienen este tipo de relaciones, se descomponen en dos relaciones de 1:N, es decir se crea una nueva tabla con las claves primarias de las entidades implicadas. . En nuestro caso, será la tabla persona_reunion y contendrá las dos claves primarias de dichas entidades.



Asociaciones entre entidades: : Relaciones M,N

Ejemplo:

Como se ha comentado, debemos gestionarlo como dos relaciones

1. Agregaremos a la clase **Persona** un Set de Reuniones junto con **@ManyToMany**
2. En la clase **Reunion** haremos los mismo con una colección de participantes

Sin embargo, para lograr que todo funcione, hemos tenido que decir qué personas participan en las reuniones y en qué reuniones participan las personas. → **Redundante, y propenso a errores.**

Asociaciones entre entidades: : Relaciones M,N

```java
@Entity
@Table(name = "persona")
public class Persona {
    @Id
    private String dni;

    @ManyToMany()
    Set<Reunion> reuniones;
```

```java
@Table(name = "reunion")
public class Reunion {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int idReunion;

    @ManyToMany( mappedBy="reuniones", cascade= CascadeType.ALL )
    private Set<Persona> personas;
```

Crear el dao de Persona y añadir personas a una reunión. Probar primero sin cascade. Prueba agregando personas y reuniones nuevas. Y con cascade y sin ella. cascade= CascadeType.ALL en Reunion.

Asociaciones entre entidades: : Relaciones M,N

Probar sin generar dao de Persona a añadir personas a una reunión. Probar primero sin cascade. Prueba con personas y reuniones nueva. Y con cascade y sin ella. cascade= CascadeType.ALL en Reunion.

```
Reunion nuevaReunion = new Reunion(LocalDateTime.now().plusDays(3), "Reunión futura1_", sala);
dao.create(nuevaReunion);
Date fechaN = null;
try {
        fechaN = new SimpleDateFormat("yyyy-MM-dd").parse("2000-11-04");
} catch (ParseException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
}
Persona p = new Persona("12345678p", "Rosario Rosa", "rosaio@gmail.com", fechaN, "612345789");
nuevaReunion.addPersona(p);
dao.update(nuevaReunion);
```

Asociaciones entre entidades: : Relaciones M,N

Probar:

Crear el dao de Persona y añadir personas a una reunión.
Probar primero sin cascade. Prueba con personas y reuniones nueva. Y con cascade y sin ella. cascade= CascadeType.ALL en Reunion.

La forma más sencilla de gestionar la redundancia es permitiendo añadir (con método add) elementos a las colecciones pero no la asignación total de estas.
En nuestro caso:

1. Generamos, en Reunión, el método addParticipante y removeParticipante
2. Generamos en Persona, el método addReunion y removeReunion
3. Eliminamos los métodos setReuniones y setParticpantes.

Asociaciones entre entidades: : Relaciones M,N

Solución:

1. En los constructores inicializo las listas vacias
2. Generamos el método addPersona y removePersona
3. Generamos el método addReunion y removePersona
4. Eliminamos los métodos setReuniones y setPersonas.

```
public void addPersona(Persona p)
{
    this.personas.add(p);
    if(!p.getReuniones().contains(this))
    {
        p.getReuniones().add(this);
    }
}

public void removePersona(Persona p)
{
    this.personas.remove(p);
    if(p.getReuniones().contains(this))
    {
        p.getReuniones().remove(this);
    }
}

public Set<Persona> getPersonas() {
    return personas;
}
```

En la clase Reunion

```
public void addReunion(Reunion r)
{
    this.reuniones.add(r);
    if(!r.getPersonas().contains(this))
    {
        r.getPersonas().add(this);
    }
}

public void removeReunion(Reunion r)
{
    this.reuniones.remove(r);
    if(r.getPersonas().contains(this))
    {
        r.getPersonas().remove(this);
    }
}

public Set<Reunion> getReuniones() {
    return reuniones;
}
```

En la clase Persona

Asociaciones entre entidades: : Relaciones M,N

Solución:

```
nuevaReunion = daoReunion.mergeaObjeto(nuevaReunion);
//Sincronizo nuevaReunion cargando su id

Persona p = new Persona("12345677p", "Pepa Rosa",
"rosaio@gmail.com", fechaN, "612345789");
personaDao.create(p);

//Creo la persona en la BBDD
p.addReunion(nuevaReunion);
personaDao.update(p);
//Propago el cambio. En este momento se rellena la tabla intermedia.
```

Sincronizar y recuperar objeto con el id generado

Imagina que damos de alta una entidad en la base de dato cuyo id es autogenerado. Al persistirlo en la BD, se rellena el id. Pero, ¿cómo sabemos qué identificador se ha creado?

Para ello tendremos, que generar un método que nos sincronice el objeto con la base de datos y nos lo devuelva. Crearemos un nuevo método, en AbstractDao. De esta forma ya tendríamos el identificador generado

```
public T mergeaObjeto(T t)
{
    Session sesion = HibernateUtil.getFactoriaSession().openSession();
    return sesion.merge(t);
}
```

Comitear una transacción

La implementación realizada en AbstractDao comitea cada transacción.

Al comitear una transacción, el objeto implicado y los posibles objetos que se propagan por según el atributo cascade, quedan desvinculados de la sesión.

Es por ello, que deberemos volver a vincularlos. Alternativas:

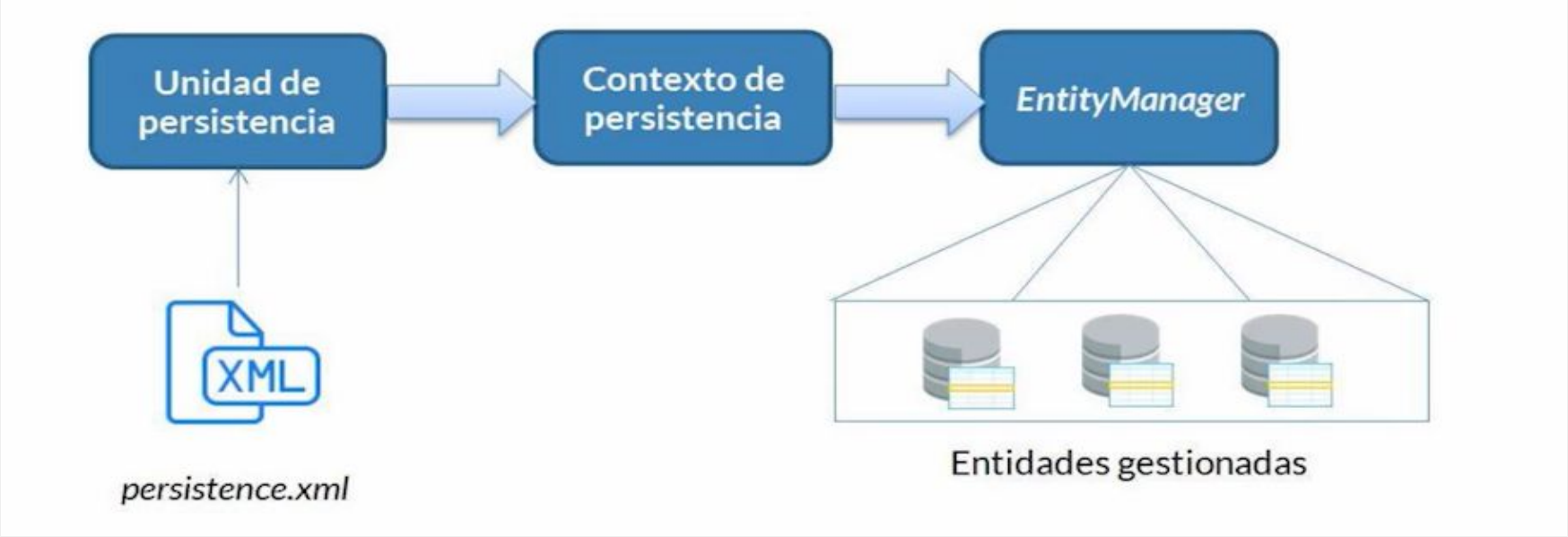
- Lo recuperamos con get(id) aunque tendríamos que saber con qué id se ha guardado.
- Utilizando una llamada al método generado en la transparencia anterior (*mergeaObjeto*) de esta forma recuperamos el objeto con su id y además lo volvemos a sincronizar en la sesión.

Ciclo de vida de una transacción

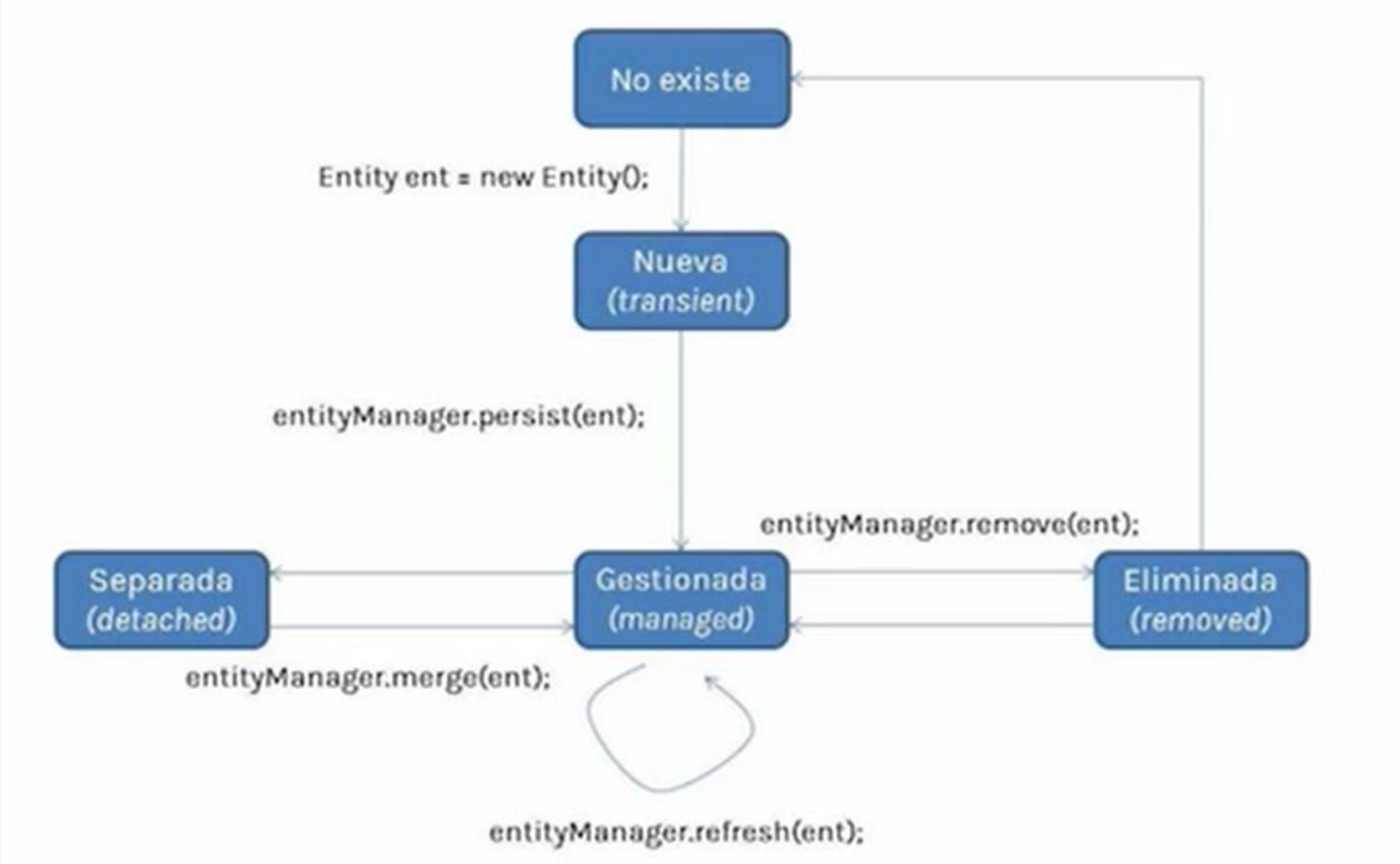
Concepto de JPA

Unidad de Persistencia: Representación de todas las entidades que pueden ser mapeadas en una base de datos concreta.

Contexto de persistencia: Representa las entidades gestionadas en un momento determinado. El contexto de persistencia nos permite obtener una referencia a las entidades gestionadas en ese momento.



Ciclo de vida de una transacción



Ciclo de vida de una transacción

Propagar cambios entre el EntityManager y la BBDD:

```
Person person = entityManager.find( Person.class, personId );
person.setName("John Doe");
entityManager.flush();
```

Propagar cambios entre la BBDD y EntityManager :

```
Person person = entityManager.find( Person.class, personId );

entityManager.createQuery( "update Person set name = UPPER(name)" ).executeUpdate();

entityManager.refresh( person );
assertEquals("JOHN DOE", person.getName() );
```

Ciclo de vida de una transacción

Reintegrar en el contexto de persistencia un objeto:

```
Person person = entityManager.find( Person.class, personId );
//Al limpiar el entityManager la entidad se convierte en detached
entityManager.clear();
person.setName( "Mr. John Doe" );

person = entityManager.merge( person );
```

Cuando se comitea una transacción también se saca del Entity Manager

Comitear una transacción

Veamos un ejemplo:

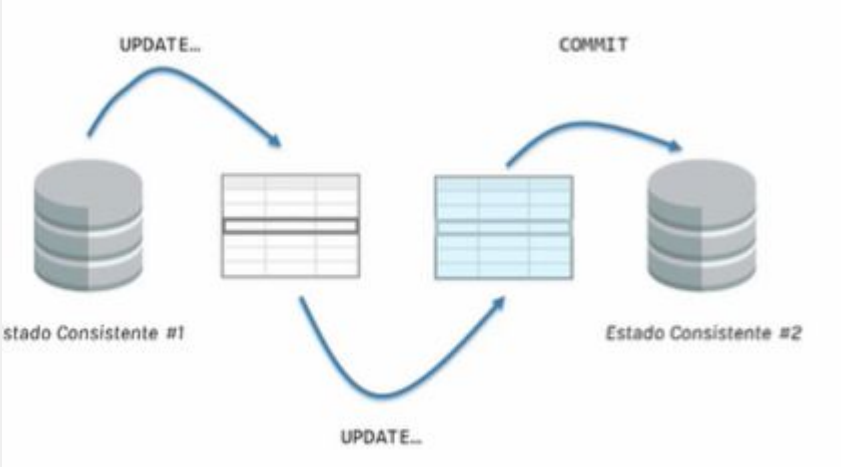
```
SalaDao salaDao = new SalaDao();
Sala sala =  new Sala("sala azul");
Reunion nuevaReunion = new Reunion(LocalDateTime.now().plusDays(3),
"Reunión futura1_", sala);
//daoReunion.create(nuevaReunion);
salaDao.create(sala);
//Aquí se crea la sala y la reunión. SIn embargo, ambos objetos quedan
//  desvinculados. Es por ello que la siguiente instrucción da un error.
Acta acta = new Acta("Este el acta de la reunión futura1", nuevaReunion);
actaDao.create(acta);
```

Para solucionarlo, debemos recuperar el objeto reunión, añadiendo una llamada con get:

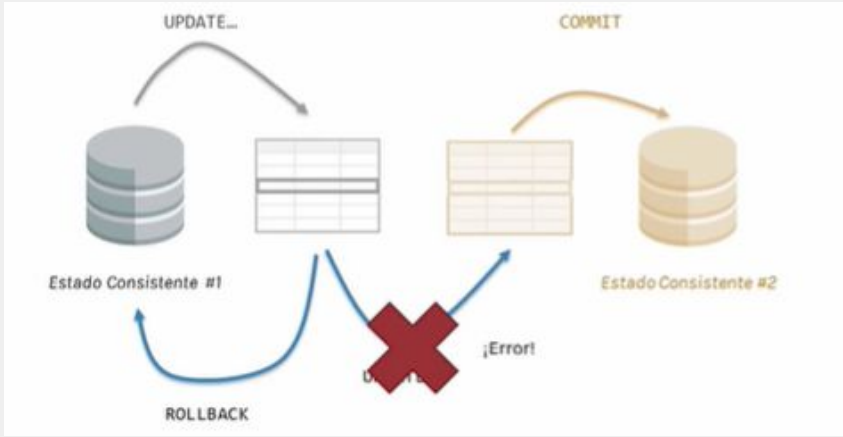
```
salaDao.create(sala);
nuevaReunion = daoReunion.mergeaObjeto(nuevaReunion);
Acta acta = new Acta("Este el acta de la reunión futura1", nuevaReunion);
actaDao.create(acta);
```

Transacciones

Una transacción permite realizar una serie de operaciones de manera atómica (como si fuera una sola)



Si algo fallara, se realizaría el rollback de la transacción completa (de todas las operaciones que contiene)



Transacciones

Las transacciones se:

- Inician con el método begin()
- Se comitean, con el método commit()
- O se revierten con el método rollback()

```
try {
    //Iniciamos una transacción
    em.getTransaction().begin();
    //Persistimos los datos
    em.persist(userAccount);
    //Commiteamos la transacción
    em.getTransaction().commit();
    System.out.println("El objeto ha sido dado de alta correctamente. Muchas gracias.");
} catch(Exception e) {
    System.out.println("El objeto no ha sido dado de alta correctamente. Disculpe las molestias");
    System.err.println(e.getMessage());
    if (em.getTransaction().isActive()) {
        em.getTransaction().rollback();
    }
}
```

Transacciones

Si utilizamos los métodos de AbstractDao, **cada operación es una transacción pero esto podría no ser así.** Lo que se conoce como **autocommit.**

Realmente, lo óptimo es definir las diferentes transacciones que tienen sentido en nuestra aplicación. Por ejemplo:

- Imagina que tenemos un formulario de alta un cliente con su domicilio. El domicilio está almacenado en una entidad distinta. En ese caso, tendría sentido entender esta petición como una transacción. De forma que el alta se realice atómicamente.

Es tu turno:

Añade un método en tu clase PersonaDao que reciba una lista de Personas y los persista en la BBDD con una sola transacción.

Consultas - HQL: Hibernate Query Language

La HQL es un lenguaje **orientado a objetos que se parece al SQL**, pero en lugar de tablas y columnas, **trabaja con entidades y sus atributos.**
Hibernate traduce consultas HQL en consultas específicas de la base de datos para realizar acciones.
La interfaz de consulta proporciona los métodos y la funcionalidad para representar y manipular una consulta HQL de forma orientada a objetos.

Regla de oro:
Las palabras clave (SELECT, FROM) no distinguen mayúsculas, pero los nombres de las clases Java y sus atributos sí

Consultas - HQL: Hibernate Query Language

Crearemos consultas con el método createQuery del Entity Manager.

```
TypedQuery<Reunion> query = sesion.createQuery(cadenaQuery)
```

HQL tiene 4 posibles sentencias. Es decir, dentro de esa cadenaQuery puedo querer hacer alguna de estas 4 operaciones, aunque lo más habitual es que lo usemos para consultas (select)

- select
- update
- delete
- insert

Importante: la operaciones no se reflejan si no hacemos el commit

Consultas - HQL: Hibernate Query Language

```java
// Select Básico con Parámetro
String hql = "FROM Reunion r WHERE r.asunto = :tema";

// Join Implícito (Navegación por el grafo)
String hql2 = "SELECT r.sala.nombre FROM Reunion r";

// Join Fetch (Optimización - Evita N+1)
String hql3 = "FROM Sala s JOIN FETCH s.reuniones";
```

Tipos de Resultado

Objeto Único (Entidad)

Lista de Objetos

Array de Objetos
(Object[] - Proyecciones)

Consultas - HQL: Hibernate Query Language

Consulta básica:

Select * from Esto devuelve una lista de Entidades T

```
String queryString = "FROM " + clase.getName();
TypedQuery<T> query = sesion.createQuery(queryString, clase);
  List<T> resultados = query.getResultList();
```

Ejemplo:

```
Session sesion = HibernateUtil.getFactoriaSession().openSession();
String queryString = "FROM "+ Reunion.class.getName();

TypedQuery<Reunion> query = sesion.createQuery(queryString,
Reunion.class);
List<Reunion> reuniones = query.getResultList();
```

Consultas - HQL: Hibernate Query Language

Ejecución de una consulta HQL

- Devuelve una lista de entidades T
  List<T> libros = hqlQuery.getResultList();
- Si estamos seguros de que devuelve un único resultado (búsquedas por id)
  T libro = hqlQuery.getSingleResult();

Consultas - HQL: Hibernate Query Language

Consulta de un campo de una entidad

```
String queryString = "SELECT nombreCampo FROM "+ getClase().getName();

TypedQuery<TipoCampo> query = sesion.createQuery(queryString,
TipoCampo.class);

List<TipoCampo> reuniones= query.getResultList();
```

Ejemplo:

```
String queryString = "SELECT idReunion FROM "+ getClase().getName();

TypedQuery<Integer> query = sesion.createQuery(queryString, Integer.class);

List<Integer> idReuniones= query.getResultList();
```

Consultas - HQL: Hibernate Query Language

Consulta de varios campos de una entidad

En este caso queremos devolver varios campos pero no todos de una entidad.

Hibernate devolverá los resultados como una lista de objetos de tipo Object[], donde cada posición del array corresponde a un campo seleccionado.

```
String queryString = "SELECT campo1, campo2 FROM "+ getClase().getName();
TypedQuery<Object[]> query = sesion.createQuery(queryString, Object[].class);
List<Object[]> resultados= query.getResultList();
```

A veces se crea un tipo de clase mapeador que llamaremos el nombre de nuestro ModeloDTO. Esta clase sólo contendría los atributos que devuelve la consulta.

Consultas - HQL: Hibernate Query Language

Consulta de varios campos de una entidad

Ejemplo:

```
String queryString = "SELECT idReunion, asunto FROM "+ getClase().getName();

TypedQuery<Object[]> query = sesion.createQuery(queryString, Object[].class);

List<Object[]> resultados= query.getResultList();

for( Object[] o: resultados)
{
      logger.debug("id:"+o[0]);
      logger.debug("asunto:"+o[1]);
}
```

Consultas - HQL: Hibernate Query Language

Consultas parametrizadas:

Usaremos el método setParameter y le pasaremos el parámetro mediante :nombreParametro

```
String queryString = "FROM "+ getClase().getName()+" WHERE fecha < :fechaParametro";

TypedQuery<Reunion> query = sesion.createQuery(queryString, getClase());
query.setParameter("fechaParametro", fechaAnterior);
List<Reunion> reuniones= query.getResultList();
sesion.close();
```

Consultas - HQL: Hibernate Query Language

Count:

```
String queryString = "SELECT count(*) FROM "+ getClase().getName();

TypedQuery<Long> query = sesion.createQuery(queryString, Long.class);

Long numElementos = query.getSingleResult();
```

Consultas - HQL: Hibernate Query Language

Consultas agregadas: count, max, avg

```
// Definición de la consulta HQL
String queryString = "SELECT count(*) FROM " + getClase().getName();

// Creación de la TypedQuery especificando que el resultado es Long
TypedQuery<Long> query = sesion.createQuery(queryString, Long.class);

// Ejecución y obtención del número total de elementos
Long numElementos = query.getSingleResult();
```

HQL

Consultas con filtros WHERE

Es tu turno:

El operador **LIKE** nos permite buscar un patrón dentro de una cadena.
asunto LIKE '%palabraClave%'

```
String hql = "FROM Alumno a WHERE a.nombre LIKE :patron";
Query query = session.createQuery(hql);
query.setParameter("patron", "A%"); //ALgo que empieza por A mayúsculas
List<Alumno> lista = query.list();
```

Es tu turno:

Genera un método dentro de ReunionDao que devuelva las reuniones que tienen en su asunto la palabra “Reunión”. Para ello tendrás que usar el operador like que se encuentra en CriteriaBuilder.

Consultas - HQL: Hibernate Query Language

Consultas con Joins Joins implícitos

Un join implícito ocurre cuando accedes a una propiedad de asociación directamente en la cláusula FROM o WHERE.
El ORM traduce esa navegación en un **INNER JOIN** automático.

Supongamos que tenemos Sala y Reunion con relación 1:N:

```
String hql = "FROM Reunion r WHERE r.sala.nombre = :nombreSala";
```

- No escribes JOIN explícitamente.
- Hibernate detecta que r.sala es una asociación y genera el JOIN necesario contra la tabla Sala.
- Siempre será un INNER JOIN.

Consultas - HQL: Hibernate Query Language

Consultas con Joins explícitos

Un **join explícito** se escribe con la palabra clave **JOIN**. Esto te da más control: puedes elegir INNER JOIN, LEFT JOIN, y además usar FETCH.

```
String hql = "FROM Sala s JOIN s.reuniones r WHERE r.tema = :tema";
```

- Aquí se declara explícitamente la relación.
- Puedes usar alias (r) para referirte a las reuniones en la cláusula WHERE o SELECT.

Consultas - HQL: Hibernate Query Language

Consultas con Joins explícitos con FETCH

El modificador **FETCH** indica que quieres traer la asociación completa en la misma consulta, evitando el problema del lazy loading

```
String hql = "FROM Sala s LEFT JOIN FETCH s.reuniones";
```

- Esto carga todas las salas junto con sus reuniones en una sola consulta.
- Sin FETCH, Hibernate solo cargaría las salas y dejaría las reuniones en lazy, disparando más consultas al acceder.
- Con FETCH, se inicializa la colección inmediatamente.

Consultas - HQL: Hibernate Query Language

Es tu turno:

Compara el resultado al ejecutar:

- `FROM Sala s;`
  y accede luego a s.getReuniones() en código

  Supongamos que hay 3 salas en la BD:

```
List<Sala> salas = session.createQuery("FROM Sala", Sala.class).list();
    for (Sala s : salas) {
       System.out.println(s.getReuniones().size());
    }
```

  - Hibernate hace 1 SELECT para traer las salas. Luego, al acceder a getReuniones(), dispara otro SELECT por cada sala.
  - Total: 4 consultas (1 + 3).

  `FROM Sala s LEFT JOIN FETCH s.reuniones`
  Hibernate hace una sola consulta con JOIN y carga todas las salas junto con sus reuniones.

Consultas - HQL: Hibernate Query Language

Consultas CON NOMBRES

- Comparación exacta → **= :param**

```
String hql = "FROM Sala s WHERE s.nombre = :nombre";
query.setParameter("nombre", "Sala Azul");
```

- Coincidencias parciales → **LIKE**

```
String hql = "FROM Sala s WHERE s.nombre LIKE :patron";
query.setParameter("patron", "%Azul%");
```

- Proyección → SELECT s.nombre

```
String hql = "SELECT s.nombre FROM Sala s";
```

- Join con filtro → **JOIN** s.reuniones r WHERE s.nombre = …

```
String hql = "SELECT r.tema FROM Sala s JOIN s.reuniones r WHERE
s.nombre = :nombreSala";
```

- Join con **fetch** → carga asociaciones completas en una sola consulta

```
String hql = "FROM Sala s LEFT JOIN FETCH s.reuniones WHERE
s.nombre = :nombreSala";
```

Consultas - HQL: Hibernate Query Language

SQL Tradicional		HQL	
	`SELECT * FROM reunion`		`FROM Reunion`
	`SELECT * FROM reunion` `WHERE fecha < '2023-01-01'`		`FROM Reunion r` `WHERE r.fecha < :paramFecha`
	`SELECT * FROM reunion` `JOIN sala ON ...`		`FROM Reunion r` `WHERE r.sala.nombre = 'Azul'`

CriteriaBuilder

CriteriaBuilder es una interfaz que proporciona métodos para construir consultas de tipo CriteriaQuery. Estas consultas son más legibles y seguras, ya que están **basadas en clases y métodos** en lugar de cadenas de texto como HQL o SQL.

1. Obtener objeto CriteriaBuilder

   ```
   Session sesion = HibernateUtil.getFactoriaSession().openSession();
   CriteriaBuilder cb = sesion.getCriteriaBuilder();
   ```

2. Definimos la clase del tipo que devolvemos:

   ```
   CriteriaQuery<Reunion> query = cb.createQuery(Reunion.class);
   ```

   En este caso devolvemos una lista de objetos Reunion

3. Definimos la entidad base sobre la que hacemos la consulta: Root

   ```
   Root<Reunion> root = query.from(Reunion.class);
   ```

CriteriaBuilder

4. Hacemos la query más sencilla

query.select(root);
List<Reunion> reuniones =
sesion.createQuery(query).getResultList();

```
public List<Reunion> getReunionesCB()
	{
		Session sesion = HibernateUtil.getFactoriaSession().openSession();
		CriteriaBuilder cb = sesion.getCriteriaBuilder();
		CriteriaQuery<Reunion> query = cb.createQuery(Reunion.class);
		Root<Reunion> root = query.from(Reunion.class);
		query.select(root);
		List<Reunion> reuniones = sesion.createQuery(query).getResultList();
		return reuniones;
	}
//Equivalente a SELECT  * FROM Reunion
```

CriteriaBuilder

Consultas con filtros WHERE

A partir de la consulta sencilla:

```
query.select(root);
```

Añadiremos condiciones concatenando la llamada a where.

```
public List<Reunion> getReunionesConAsunto(String asunto)
{
    Session sesion = HibernateUtil.getFactoriaSession().openSession();
    CriteriaBuilder cb = sesion.getCriteriaBuilder();
    CriteriaQuery<Reunion> query = cb.createQuery(Reunion.class);
    Root<Reunion> root = query.from(Reunion.class);
    query.select(root);
    query.where(cb.equal(root.get("asunto"), asunto));
    List<Reunion> reuniones = sesion.createQuery(query).getResultList();
    return reuniones;     }
```

CriteriaBuilder

CriteriaBuilder: Consultas Programáticas

Consultas seguras (Type-safe) sin Strings propensos a errores.

```
CriteriaBuilder cb =
session.getCriteriaBuilder();
```

↓

```
CriteriaQuery<Reunion> q =
cb.createQuery(Reunion.class);
```

↓ El FROM

```
Root<Reunion> root =
q.from(Reunion.class);
```

↓ La Lógica

```
q.select(root).where(cb.equal(root.get('asunto'), 'Daily'));
```

El **compilador detecta errores** de nombres de campos inmediatamente.

CriteriaBuilder

Consultas con ORDER BY

A partir de la consulta sencilla:

```
query.orderBy(cb.asc(root.get("campo1")),
              cb.desc(root.get("campo2"))      );
```

Añadiremos condiciones concatenando la llamada a where.

```
public List<Reunion> getAllReunionesOrdenadasPorFechaYAsunto()
{
    Session sesion = HibernateUtil.getFactoriaSession().openSession();
    CriteriaBuilder cb = sesion.getCriteriaBuilder();
    CriteriaQuery<Reunion> query = cb.createQuery(Reunion.class);
    Root<Reunion> root = query.from(Reunion.class);
    query.select(root);
    query.orderBy(cb.asc(root.get("fecha")),
                  cb.desc(root.get("asunto"))
    );
    List<Reunion> reuniones = sesion.createQuery(query).getResultList();
    return reuniones;        }
```

CriteriaBuilder

Consultas multiselect

Son consultas que devuelven varios campos ya sean varias columnas o columnas calculadas (count, avg, sum)

Para ello usamos query.multiselect(campo1, expr2, …);

Cada expresión puede ser:

- Un atributo de la entidad → root.get("nombreAtributo")
- Un campo calculado → cb.count
- Una combinación o transformación: cb.concat, cb.upper

Importante: el resultado es una tabla de Object:

```
CriteriaQuery<Object[]> query = cb.createQuery(Object[].class);
```

CriteriaBuilder

Consultas multiselect

```
CriteriaQuery<Object[]> query = cb.createQuery(Object[].class);
Root<Reunion> root = query.from(Reunion.class);
query.multiselect(
                root.get("sala").get("nombre"),
                cb.count(root));
query.groupBy(root.get("sala").get("nombre"));
List<Object[]> resultados = sesion.createQuery(query).getResultList();
  for (Object[] fila : resultados) {
        System.out.println("Sala: " + fila[0] + ", Nº reuiones: " + fila[1]);
    }
```

CriteriaBuilder

Consultas multiselect

```
public void muestraInformacionReunion()
    {
        Session sesion = HibernateUtil.getFactoriaSession().openSession();
        CriteriaBuilder cb = sesion.getCriteriaBuilder();

        CriteriaQuery<Object[]> query = cb.createQuery(Object[].class);
        Root<Reunion> root = query.from(Reunion.class);
        query.multiselect(
                root.get("asunto"),
                root.get("idReunion"),
                root.get("sala").get("nombre"));

    List<Object[]> resultados = sesion.createQuery(query).getResultList();
    for (Object[] fila : resultados) {
        System.out.println("Asunto: " + fila[0] + ", id: " + fila[1]+ ", Sala: " + fila[2]);
    }
    }
```

CriteriaBuilder

Consultas con GROUP BY (sum, count, avg)

Esto es útil cuando necesitas agrupar resultados por una o más columnas y aplicar funciones de agregación como COUNT, SUM, AVG, etc.

```
query.groupBy(root.get("campo"));
query.having(cb.gt(campo2,  condicion));
```

Por ejemplo: Quiero contar las reuniones que ocurren en cada sala:

```
Root<Reunion> root = query.from(Reunion.class);
query.select(root);
 query.groupBy(root.get("sala").get("nombre"));
```

CriteriaBuilder

Consultas de actualización masivo: UPDATE

Trabajando con objetos podemos cambiar el valor de un atributo de un objeto y ese cambio posteriormente se persistirá en la base de datos.

En este caso quiero lanzar actualizaciones masivas de objetos.

Hasta ahora hemos hecho consultas con select o multiselect, ahora usaremos update. Usaremos la clase CriteriaUpdate

```
CriteriaUpdate<Clase> update = cb.createCriteriaUpdate(Clase.class);

update.set(campo1AActualizar, valor);

update.where(condición);
```

Importante para que tengan efecto en la BBDD debo iniciar y comitear una transacción

CriteriaBuilder

Consultas con DELETE

Borrar reuniones por tema

```
// 1. Crear CriteriaDelete para la entidad Reunion
CriteriaDelete<Reunion> delete = cb.createCriteriaDelete(Reunion.class);

 // 2. Definir la raíz
Root<Reunion> root = delete.from(Reunion.class);

// 3. Añadir condición (WHERE tema = 'Planificación')
delete.where(cb.equal(root.get("tema"), "Planificación"));

// 4. Ejecutar
em.getTransaction().begin();
int filas = em.createQuery(delete).executeUpdate();
em.getTransaction().commit();
System.out.println("Filas borradas: " + filas);
```


¿Dudas?