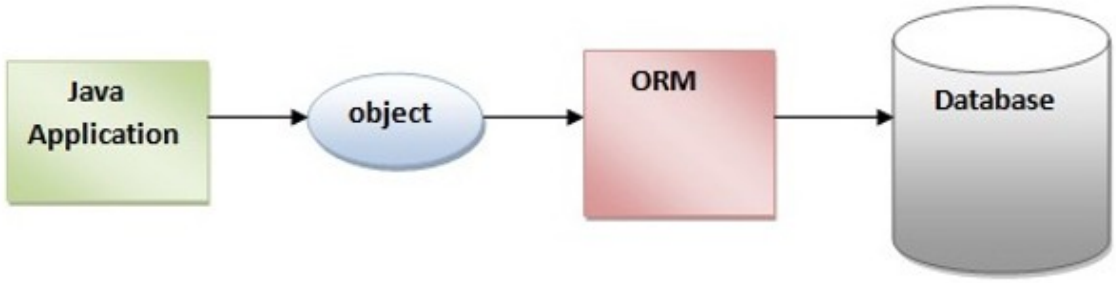
UNIDAD 3. Acceso a datos con herramientas ORM (Hibernate y Java)

Introducción y objetivos

Este tema trata del acceso a datos desde aplicaciones Java utilizando herramientas ORM (Object-Relational Mapping), con especial atención a **Hibernate**. Presenta conceptos teóricos, configuración práctica, ejemplos de mapeo (XML y anotaciones), manejo de sesiones y transacciones, y buenas prácticas para consultas seguras.

Tendremos:

- Entradas: clases Java (POJO), esquema relacional (BD), archivos de configuración (Maven/Gradle, `hibernate.cfg.xml` o `persistence.xml`).
- Salidas: objetos persistidos en la base de datos; consultas que devuelven objetos/tuplas; registros en BD modificados correctamente.
- Errores manejados: fallos de conexión, errores de mapeo, violaciones de integridad, transacciones abortadas.

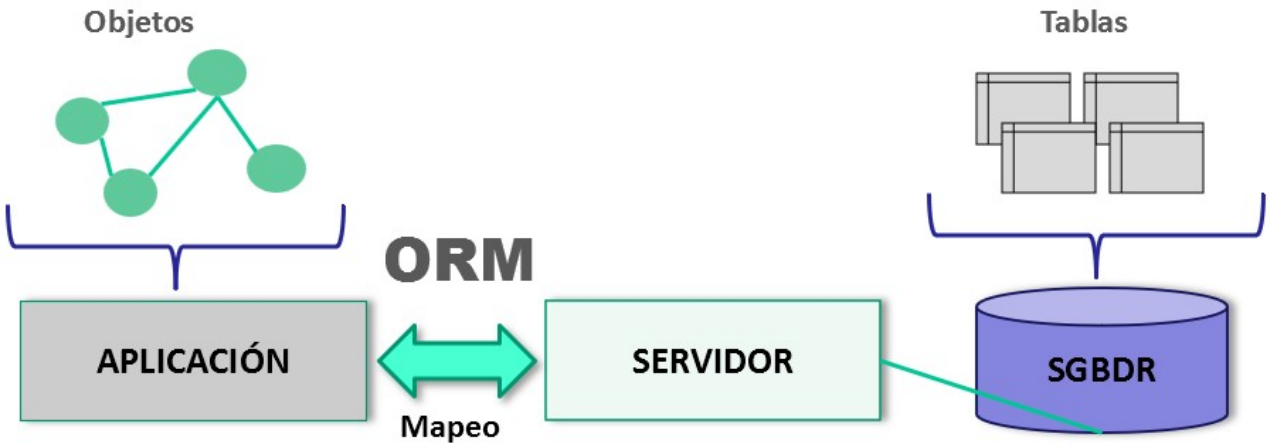


Objetivos:

- Comprender el mapeo objeto-relacional y las responsabilidades de un ORM.
- Instalar y configurar Hibernate en un proyecto Java.
- Definir mapeos con XML y con anotaciones.
- Implementar operaciones CRUD, consultas y transacciones de forma segura.
- Demostrar prácticas que eviten inyección SQL.

1. Concepto de mapeo objeto-relacional (O-R)

El mapeo O-R es la técnica que relaciona conceptos de la programación orientada a objetos (clases, atributos, asociaciones) con el modelo relacional (tablas, columnas, claves foráneas), usando un motor de persistencia. Un `ORM` automatiza la transformación entre objetos en memoria y filas en la base de datos, gestionando SQL, relaciones, y el ciclo de vida de las entidades.



2. Características de las herramientas ORM

- Las herramientas ORM permiten crear una capa de acceso a datos orientada a objetos.
- El programador interactúa con esta capa de acceso persistiendo y consultando objetos.
- Las herramientas ORM soportan un lenguaje de consultas OO propio e independiente de la BD, lo que permite migrar de una BD a otra sin tocar nuestro código, solo cambiando alguna línea en algún fichero de configuración.

Ventajas

- Reducir tiempo desarrollo software
- Abstracción de la base de datos
- Reutilización
- Permiten la producción de mejor código
- Son independientes del SGBD
- Usan un lenguaje propio para consultas

Inconvenientes

Las aplicaciones son más lentas debido a que las consultas a la BD:

- Usan primero el lenguaje propio de la herramienta para generar las instrucciones de consulta.
- Se transforman las instrucciones del lenguaje propio al lenguaje SQL.
- Se mapean los objetos usados en el lenguaje propio a las columnas y tablas de la base de datos.
- En la respuesta del servidor del SGBD, se transforman los resultados en objetos.

3. Herramientas ORM más utilizadas

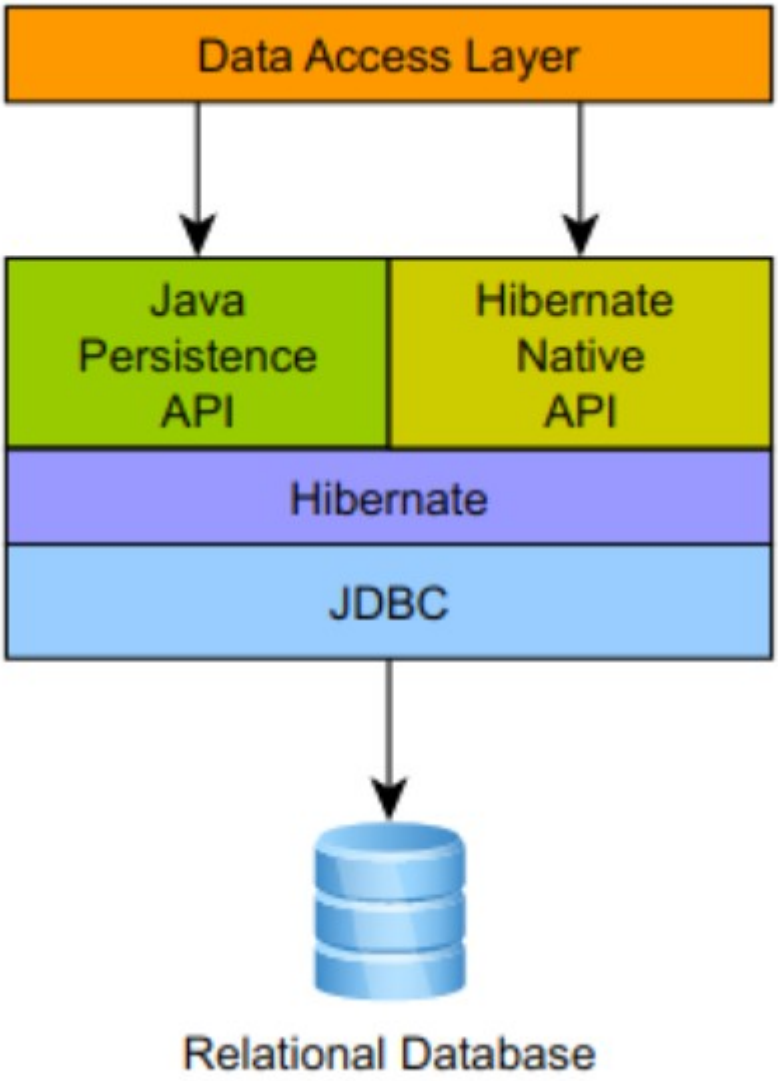
- Hibernate (muy usado en el ecosistema Java, implementación de JPA y ampliaciones propias).
- EclipseLink (implementación de JPA).
- OpenJPA.
- MyBatis (no es un ORM completo; es un mapper SQL semiautomático, útil cuando se necesita control sobre SQL).

En este tema nos centramos en `Hibernate` (y en `JPA` cuando sea relevante).

- **Hibernate** es la herramienta ORM más popular para Java. Se trata de un *framework* para Java.
- Es de código abierto. También tiene una distribución para `.NET`.
- Con **Hibernate** no se usará SQL para acceder a los datos sino que el **propio motor de Hibernate** construirá esas consultas por nosotros.
- Tiene un lenguaje HQL para acceder a datos mediante POO.

Hibernate y JPA

- **JPA (Java Persistence API)** es una especificación mientras que Hibernate es un framework.
- **JPA** es un documento en el cual se especifican los principios básicos de gestión de la capa de persistencia en el mundo de Java EE
- **Hibernate** es un framework que gestiona la capa de persistencia a través de ficheros xml o a través de anotaciones.
- **Hibernate implementa JPA** (además de otras especificaciones)
- **Hibernate** implementa como parte de su código la especificación de **JPA**
- Puede usarse **Hibernate** para construir una capa de persistencia apoyándose en las definiciones y reglas de la especificación **JPA**, aunque no es obligatorio.
- **Hibernate** es mucho más grande que la especificación de **JPA** y añade más funcionalidades.



- Usaremos principalmente JPA en nuestras aplicaciones.
- Nos proporciona:
  - Utilidades para especificar cómo nuestros objetos se relacionan con la base de datos (a través de anotaciones).
  - La API para realizar operaciones CRUD: jakarta.persistence.EntityManager
  - Lenguaje para realizar consultas: JPQL

💻 Práctica guiada-Primer Proyecto Hibernate

4. Instalación de una herramienta ORM. Configuración.

Ejemplo mínimo (Maven): añadir dependencias en `pom.xml` (Hibernate + driver JDBC):

```
<!-- Dependencias mínimas (ejemplos según enfoque) -->
<!-- A) Hibernate nativo (Session/SessionFactory) -->
<dependency>
  <groupId>org.hibernate.orm</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>7.2.0.CR1</version>
</dependency>
```

```
<!-- Driver JDBC para MariaDB/MySQL -->
<dependency>
  <groupId>org.mariadb.jdbc</groupId>
  <artifactId>mariadb-java-client</artifactId>
  <version>3.5.6</version>
</dependency>

<!-- B) JPA (persistence.xml + EntityManager) -->
<dependency>
  <groupId>jakarta.persistence</groupId>
  <artifactId>jakarta.persistence-api</artifactId>
  <version>3.1.0</version>
</dependency>
<!-- Hibernate como proveedor JPA (hibernate-core incluye el soporte JPA en
versiones modernas) -->
<dependency>
  <groupId>org.hibernate.orm</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>7.2.0.CR1</version>
</dependency>
```

Configuración clásica Nativa con `hibernate.cfg.xml` (ubicación típica: `src/main/resources/hibernate.cfg.xml`):

```
<?xml version='1.0'?>
<hibernate-configuration>
  <session-factory>
    <property
name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/mi_bd?
useSSL=false</property>
    <property name="hibernate.connection.username">usuario</property>
    <property name="hibernate.connection.password">clave</property>
    <property
name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.hbm2ddl.auto">validate</property>
    <!-- mapping entries (si se usan ficheros .hbm.xml)
      Ejemplo: una entrada <mapping> por cada fichero .hbm.xml en el classpath,
por ejemplo:
      <mapping resource="ejemplo/enativo/persona.hbm.xml"/>
      <mapping resource="ejemplo/enativo/direccion.hbm.xml"/>
      <mapping resource="ejemplo/enativo/telefono.hbm.xml"/>

      Nota: el valor de "resource" es la ruta relativa al classpath / al paquete
dentro del JAR.
      En un proyecto Maven coloca los .hbm.xml en `src/main/resources`; por
ejemplo,
      si el fichero está en `src/main/resources/ejemplo/enativo/persona.hbm.xml`
la entrada será
      `ejemplo/enativo/persona.hbm.xml`.
    -->
```

```
  </session-factory>
</hibernate-configuration>
```

Configuración para proyectos modernos con JPA se suele usar `persistence.xml` o configuración a través de Spring Boot.

Configuración JPA con `persistence.xml`

```
<persistence xmlns="https://jakarta.ee/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd"
             version="3.0">
  <persistence-unit name="miPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <!-- Añadir una entrada <class> por cada entidad del modelo, por ejemplo:
      <class>com.ejemplo.model.Persona</class>
      <class>com.ejemplo.model.Direccion</class>

    -->
    <properties>
      <property name="jakarta.persistence.jdbc.driver"
value="org.mariadb.jdbc.Driver"/>
      <property name="jakarta.persistence.jdbc.url"
value="jdbc:mariadb://localhost:3306/mi_bd?useSSL=false"/>
      <property name="jakarta.persistence.jdbc.user" value="usuario"/>
      <property name="jakarta.persistence.jdbc.password" value="clave"/>
      <property name="hibernate.dialect"
value="org.hibernate.dialect.MariaDBDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="validate"/>
      <property name="hibernate.show_sql" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

Observaciones:

- El dialecto (`hibernate.dialect`) debe coincidir con el SGBD usado (MySQL, MariaDB, PostgreSQL, Oracle, SQL Server, H2, etc.). Es importante para que Hibernate genere SQL compatible.
- `hibernate.hbm2ddl.auto` controla la verificación y/o la generación automática del esquema de la base de datos a partir de los mapeos. Valores habituales:
  - `none` (o omitir la propiedad): no hace ninguna comprobación ni modificación del esquema.
  - `validate`: comprueba que las tablas/columnas existen y coinciden con los mapeos; no modifica la base de datos. Recomendado en entornos de producción para detectar discrepancias sin riesgo.
  - `update`: intenta actualizar el esquema existente aplicando cambios necesarios (crear tablas/columnas nuevas, alterar columnas en algunos casos). No elimina columnas ni siempre realiza alteraciones complejas; puede generar DDL inesperado según el dialecto. Úsalo sólo en desarrollo o con mucha precaución y siempre con copias de seguridad.

- `create`: crea el esquema desde cero en el arranque (normalmente elimina/reescribe tablas existentes). Útil para pruebas locales cuando se quiere partir de cero.
- `create-drop`: como `create`, pero además elimina el esquema al cerrar la sesión/SessionFactory. Muy útil en suites de tests donde se necesita un esquema limpio por ejecución.

Consejos:

- En producción usa `validate` (o omite la propiedad) y gestiona las migraciones con herramientas dedicadas (Flyway, Liquibase) o scripts SQL controlados.
- `update` facilita desarrollo pero puede generar cambios indeseados y no cubre todos los casos (ej.: renombrados complejos). No confiar en él para cambios de esquema críticos.
- `create`/`create-drop` son convenientes para pruebas automatizadas o demos, nunca para datos reales sin respaldos.
- Si ves errores de DDL (sintaxis) al usar `update` o `create`, revisa el `hibernate.dialect` y la versión del SGBD: un dialecto incorrecto puede producir sentencias incompatibles.

5. Estructura de un fichero de mapeo (HBM XML). Elementos y propiedades

Ejemplo de fichero HBM (`Persona.hbm.xml`):

```
<?xml version="1.0"?>
<hibernate-mapping>
  <class name="com.ejemplo.model.Persona" table="persona">
    <id name="id" column="id">
      <generator class="identity"/>
    </id>
    <property name="nombre" column="nombre"/>
    <property name="edad" column="edad"/>
    <many-to-one name="direccion" class="com.ejemplo.model.Direccion"
column="direccion_id"/>
    <set name="telefonos" cascade="all" inverse="true" lazy="true">
      <key column="persona_id"/>
      <one-to-many class="com.ejemplo.model.Telefono"/>
    </set>
  </class>
</hibernate-mapping>
```

Elementos clave:

- `<class>`: mapea una clase a una tabla.
- `<id>`: define la clave primaria y su generador.
- `<property>`: mapea un campo simple.
- `<many-to-one>`: referencia desde la entidad actual a otra entidad (columna FK en la tabla actual). Representa multiplicidad muchos → uno (por ejemplo, `Persona` tiene una `Direccion`).
- `<one-to-many>`: colección en la entidad "uno" que agrupa a muchas instancias de la entidad relacionada. En la BD la tabla "muchos" suele contener la FK. Suele mapearse en HBM con una colección (`<set>`, `<bag>`) y la clave en la tabla opuesta.

- `<one-to-one>`: relación 1 → 1 entre dos tablas; puede implementarse mediante una FK única o una primary-key compartida según el diseño.
- `<many-to-many>`: relación N ↔ N que se materializa con una tabla intermedia (join table). En HBM se usa una colección (`<set>`/`<bag>`) junto con `<many-to-many>` que indica la tabla/columnas de unión.
- `<set>`: colección sin duplicados (no ordenada). Útil cuando no se quieren duplicados en la colección (por ejemplo, telefonos únicos por persona).
- `<bag>`: colección no ordenada que permite duplicados. Es la opción por defecto si no se necesita orden ni índice.
- `<list>`: colección ordenada que mantiene un índice (columna de posición) en la BD.
- `<map>`: colección clave → valor mapeada a la BD, donde la clave se guarda en una columna separada.

Notas sobre uso práctico:

- En relaciones bidireccionales hay que decidir el lado `inverse`/`mappedBy` que mantiene la FK para evitar inconsistencias.
- `cascade` controla si las operaciones (persist, remove...) se propagan a las entidades relacionadas.
- `fetch` (LAZY/EAGER) define cuándo se cargan las asociaciones; `LAZY` es la opción recomendada por defecto.
- `cascade`, `lazy`, `fetch`: controlan comportamiento de cascada y carga.

Nota sobre HBM vs JPA:

El formato `*.hbm.xml` es el mapeo nativo de Hibernate: está pensado para usarse con la configuración/arranque nativa (por ejemplo `hibernate.cfg.xml` y `org.hibernate.cfg.Configuration`). Hibernate como proveedor JPA puede cargar ficheros HBM si se los añades explícitamente, pero esto no forma parte de la especificación JPA y por tanto no es totalmente portable a otros proveedores.

Alternativas portables y recomendadas:

- Anotaciones JPA (`@Entity`, `@ManyToOne`, ...) — la opción más utilizada hoy en día y portable entre proveedores.
- `orm.xml` (mapeo JPA en XML) — estándar JPA, referenciado desde `persistence.xml` con `<mapping-file>...</mapping-file>`.

Cuándo usar cada cosa:

- `*.hbm.xml`: utilízalo sólo si necesitas alguna funcionalidad concreta de Hibernate o trabajas con código/ejemplos legacy que ya usan HBM.
- Anotaciones JPA / `orm.xml`: usa estas opciones en proyectos nuevos para mantener portabilidad y sencillez.

Ejemplos breves:

- Referenciar un HBM en `hibernate.cfg.xml` (nativo Hibernate):
  `<mapping resource="ejemplo/enativo/persona.hbm.xml"/>`
- Referenciar `orm.xml` en `persistence.xml` (JPA):
  `<mapping-file>META-INF/orm.xml</mapping-file>`

6. Mapeo basado en anotaciones (JPA/Hibernate)

Las anotaciones JPA son el enfoque más usado hoy en día. Ejemplo de clase persistente:

```
import jakarta.persistence.*;
import java.util.Set;

@Entity
@Table(name = "persona")
public class Persona {

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

```
// EJEMPLO equivalente con JPA (EntityManager)
EntityManagerFactory emf = Persistence.createEntityManagerFactory("miPU");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = null;
try {
  tx = em.getTransaction();
  tx.begin();

  // operaciones en la transacción
  Persona p = new Persona();
  p.setNombre("María");
  em.persist(p); // persist -> pasa a estado managed

  // ejemplo: buscar y actualizar
  Persona otra = em.find(Persona.class, 1L);
  if (otra != null) {
    otra.setEdad(40);
    // no es necesario llamar a merge si 'otra' es managed
  }

  tx.commit();
} catch (RuntimeException e) {
  if (tx != null && tx.isActive()) tx.rollback();
  throw e;
} finally {
  em.close();
  emf.close();
}
```

💻 Hoja02 (ejercicio 1)

Anotaciones comunes (qué hacen):

- `@Entity` — marca la clase como entidad persistente que será gestionada por el proveedor JPA/Hibernate.
- `@Table(name = "...")` — opcional; especifica el nombre de la tabla y parámetros de esquema/índices si se desea.

- `@Id` — indica el atributo que representa la clave primaria de la entidad.
- `@GeneratedValue(strategy = ...)` — define la estrategia de generación de la PK (IDENTITY, SEQUENCE, TABLE, AUTO).
  Estrategias:
    - `IDENTITY`: la base de datos genera el valor automáticamente (auto-increment / identity column). Es simple y ampliamente usado en MySQL/MariaDB. Limitación: impide algunos optimizaciones de Hibernate (batch inserts) porque el id se conoce sólo tras el insert.
    - `SEQUENCE`: usa un objeto SEQUENCE del SGBD (PostgreSQL, Oracle). Más eficiente para inserciones en lote; Hibernate puede reservar/obtener valores de la secuencia sin hacer insert inmediato. Requiere que el SGBD soporte secuencias.
    - `TABLE`: simula un generador usando una tabla auxiliar que guarda el último valor (portable a SGBD que no soportan secuencias). Funciona en cualquier BD pero es menos eficiente y puede tener contención en alto volumen.
    - `AUTO`: deja que el proveedor elija la estrategia apropiada según el dialecto/BD; por ejemplo, seleccionará `SEQUENCE` si la BD lo soporta, o `IDENTITY` en otros casos.

  Consejos prácticos: usar `IDENTITY` para compatibilidad rápida con MySQL/MariaDB; preferir `SEQUENCE` en DB que soporten secuencias si necesitas rendimiento en inserciones masivas; evitar `TABLE` salvo por portabilidad extrema; `AUTO` es cómodo pero menos predecible.
- `@Column(name = "...", nullable = ..., length = ...)` — mapea un campo a una columna y permite configurar nombre, nulabilidad, longitud y otras propiedades de la columna.
    - `name`: permite modificar el nombre que tendrá la columna mapeada (si no se usa, será el nombre del atributo).
    - `length`: nos permite definir el número de caracteres de la columna.
    - `nullable`: nos permite indicar si la columna mapeada puede o no almacenar valores nulos.
    - `columnDefinition`: para especificar cual será el tipo de dato en la columna mapeada.
    - `insertable`, `updatable`: Cuando se ponen a false, la columna no es considerada para esas operaciones. Útil, por ejemplo, para una columna fecha en la que se asigna en BD por defecto la fecha actual, al hacer una inserción.
- `@Embedded` y `@Embeddable`- tipos embebidos: Imaginemos que partimos de una tabla de una base de datos en la que un alumno, además del id, nombre y fecha de nacimiento tenga una calle y un número de vivienda.En **OO** podríamos tener una clase *Alumno* y una clase *Direccion*.

Podemos definir nuestro modelo de clases de tal forma que mapeemos ambas clases contra la misma tabla alumnos utilizando las anotaciones de JPA `@Embedded` y `@Embeddable`.

- *Direccion* sea una clase **Embeddable**. No es una entidad, no es persistente por sí misma.
- *Alumno* añadimos un atributo marcado como **Embedded**
  Ejemplo:

```
@Embeddable
public class Direccion {
  private String calle;
  private String numero;
```

```
  // getters y setters
}
```

```
@Entity
public class Alumno {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;
  private String nombre;
  private LocalDate fechaNacimiento;

  @Embedded
  private Direccion direccion;
  // getters y setters
}
```

 Hoja02 (ejercicio 2)

 Ejemplos de asociaciones

- Relaciones (anotaciones de asociación):
  Las entidades pueden estar relacionadas entre ellas.
  A nivel de base de datos las asociaciones se representan mediante claves externas o FOREIGN KEY.
  Con JPA podemos modelarlas estas asociaciones. Pueden ser unidireccionales o bidireccionales:

  - `@ManyToOne` — (muchos → uno); la entidad actual tiene una FK hacia la entidad referenciada. Es el lado propietario de la relación.
  - `@OneToMany` — (uno → muchos); normalmente mapea una colección y suele declararse en el lado inverso con `mappedBy` que apunta al campo propietario.
  - `@OneToOne` — (uno → uno); puede implementarse con una FK única o compartiendo la PK entre tablas; usar `@JoinColumn` para indicar la columna de unión.
  - `@ManyToMany` — (muchos ↔ muchos); se materializa mediante una tabla intermedia (configurable con `@JoinTable`).

- `@JoinColumn(name = "columna_fk")` — especifica la columna de la tabla que almacena la FK para la relación (se usa en el lado propietario).

- `mappedBy` (atributo en `@OneToMany`/`@OneToOne`/`@ManyToMany`) — indica el nombre del atributo en el lado propietario; el lado que tiene `mappedBy` no mantiene la FK.

- `cascade` (ej.: `cascade = CascadeType.PERSIST`) — controla qué operaciones (PERSIST, MERGE, REMOVE, REFRESH, DETACH, ALL) se propagan a las entidades relacionadas.

  Descripción de las operaciones de cascade:

  - `PERSIST` — Al persistir (guardar) la entidad propietaria, también se persistirán las entidades relacionadas que estén en estado transitorio. Útil para insertar un grafo completo de objetos sin llamar a `persist` en cada uno. Nota: si la entidad relacionada ya existe en BD y está detached/managed, se debe tener cuidado para no crear duplicados.

    - `MERGE` — Fusiona el estado de una entidad (normalmente detached) en el contexto de persistencia; devuelve una instancia managed con los cambios aplicados. No modifica la instancia original si ésta está detached: el `EntityManager` crea o localiza una entidad managed y copia el estado. Usar `merge` para reattach entidades fuera del contexto.
    - `REMOVE` — Marca la entidad (y, si se propaga, las relacionadas) para eliminación; la eliminación se realiza en el `flush`/`commit`. Solo funciona sobre entidades managed; si una entidad está detached primero debe reattach o ser recuperada antes de eliminarla.
    - `REFRESH` — Sobrescribe el estado actual de la entidad con los datos actuales de la base de datos. Descarta los cambios no sincronizados en memoria para esa entidad. Si la fila ha sido borrada en BD, `refresh` puede lanzar una excepción (`EntityNotFoundException` o similar según el proveedor).
    - `DETACH` — Desasocia la entidad del contexto de persistencia; tras `detach` los cambios en el objeto no se sincronizan con la BD. Propagar `DETACH` permite que al desactivar la entidad propietaria se desactiven también las entidades relacionadas.
    - `ALL` — Es un atajo que equivale a `{PERSIST, MERGE, REMOVE, REFRESH, DETACH}`; aplica todas las operaciones anteriores. Usar `ALL` es cómodo pero exige comprender el efecto sobre el grafo de entidades (por ejemplo, `REMOVE` en cascada puede borrar filas relacionadas inesperadamente).

  Notas prácticas y casos límite:

    - `persist` con relaciones: si una entidad relacionada es transitoria, necesita `PERSIST` en cascade o llamadas explícitas a `persist` para evitar errores de clave foránea.
    - `merge` devuelve la instancia managed; no asumas que la referencia pasada se vuelve managed. Ejemplo: `Persona managed = em.merge(detachedPersona);` — usa `managed` para seguir operando.
    - `remove` requiere que la entidad sea managed; si tienes un detached entity, recupera la entidad con `find` o `merge` antes de `remove`.
    - `refresh` descarta cambios en memoria; no usarlo para “confirmar” cambios locales.
    - Evitar `CascadeType.ALL` si no quieres propagar `REMOVE` o `DETACH` automáticamente; es preferible enumerar solo los tipos necesarios (`PERSIST`, `MERGE`, etc.).

- `fetch` (LAZY o EAGER) — determina la estrategia de carga de la asociación; `LAZY` retrasa la carga hasta acceso y es recomendado por defecto para colecciones.

- `orphanRemoval = true` — en colecciones, indica que entidades huérfanas (ya no referenciadas) deben eliminarse automáticamente.

Estas anotaciones pueden combinarse para definir comportamiento completo de la relación (propietario/inverso, cascada y estrategia de carga). En general, elegir `LAZY` para colecciones y controlar cascadas explícitamente evita sorpresas en producción.

Cuando mapeamos una entidad, todos sus atributos son considerados persistentes.Excepto los declarados **static y/o final** y los anotados como transitorios con `@Transient`.

Todo **atributo persistente** se mapea a una columna en una tabla de la base de datos.

Hibernate escoge la mejor correspondencia de tipos de datos en el SGBD para los tipos Java que hayamos usado en las entidades.

💻 Hoja02(resto de ejercicios)

7. Clases persistentes

Se denomina **clase persistente** a una clase Java cuyo estado puede guardarse/recuperarse de la base de datos por el proveedor de persistencia (Hibernate). Se marca típicamente con `@Entity` (o con un `<class>` en HBM).

Recomendaciones:

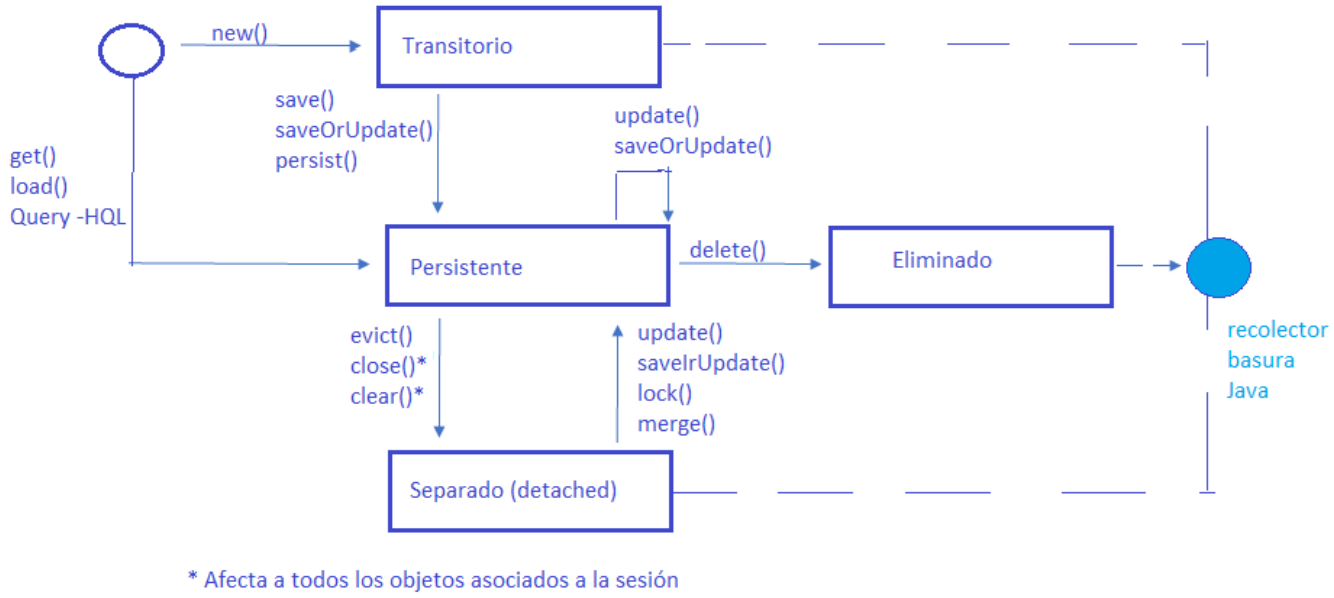
- Tener un constructor sin argumentos (público o protegido): Hibernate lo crea por reflexión.
- Usar un tipo envolvente para la PK (por ejemplo `Long id`) y anotarla con `@Id` y `@GeneratedValue` cuando se genere en BD; evita tipos primitivos para la PK.
- Mantener un único modo de acceso: anotar campos (field access) o getters (property access), no mezclar ambos.
- No declarar la clase ni los métodos importantes como `final` (Hibernate puede crear proxies que extienden la clase).
- `equals` / `hashCode`: implementar basándolos en la PK cuando ésta ya está asignada; usar `instanceof` (no `getClass()`) para soportar proxies y evitar usar campos mutables.
- Evitar lógica pesada en getters, `toString()`, `equals()` o `hashCode()` (pueden disparar loads perezosos o consultas inesperadas).
- Preferir `LAZY` en colecciones y controlar la carga con `JOIN FETCH` cuando necesites datos relacionados.
- Controlar explícitamente `cascade` (no usar `CascadeType.ALL` sin entender el impacto) y usar `orphanRemoval` sólo cuando corresponda.
- Para capas de presentación o API, usar DTOs en lugar de exponer entidades gestionadas directamente.
- La serialización de las clases java, es necesario si la clase esta compuesta por (`@IdClass` y `@EmbeddedId`) se va a usar en contextos donde se requiere serialización como en sesiones distribuidas o almacenamiento en caché distribuido. Sin embargo, es una buena práctica que las clases que representan claves compuestas implementen la `interfaz Serializable` porque:
    - Hibernate lo recomienda para asegurar compatibilidad con todas sus funciones internas
    - Algunas implementaciones de JPA lo requieren para el correcto funcionamiento de la persistencia y la replicación.

8. Sesiones; estados de un objeto

Hibernate nos ha permitido realizar la correspondencia ORM para clases de Java y para las relaciones entre ellas. Nos permite crear objetos transitorios que se han almacenado en la BD como objetos persistentes.

La clase para la que establecemos una correspondencia mediante Hibernate se llama clase persistente y podemos crear objetos persistentes instanciando la clase.

Ciclo de vida de los objetos persistentes:



Este ciclo de vida muestra los estados en los que puede estar los objetos persistentes, las operaciones que permiten recuperarlos, modificarlos y grabar de nuevo estas modificaciones en la BD.
Una sesión (interfaz `org.hibernate.Session`) es esencial en Hibernate porque se construye sobre una conexión a la base de datos.

Una **sesión**, junto con un gestor de entidades asociado, constituye un contexto de persistencia. El gestor de entidades (interfaz `jakarta.persistence.EntityManager`) lleva el control de los cambios que se realizan sobre objetos persistentes.
La interfaz `Session` es de la `API especifica de Hibernate`.

Para garantizar la portabilidad y compatibilidad con otros sistemas JPA, es preferible utilizar la interfaz `EntityManager` pertenece a la de JPA. Es posible obtener una `EntityManager` a partir de una `Session` y al revés porque hay un puente entre ambas.

Los cambios que hacemos sobre los objetos quedan reflejados en la BD porque están asociados a un contexto de persistencia.

Estados de una instancia:

- Transient (transitorio): objeto nuevo, no asociado a sesión y no existe en BD.
- Persistent (persistente): asociado a una sesión; cualquier cambio se sincroniza con la BD en flush/commit.
- Detached (desconectado): estuvo persistente, pero la sesión se cerró; cambios no se sincronizan hasta reattach.
- Removed (eliminado): marcado para borrado en la sesión.

Sesión (Hibernate `Session`) y `EntityManager` en JPA: representan la unidad de trabajo y el contexto de persistencia.

Creación típica de `SessionFactory` (Hibernate nativo):

```
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import org.hibernate.Session;
```

```
Configuration configuration = new Configuration().configure(); // lee
hibernate.cfg.xml
SessionFactory sessionFactory = configuration.buildSessionFactory();
Session session = sessionFactory.openSession();
```

```
// Ejemplo mínimo de uso de JPA con EntityManager
// (suponiendo persistence.xml con persistence-unit "miPU")
EntityManagerFactory emf = Persistence.createEntityManagerFactory("miPU");
EntityManager em = emf.createEntityManager();
```

9. Carga, almacenamiento y modificación de objetos (CRUD)

Operaciones básicas con `Session`:

- **Crear / persistir**:

```
try (Session session = sessionFactory.openSession()) {
  Transaction tx = session.beginTransaction();
  Persona p = new Persona();
  p.setNombre("Ana");
  session.persist(p); // ahora p pasa a estado persistent
  tx.commit();
}
```

- **Leer**:

```
try (Session session = sessionFactory.openSession()) {
  Persona p = session.get(Persona.class, 1L); // devuelve null si no existe
}
```

- **Actualizar**:

```
try (Session session = sessionFactory.openSession()) {
  Transaction tx = session.beginTransaction();
  Persona p = session.get(Persona.class, 1L);
  p.setEdad(30); // si p es persistent, el cambio se detecta automáticamente
  tx.commit();
}
```

- **Borrar**:

```
try (Session session = sessionFactory.openSession()) {
  Transaction tx = session.beginTransaction();
```

```
    Persona p = session.get(Persona.class, id);
    if (p != null) session.remove(p);
    tx.commit();
  }
```

Operaciones básicas con JPA con EntityManager

```
// Ejemplo mínimo de uso de JPA con EntityManager
// (suponiendo persistence.xml con persistence-unit "miPU" y entidad Persona)
EntityManagerFactory emf = Persistence.createEntityManagerFactory("miPU");
EntityManager em = emf.createEntityManager();
try {
  // INSERT (persist)
  em.getTransaction().begin();
  Persona nuevo = new Persona();
  nuevo.setNombre("Luis");
  nuevo.setEdad(35);
  em.persist(nuevo);
  em.getTransaction().commit();

  // READ (find)
  Persona encontrado = em.find(Persona.class, nuevo.getId());

  // UPDATE (merge) — útil para objetos detached
  em.getTransaction().begin();
  if (encontrado != null) {
    encontrado.setEdad(36);
    em.merge(encontrado);
  }
  em.getTransaction().commit();

  // DELETE (remove)
  em.getTransaction().begin();
  if (encontrado != null) {
    em.remove(encontrado);
  }
  em.getTransaction().commit();

} finally {
  em.close();
  emf.close();
}
```

10. Consultas: HQL/JPQL

Consultas HQL

La interfaz Query permite hacer consultas **HQL(Hibernate Query Language)** usando la API de Hibernate. Vamos a ver como se puede obtener una sentencia de **Query** a partir de una instancia de **Session** mediante el

método `createQuery(String consulta)` devuelve una instancia de la interfaz `org.hibernate.query.Query`

Una Query puede tener parámetros que se pueden identificar por nombre o por posición.

```
import org.hibernate.query.Query;

// lista de objetos getResultList()
try (Session session = sessionFactory.openSession()) {
  Query<Persona> query = session.createQuery("from Persona p where p.edad >
:edad", Persona.class);
  query.setParameter("edad", 18);
  List<Persona> lista = query.getResultList();
// una consulta únicamente devuelve cero o un resultado uniqueResult()
  Query<Persona> query2 = session.createQuery("from Persona p where p.id > :id",
Persona.class);
  query2.setParameter("id", 1L);
  Persona persona = query2.uniqueResult();
}
```

 Hoja03

Consultas JPQL

JPQL nos permite hacer consultas en base a muchos criterios. También permite obtener más de un valor por consulta
La consulta JPQL más básica tiene la siguiente estructura:

```
SELECT alias FROM NombreEntidad alias WHERE condición
```

Estas otras consultas son equivalentes a la anterior(aunque por claridad es mejor usar la anterior)

```
FROM NombreEntidad alias WHERE condición
FROM NombreEntidad
```

En JPA tenemos dos interfaces para crear consultas. Son **Query** y **TypedQuery** del paquete `jakarta.persistence`
Podemos crear una consulta con el método `createQuery` del `EntityManager`.

```
String jpql=" Select p From Propietario p";
Query query=em.createQuery(jpql);
List<Propietario> listaPropietarios=query.getResultList();
```

Como se puede ver, para las consultas que devuelven un conjunto de datos usaremos el método `getResultList()`

El método devuelve un `List`. Esa última instrucción queda marcada con un **warning** indicando que se deberá chequear que lo devuelto por `getResultList` debería comprobarse que es una lista de propietarios. Si queremos quitar el warning lo podemos usar la anotación `@SuppressWarnings("unchecked")`

Además de otras ventajas, el uso de `TypedQuery` en lugar de `Query` evita que se produzcan los warnings por no chequear el tipo de resultado.
En la construcción de una `TypedQuery` ya se incluye el tipo de dato base de la lista que devuelve `getResultList`.

```
String jpql=" Select p From Propietario p";
TypedQuery<Propietario> typedQuery=em.createQuery(jpql, Propietario.class);
List<Propietario> listaPropietarios=typedQuery.getResultList();
```

Si una consulta devuelve un solo resultado, tendremos entonces que llamar a `getSingleResult()`.

```
String jpql=" Select p From Propietario p Where p.id=:id";
TypedQuery<Propietario> typedQuery=em.createQuery(jpql, Propietario.class);
typedQuery.setParameter("id", 1L);
Propietario propietario=typedQuery.getSingleResult();
```

Select en JPQL

Las palabras **SELECT** y **FROM** tienen el mismo significado que en el lenguaje SQL. Además, la **p** será un alias.

```
String jpql=" Select p.nombre,p.edad From Propietario p";
```

Filtrar los resultados
Para filtrar los resultados de una consulta JPQL se utiliza la cláusula **WHERE** y operadores relacionales, lógicos(**AND**, **OR**, **NOT**) y otros como **IN**, **BETWEEN**, **LIKE**

Ordenar los resultados
Para ordenar los resultados de una consulta JPQL se utiliza la cláusula **ORDER BY** seguida del nombre del atributo por el que se quiere ordenar y la palabra clave **ASC** o **DESC** para indicar si el orden es ascendente o descendente.

```
String jpql=" Select p From Propietario p WHERE p.edad > :edad ORDER BY p.nombre
ASC";
```

Uso de parámetros en consultas
Los parámetros en las consultas JPQL se pueden definir de dos formas:

- Parámetros nombrados: se indican con dos puntos seguidos del nombre del parámetro.
- Parámetros posicionales: se usan por indice.

```
// EJEMPLO: parámetros nombrados en JPQL
String jpql=" Select p From Propietario p WHERE p.edad > :edad AND p.nombre LIKE
:nombre";
TypedQuery<Propietario> typedQuery=em.createQuery(jpql, Propietario.class);
typedQuery.setParameter("edad", 18);
typedQuery.setParameter("nombre", "A%");
List<Propietario> listaPropietarios=typedQuery.getResultList();
```

```
// EJEMPLO: parámetros posicionales / por índice en JPQL
// En JPQL se usan ?1, ?2, ... y se asignan con setParameter(1, valor),
setParameter(2, valor)
String jpqlIdx = "SELECT p FROM Propietario p WHERE p.edad > ?1 AND p.nombre LIKE
?2";
TypedQuery<Propietario> qIdx = em.createQuery(jpqlIdx, Propietario.class);
qIdx.setParameter(1, 18);       // posición 1
qIdx.setParameter(2, "A%");    // posición 2
List<Propietario> listaIdx = qIdx.getResultList();
```

Agrupaciones y funciones de agregado

JPQL soporta las funciones de agregado comunes como **COUNT**, **SUM**, **AVG**, **MIN**, **MAX** y la cláusula **GROUP BY** para agrupar resultados.

```
String jpql=" Select p.edad, COUNT(p) From Propietario p GROUP BY p.edad HAVING
COUNT(p) > :minimo";
TypedQuery<Object[]> typedQuery=em.createQuery(jpql, Object[].class);
typedQuery.setParameter("minimo", 5L);
List<Object[]> resultados=typedQuery.getResultList();
for (Object[] fila : resultados) {
  Integer edad = (Integer) fila[0];
  Long cuenta = (Long) fila[1];
  System.out.println("Edad: " + edad + ", Cuenta: " + cuenta);
}
```

Joins en JPQL

JPQL soporta **JOINs** para navegar y filtrar en asociaciones entre entidades.
JPQL permite realizar los mismos JOIN que en SQL (**JOIN**, **INNER JOIN**, **LEFT JOIN**, **RIGHT JOIN**).

JOIN implícito

El desarrollador no especifica ninguna combinación. Si en las entidades hay asociaciones JPA, crea automáticamente un JOIN implícito

```
//obtener(sin repetir)propietarios de coches
String jpql=" Select DISTINCT c.propietario From Coche c";
TypedQuery<Propietario> typedQuery=em.createQuery(jpql, Propietario.class);
List<Propietario> listaPropietarios=typedQuery.getResultList();
```

JOIN explícito

```
//obtener propietarios de coches de una marca concreta
String jpql=" Select p From Propietario p JOIN p.coches c WHERE c.marca = :marca";
TypedQuery<Propietario> typedQuery=em.createQuery(jpql, Propietario.class);
typedQuery.setParameter("marca", "Toyota");
List<Propietario> listaPropietarios=typedQuery.getResultList();
```

JOIN Fetch

```
// Para obtener los propietarios junto con todos sus coches (evita N+1)
String jpqlFetch = "Select DISTINCT p From Propietario p JOIN FETCH p.coches WHERE
p.edad > :edad";
TypedQuery<Propietario> qFetch = em.createQuery(jpqlFetch, Propietario.class);
qFetch.setParameter("edad", 18);
List<Propietario> listaPropietarios = qFetch.getResultList();

// Recorrer la lista de propietarios y obtener los coches de cada uno
for (Propietario propietario : listaPropietarios) {
  // suponiendo que la relación en Propietario se llama 'coches' y tiene getter
getCoches()
  Collection<Coche> coches = propietario.getCoches();
  for (Coche coche : coches) {
    // ejemplo: procesar o imprimir información del coche
    System.out.println("Propietario: " + propietario.getNombre() + " - Coche: " +
coche.getMarca() + " " + coche.getModelo());
  }
}
```

Consultas de actualización

UPDATE

```
String jpqlUpdate = "UPDATE Propietario p SET p.edad = p.edad + 1 WHERE p.edad <
:edadLimite";
Query queryUpdate = em.createQuery(jpqlUpdate);
queryUpdate.setParameter("edadLimite", 18);
int filasAfectadas = queryUpdate.executeUpdate();
System.out.println("Filas actualizadas: " + filasAfectadas);
```

DELETE

```
String jpqlDelete = "DELETE FROM Propietario p WHERE p.edad > :edadLimite";
Query queryDelete = em.createQuery(jpqlDelete);
```

```
queryDelete.setParameter("edadLimite", 100);
int filasEliminadas = queryDelete.executeUpdate();
System.out.println("Filas eliminadas: " + filasEliminadas);
```

Named Queries

Para evitar consultas repetidas en distintas partes del código podemos usar **NamedQueries**. Son predefinidas usando la notación `@NamedQuery` o `@NamedQueries` en la definición de la entidad.

Este comportamiento estático las hace mas eficientes y por lo tanto ofrecen un mejor rendimiento. El alcance es el contexto de persistencia actual, por lo tanto, **el nombre de cada query será único bajo el mismo contexto de persistencia**

```
@Entity
@Table(name = "propietarios")
@NamedQueries({
  @NamedQuery(
    name = "Propietario.findByEdad",
    query = "SELECT p FROM Propietario p WHERE p.edad > :edad"
  ),
  @NamedQuery(
    name = "Propietario.findAll",
    query = "SELECT p FROM Propietario p"
  )
})
```

Uso de NamedQuery:

```
TypedQuery<Propietario> query = em.createNamedQuery("Propietario
.findByEdad", Propietario.class);
query.setParameter("edad", 18);
List<Propietario> resultados = query.getResultList();
```

Funciones en consultas JPQL

JPQL soporta varias funciones integradas que se pueden usar en las consultas para manipular datos. Algunas de las funciones más comunes son:

Tipo	Función
De fecha/hora actual	current_date(), current_time(), current_timestamp()
De extracción de fecha	day(fecha), month(fecha), year(fecha)
De extracción de tiempo	hour(time), minute(time), second(time)
De String	substring(), trim(), lower(), upper(), length(), locate(), concat()
Conversión a string	str(valor)

Tipo	Función
De agregación o resumen	count(..), avg(..), sum(..), max(..), min(..)
Sobre colecciones	SIZE(coleccion), INDEX(coleccion)

> Nota: las funciones y la sintaxis pueden variar ligeramente según la implementación JPA/Hibernate y la versión; usa las funciones estándar de JPQL cuando sea posible para mantener portabilidad.

Seguridad frente a inyección SQL:

- Nunca concatenar valores de entrada directamente en la cadena de consulta.
- Usar parámetros nombrados (`:param`) o posicionales y `setParameter`.
- Validar y sanear entradas (longitud, formato) en la capa de aplicación.
- Usar cuentas de BD con privilegios mínimos.

 Hoja04

11. Gestión de transacciones

Transacciones programáticas (ejemplo con Hibernate puro):

```
try (Session session = sessionFactory.openSession()) {
  Transaction tx = session.beginTransaction();
  try {
    // operaciones
    tx.commit();
  } catch (RuntimeException e) {
    if (tx != null && tx.isActive()) tx.rollback();
    throw e;
  }
}
```

Transacciones con JPA (EntityManager) usando Hibernate como proveedor

```
// Suponiendo un persistence-unit llamado "miPU" en persistence.xml
EntityManagerFactory emf = Persistence.createEntityManagerFactory("miPU");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = null;
try {
  tx = em.getTransaction();
  tx.begin();
  //operaciones
  tx.commit();
} catch (RuntimeException e) {
  if (tx != null && tx.isActive()) tx.rollback();
  throw e;
} finally {
  em.close();
  emf.close();
}
```

Buenas prácticas:

- Acortar la duración de las transacciones.
- Evitar operaciones de I/O pesadas dentro de la transacción.
- Manejar correctamente el rollback en excepciones recuperables y no recuperables.

12. Conclusión y referencias

Hibernate es una herramienta madura y potente para mapear modelos OO a bases de datos relacionales. Conocer su configuración, mapeos (XML y anotaciones), gestión de sesiones y transacciones, y técnicas para evitar inyección SQL permite desarrollar aplicaciones robustas y seguras.

Referencias rápidas:

- Documentación oficial Hibernate: https://hibernate.org/
- JPA specification: https://jakarta.ee/specifications/persistence/
