UTN FACULTAD REGIONAL CÓRDOBA
INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Backend de Aplicaciones
Apunte de Clases

Apunte 13 - ORM con JPA (Java Persistence API)

Introducción y fundamentos de ORM

El problema de la impedancia objeto–relacional

En la programación orientada a objetos (POO) trabajamos con **clases, objetos, herencia, polimorfismo y asociaciones**. En cambio, en las bases de datos relacionales usamos **tablas, filas, claves primarias, claves foráneas y joins**. Estos dos mundos no encajan de manera natural: a esto se lo conoce como el **problema de la impedancia objeto–relacional**.

Ejemplos de desajustes típicos:

- **Objetos vs. Tablas**: un objeto puede tener jerarquía y composición, mientras que una tabla es plana.
- **Identidad**: en Java la identidad es la referencia en memoria; en la base se define con claves primarias.
- **Relaciones**: en Java tenemos referencias y colecciones; en la base tenemos claves foráneas y tablas de unión.
- **Herencia**: no existe de forma directa en el modelo relacional.

Estrategias históricas

Antes de los ORM, las aplicaciones Java usaban directamente:

- **JDBC** con sentencias SQL escritas a mano.
- **DAOs (Data Access Objects)** que encapsulaban las consultas.

Esto funcionaba, pero resultaba repetitivo, propenso a errores y difícil de mantener en proyectos grandes.

El concepto de ORM

Un **ORM (Object Relational Mapping)** es una técnica que permite mapear clases y objetos de un lenguaje de programación a tablas y registros de una base de datos. De esta manera:

- Las entidades del dominio se representan como objetos Java.
- El ORM se encarga de traducir operaciones sobre los objetos en sentencias SQL.

Ventajas de un ORM

- **Productividad**: menos código SQL repetitivo.
- **Portabilidad**: cambio de motor de base de datos sin reescribir todas las consultas.
- **Mantenibilidad**: las entidades se describen en un único lugar y reflejan el modelo del dominio.
- **Integración**: se conecta naturalmente con frameworks de alto nivel (ej. Spring Data).

Riesgos y limitaciones

- **Sobrecarga de abstracción**: ocultar el SQL puede generar consultas ineficientes.
- **Consultas complejas**: no siempre son fáciles de expresar en JPQL/Criteria.
- **Tuning**: a veces es necesario optimizar con SQL nativo.

El estándar JPA y Hibernate

- **JPA (Jakarta Persistence API)**: especificación estándar para ORM en Java.
- **Hibernate**: la implementación más utilizada de JPA.

Con JPA definimos **interfaces y anotaciones estándar**; con Hibernate tenemos la implementación concreta que ejecuta las operaciones.

Características principales:

- **Entidades:** Las clases de tu modelo se anotan como entidades (@Entity), lo que permite a JPA saber qué clases deben ser mapeadas a tablas en la base de datos.
- **EntityManager:** El EntityManager es el punto de entrada para realizar operaciones CRUD en tus entidades.
- **JPQL:** JPA introduce un lenguaje de consulta llamado JPQL (Java Persistence Query Language) que se utiliza para realizar consultas de manera similar a SQL pero orientado a objetos.

JPA - Historia y Evolución

- **EJB 2.x / Entity Beans:** Antes de JPA, la persistencia se hacía con Entity Beans en EJB 2 .x, una solución compleja y difícil de manejar.
- **JPA 1.0 (2006):** Introducido con Java EE 5 (JSR 220), buscaba simplificar la persistencia.
- **JPA 2.0 (2009):** Parte de Java EE 6 (JSR 317), incorporó Criteria API, mejoras en el mapping.
- **JPA 2.1 (2013):** En Java EE 7 (JSR 338), agregó converters, stored procedures, entity graphs.
- **JPA 2.2 (2017):** En Java EE 8; añadió streaming, anotaciones repetibles y tipos de fecha/hora de Java 8.
- **Jakarta Persistence 3.1 (2022):** Renombrado como parte de Jakarta EE 10; incluye funciones JPQL nuevas y mejor soporte para UUID.
- **Jakarta Persistence 3.2 (2024):** Mejoras en la API y funcionalidades de JPQL.

Importancia de JPA en el Ecosistema Java

- **Independencia de proveedor:** Podemos cambiar entre Hibernate, EclipseLink, OpenJPA, etc., sin cambiar el código.
- **Desacoplamiento:** Separamos los objetos de dominio de la lógica de persistencia con anotaciones estándar.
- **Productividad:** Menos boilerplate, consultas legibles en JPQL.
- **Estándar de facto:** JPA (hoy Jakarta Persistence) es ampliamente aceptado en aplicaciones empresariales Java.

JPA Implementación y Uso

La especificación JPA se centra en 3 ejes fundamentales a la hora de formalizar el Mapeo Objeto Relacional en Java. Estos 3 ejes van a ser los 3 ejes fundamentales de estudio y a los que le vamos a tener que dedicar esfuerzo para comprender su funcionamiento y capacidades de control. Estas serán las capacidad que nos permitan modificar la Implementación subyacente para que responda de acuerdo con las decisiones de diseño que hemos tomado para el proyecto y se adecúe de la mejor manera a nuestro esquema funcional.

Estos 3 ejes fundamentales nos en órden de necesidad para la implementación:

- **Configuración** donde especificaremos la conexión a la base de datos, la implementación subyacente, el esquema de transacciones y las configuraciones específicas de la implementación.
- **Mapeo** donde realizaremos la conexión entre los atributos de las clases de entidad o Entidades JPA y las columnas de las tablas de la base de datos, además de marcar las restricciones o configuraciones específicas asociadas a dichos mapeos.
- **Administración de Entidades y Consultas** finalmente el código en el que aprovecharemos el uso del EntityManager para crear, actualizar o borrar entidades y las consultas JPQL para obtener listas de entidades a partir de los datos de la base de datos.

A continuación trabajaremos específicamente en cada uno de estos 3 ejes.

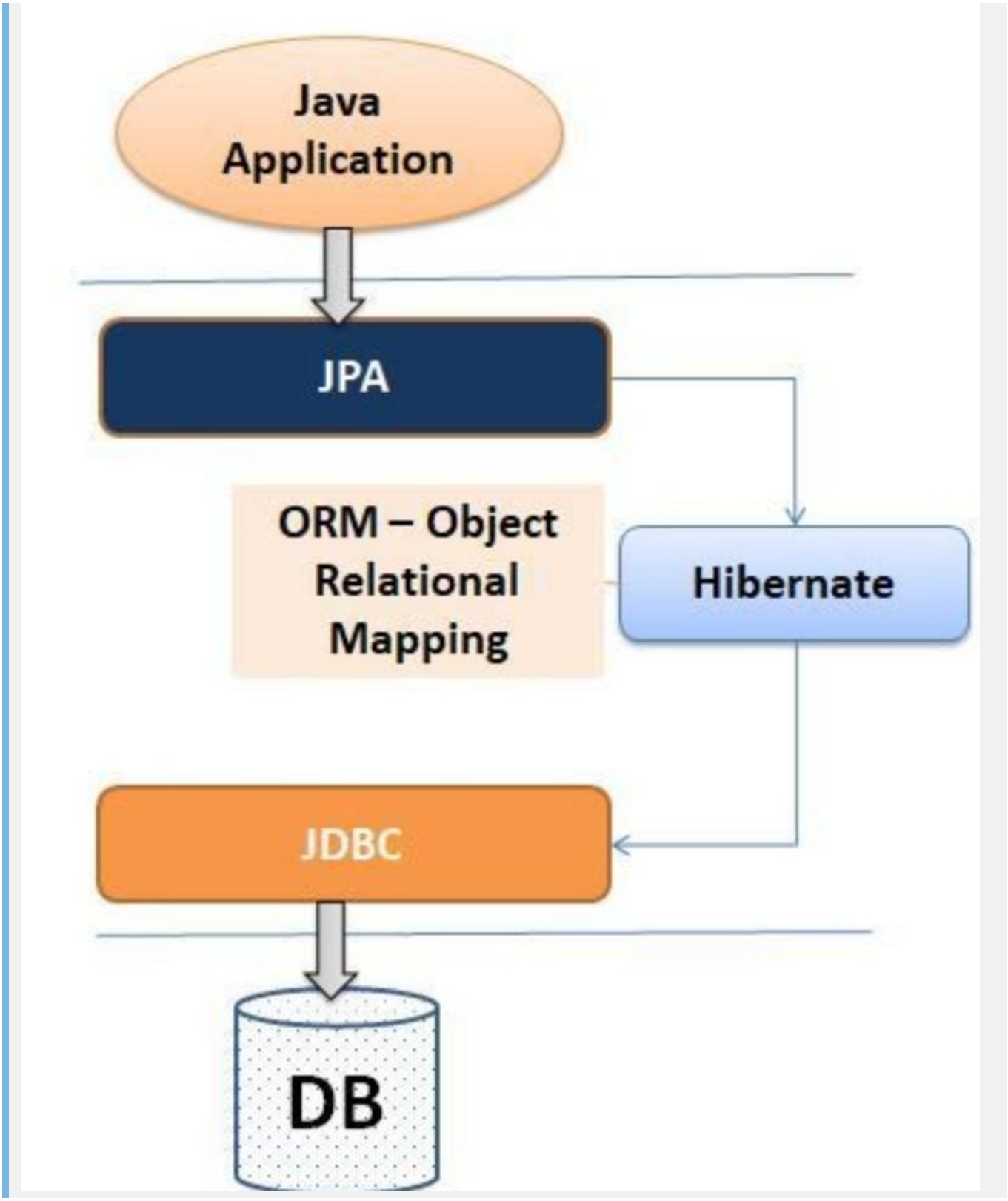
Configuración de JPA con Hibernate

¿Qué estamos configurando?

Antes de escribir código, definimos **qué piezas intervienen** en la persistencia con JPA:

- **API estándar (Jakarta Persistence)**: el contrato de anotaciones e interfaces que usamos en nuestro código.
- **Implementación (Hibernate)**: el motor ORM que ejecuta esas operaciones y traduce a SQL.
- **Driver JDBC**: el conector específico del motor (H2, PostgreSQL, etc.).
- **Pool de conexiones**: opcional en estos labs, clave en producción para eficiencia.

> Objetivo: que el alumno entienda **para qué sirve cada dependencia** y cómo encaja en la arquitectura (App → JPA → Hibernate → JDBC → Base de datos).



Dependencias Maven necesarias

- **Jakarta Persistence API** (`jakarta.persistence-api`).
- **Hibernate Core** como implementación de referencia.
- **Driver JDBC** del motor que usemos (H2, PostgreSQL, MySQL, etc.).
- Opcional: **pool de conexiones** como HikariCP.

Ejemplo de `pom.xml` mínimo:

```
<dependencies>
  <dependency>
```

```
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <version>2.2.224</version>
      <scope>runtime</scope>
    </dependency>

    <dependency>
      <groupId>jakarta.persistence</groupId>
      <artifactId>jakarta.persistence-api</artifactId>
      <version>3.1.0</version>
    </dependency>

    <dependency>
      <groupId>org.hibernate.orm</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>6.4.4.Final</version>
    </dependency>
  </dependencies>
```

Archivo `persistence.xml`

El archivo `META-INF/persistence.xml` define la **unidad de persistencia** (PU) y sus propiedades. Es el punto central de configuración cuando **no** usamos Spring. Sus bloques clave son:

- **Raíz y versión**: define el esquema XML de Jakarta Persistence.
- `<persistence-unit name=...>`: agrupa la configuración bajo un nombre lógico; nuestra app abrirá esa PU.
- `<provider>`: clase del proveedor JPA (Hibernate). Si se omite, Hibernate suele autodetectarse si está en el classpath.
- **Propiedades JDBC**: driver, URL, usuario y contraseña.
- `hibernate.dialect`: guía a Hibernate para emitir SQL compatible con el motor elegido.
- **Estrategia de DDL (`hibernate.hbm2ddl.auto`)**: controla cómo se crea/valida el esquema.
- **Opciones de log**: `show_sql`, `format_sql`, etc.
- **Transacciones**: `transaction-type` puede ser `RESOURCE_LOCAL` (lo usaremos en los labs) o `JTA` (aplicaciones EE/Spring con gestor transaccional).
- **Exploración de entidades**: por defecto, se escanean las clases anotadas en el classpath de la PU; se puede ajustar con `exclude-unlisted-classes` o `<class>` explícitas.

Ejemplo para H2 en memoria (recomendado para los primeros labs)

```
<persistence xmlns="https://jakarta.ee/xml/ns/persistence" version="3.0">
  <persistence-unit name="MiUnidad" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

    <properties>
      <!-- JDBC (H2 en memoria) -->
      <property name="jakarta.persistence.jdbc.driver" value="org.h2.Driver"/>
      <property name="jakarta.persistence.jdbc.url"
value="jdbc:h2:mem:demo;DB_CLOSE_DELAY=-1"/>
      <property name="jakarta.persistence.jdbc.user" value="sa"/>
      <property name="jakarta.persistence.jdbc.password" value=""/>

      <!-- Dialecto específico -->
      <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>

      <!-- DDL: para práctica inicial -->
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>

      <!-- Log -->
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.format_sql" value="true"/>
    </properties>
```

```
  </persistence-unit>
</persistence>
```

Ejemplo alternativo para PostgreSQL (cuando pasemos a motor real)

```
<persistence xmlns="https://jakarta.ee/xml/ns/persistence" version="3.0">
  <persistence-unit name="MiUnidad" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>

    <properties>
      <property name="jakarta.persistence.jdbc.driver" value="org.postgresql.Driver"/>
      <property name="jakarta.persistence.jdbc.url"
value="jdbc:postgresql://localhost:5432/sakila"/>
      <property name="jakarta.persistence.jdbc.user" value="appuser"/>
      <property name="jakarta.persistence.jdbc.password" value="apppass"/>

      <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect"/>

      <!-- En proyectos reales suele usarse 'validate' o 'update'; evitamos 'create' en
producción -->
      <property name="hibernate.hbm2ddl.auto" value="update"/>

      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.format_sql" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

Fundamentos de cada propiedad clave

- **`jakarta.persistence.jdbc.*`**: datos de conexión. En H2 memoria, `DB_CLOSE_DELAY=-1` mantiene la DB viva mientras exista la JVM.
- **`hibernate.dialect`**: *hints* de SQL para cada motor.
- **`hibernate.hbm2ddl.auto`**:
  - `none`/omitir: no toca el esquema.
  - `validate`: valida que el esquema exista y coincida con las entidades.
  - `update`: aplica cambios incrementales (útil en desarrollo, cuidado con drift).
  - `create`/`create-drop`: crea (y opcionalmente borra) el esquema al iniciar/finalizar.
- **`transaction-type`**:
  - `RESOURCE_LOCAL`: nosotros controlamos transacciones con `EntityTransaction`.
  - `JTA`: un contenedor/gestor externo coordina transacciones.
- **Logging**: `show_sql`/`format_sql` ayudan en el aprendizaje; en producción se recomienda usar un logger SQL y parametrización.

> Regla para `hibernate.hbm2ddl.auto`: **H2 en memoria + `create-drop`** para los primeros labs; luego migramos a **PostgreSQL + `validate`/`update`** para simular entornos reales.

Consideraciones de conexión

- Para proyectos educativos, podemos usar el pool por defecto de Hibernate.
- En producción, conviene un pool dedicado (HikariCP, C3P0, etc.).
- Es importante cerrar el `EntityManagerFactory` al finalizar la aplicación.

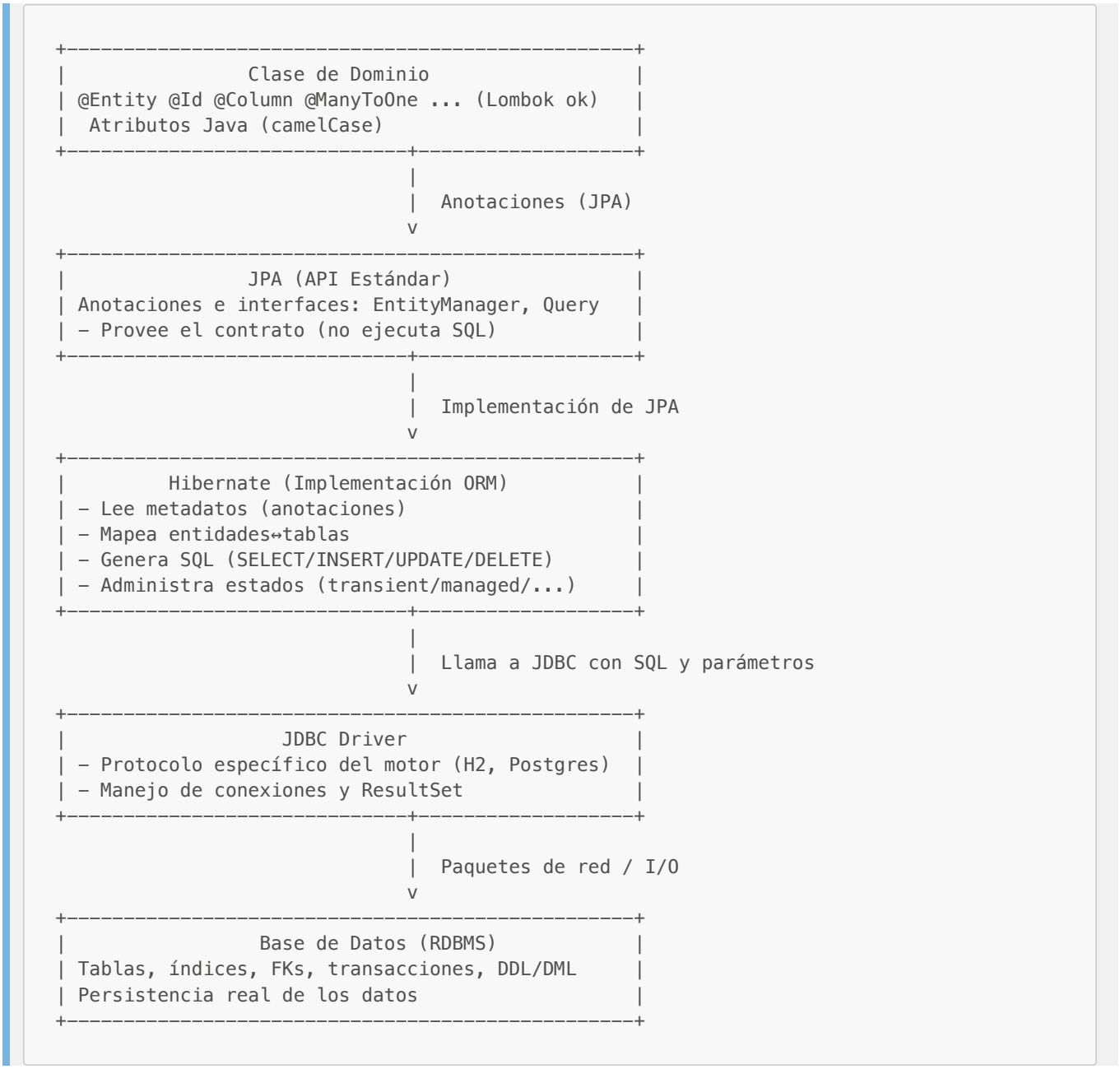
Mapeo de entidades

Proceso de Mapeo: de la tabla a la entidad

El mapeo en JPA consiste en **trazar la correspondencia** entre las estructuras de una base de datos relacional y las clases Java que representan el modelo de dominio.

Tabla ↔ Entidad

- Cada **tabla** de la base de datos se representa como una **clase anotada con `@Entity`**.
- Cada **columna** se convierte en un **atributo de la clase**, usando `@Column` cuando el nombre o propiedades difieren.
- La **clave primaria** se marca con `@Id`, y puede incluir una estrategia de generación (`@GeneratedValue`).
- Las **claves foráneas** se representan con relaciones (`@ManyToOne`, `@OneToMany`, `@ManyToMany`) según corresponda.



Restricciones generales (mínimas para que el mapeo funcione)

- **Entidad válida**
  - Debe estar anotada con `@Entity`.
  - Debe tener **constructor por defecto** (público o protegido).
  - Debe declarar **al menos una clave primaria** con `@Id` (o clave compuesta con `@EmbeddedId` / `@IdClass`).
  - La clase debe ser pública y no final; los campos persistentes no deben ser `static` ni `transient` (Java).
- **Tabla objetivo existente**
  - Si **no** se especifica `@Table`, JPA asumirá el nombre por convención (normalmente el nombre de la clase).
  - Si el nombre **no coincide** con la tabla real, usar `@Table(name = "NOMBRE_TABLA")`.
  - Si el esquema **no** es el por defecto, usar `@Table(schema = "ESQUEMA")`.

    - La tabla debe **existir** (o generarse vía `hibernate.hbm2ddl.auto`) y tener **PK** definida en DB si trabajamos con `validate`/`update`.

- **Columnas y atributos**
    - Para **diferencias de nombre** entre atributo y columna, usar `@Column(name = "COLUMNA_REAL")`.
    - **Not Null** en DB → `@Column(nullable = false)` en la entidad (coherencia de restricciones).
    - **Unique** por columna → `@Column(unique = true)`; para **unicidad compuesta**, usar `@Table(uniqueConstraints = @UniqueConstraint(columnNames = { ... }))`.
    - **Longitud** de `VARCHAR` → `@Column(length = N)`.
    - **Monetarios/precisión** → `BigDecimal` con `@Column(precision = P, scale = S)`.
    - **Enums** → `@Enumerated(EnumType.STRING)` (evita problemas si cambia el ordinal).
    - **Fechas/tiempos** → `java.time.*`; mapear con `@Temporal` solo si usás `java.util.Date/Calendar`.
- **Claves primarias y estrategias**
    - `@GeneratedValue(strategy = GenerationType.IDENTITY)` para autoincrement (H2/PostgreSQL).
    - `SEQUENCE` con `@SequenceGenerator` cuando uses secuencias nativas (muy común en PostgreSQL).
    - Para **claves compuestas**:
        - `@EmbeddedId` con clase `@Embeddable` (preferido por claridad), **o**
        - `@IdClass` (requiere duplicar campos de PK en la entidad y en la clase Id).
    - La PK debe ser **inmutable** a nivel semántico: evitá modificarla en runtime.
- **Nomenclatura: puente DB ↔ Java**
    - En DB solemos usar **UPPER_SNAKE_CASE** (p. ej., `FIRST_NAME`).
    - En Java usamos **camelCase** (`firstName`).
    - Resolver con `@Table(name = "TABLA_EN_DB")` y `@Column(name = "COLUMNA_EN_DB")`.
    - Si el motor es **case-sensitive** con identificadores entrecomillados, mantené la coherencia: mejor **evitar comillas** en DDL y mapear con nombres simples (H2 y Postgres sin comillas → nombres en minúscula normalizados).
- **Estados y DDL**
    - Si `hibernate.hbm2ddl.auto = validate`, el esquema **debe existir** y coincidir con el mapeo (tabla, columnas, tipos, PK/UK/NN).
    - Si `update`, Hibernate intentará ajustar el esquema (útil en desarrollo, con cautela).
    - Para labs con **H2 en memoria**, usar `create-drop` facilita la iteración rápida.
- **Relaciones (visión rápida)**
    - **Muchos→uno**: `@ManyToOne(fetch = LAZY)` con `@JoinColumn(name = "FK")` y tipos compatibles con la PK de la entidad objetivo.
    - **Uno→muchos**: usar con **cautela** (riesgo de cargas grandes). Preferir la navegación por consultas JPQL específicas.
    - **Muchos↔muchos**: preferible modelar **entidad intermedia** si necesitás atributos en la relación.

Checklist rápido de una entidad mínima

- `@Entity` presente.
- **PK** con `@Id` (y `@GeneratedValue` si corresponde).
- `@Table(name = "...", schema = "...")` si el nombre/esquema no coinciden.
- Atributos mapeados con `@Column(name = "...")` cuando el nombre difiere (snake_case ↔ camelCase).
- Restricciones alineadas: `nullable`, `unique`, `length`, `precision/scale`.
- Constructor por defecto y clase pública.
- Para relaciones: `@ManyToOne` + `@JoinColumn` con `fetch = LAZY`.

> Regla práctica en la cátedra: comenzamos con **entidades planas** y **N→1**; recién después, y solo si es necesario, agregamos **1→N** y consideramos **M↔M**. De esta forma mantenemos la consistencia con el esquema relacional y a la vez seguimos las convenciones de codificación de Java.

En este bloque mapeamos entidades de **Chinook** (versión simplificada, ya que luego lo haremos con Spring Data) usando **Lombok** y **JPA (Hibernate)** sobre **H2 en memoria**.

Empezamos con entidades planas, luego agregamos **relaciones muchos-a-uno** (con menos riesgo), mencionamos **uno-a-muchos** (con cautela), y cerramos con **muchos-a-muchos** a modo informativo.

Caso 1 - Entidades planas (sin relaciones)

Anotaciones fundamentales

- `@Entity`: marca la clase como entidad JPA (debe tener constructor por defecto).
- `@Table(name = "...")`: nombre de la tabla (si difiere del nombre de clase).
- `@Id`: identifica la PK.
- `@GeneratedValue(...)`: estrategia de generación de PK.
  - `IDENTITY`: autoincrement en DB.
  - `SEQUENCE`: usa secuencias (PostgreSQL, Oracle).
  - `AUTO`: deja al proveedor decidir.
- `@Column(name = "...", nullable = ..., length = ..., unique = ...)`: mapea columna y restricciones.

Lombok: recomendaciones

- `@Data`: getters/setters, `equals/hashCode`, `toString`.
- `@NoArgsConstructor` y `@AllArgsConstructor`: constructores.
- `@Builder`: patrón builder (útil en tests y labs).
- **Cuidado**: en entidades con relaciones bidireccionales, evitar ciclos en `toString`/`equals`. Podés usar `@ToString(exclude = ...)` y `@EqualsAndHashCode(of = "id")`.
  Estos son algunos de los problemas que hay que tener en cuenta al manejar el mapeo de relaciones.

Ejemplo: `Genre` (plano)

```
package utnfc.back.isi.jpa.domain;

import jakarta.persistence.*;
import lombok.*;

@Entity
@Table(name = "GENRE")
@Data @NoArgsConstructor @AllArgsConstructor @Builder
@EqualsAndHashCode(of = "id")
public class Genre {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "GENRE_ID")
  private Integer id;

  @Column(name = "NAME", length = 120, nullable = false)
  private String name;
}
```

Ejemplo: `MediaType` (plano)

```
@Entity
@Table(name = "MEDIA_TYPE")
@Data @NoArgsConstructor @AllArgsConstructor @Builder
@EqualsAndHashCode(of = "id")
public class MediaType {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "MEDIA_TYPE_ID")
  private Integer id;

  @Column(name = "NAME", length = 120, nullable = false)
```

```
  private String name;
}
```

Mini-ejemplo (persistencia básica en H2)

```
// Dentro de un método de prueba/lab
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Genre rock = Genre.builder().name("Rock").build();
em.persist(rock);
em.getTransaction().commit();
em.close();
```

Caso 2 - Relaciones muchos-a-uno (preferidas)

En este tipo de relaciones cada objeto se relaciona con otro único objeto haciendo referencia a su clave primaria. Esto mapeado en nuestra entidad provoca un nuevo atributo que contendrá una instancia del objeto de la tabla relacionada.

¿Por qué preferimos relaciones Muchos→Uno (N→1)?

En la práctica docente y en proyectos reales solemos comenzar modelando **desde el lado muchos hacia el lado uno**. La razón principal es que este tipo de relación:

- Nos garantiza que **cada fila de la tabla N solo añade la referencia a un objeto adicional** (el del lado 1).
- La carga de datos queda **acotada a un único objeto extra**, sin riesgo de traer colecciones completas por accidente.
- Evitamos los problemas de **carga masiva involuntaria** que suelen aparecer en las relaciones 1→N (colecciones potencialmente enormes).
- Es más **natural de implementar**: la tabla con la clave foránea es la que conoce a quién pertenece, por lo que el mapeo `@ManyToOne` + `@JoinColumn` refleja fielmente la estructura del esquema.
- Permite **consultas más predecibles y eficientes**, ya que Hibernate resuelve la clave ajena mediante un `JOIN` o una carga diferida de un solo objeto.

> En síntesis: las relaciones N→1 nos dan **simplicidad, previsibilidad y menor riesgo**. Por eso proponemos como primera opción al introducir el tema de asociaciones en JPA.

Anotaciones clave

- `@ManyToOne(fetch = FetchType.LAZY, optional = false)`: relación N→1.
- `@JoinColumn(name = "...", nullable = ...)`: columna FK en la tabla de la entidad propietaria.
- **Fundamento**: modelar **desde el lado N** reduce el riesgo de cargas masivas involuntarias y simplifica el grafo.

Ejemplo: `Album` → `Artist` (N→1)

```
@Entity
@Table(name = "ALBUM")
@Data @NoArgsConstructor @AllArgsConstructor @Builder
@EqualsAndHashCode(of = "id")
public class Album {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "ALBUM_ID")
  private Integer id;

  @Column(name = "TITLE", length = 200, nullable = false)
  private String title;

  @ManyToOne(fetch = FetchType.LAZY, optional = false)
  @JoinColumn(name = "ARTIST_ID", nullable = false)
```

```
  @ToString.Exclude
  private Artist artist;
}
```

```
@Entity
@Table(name = "ARTIST")
@Data @NoArgsConstructor @AllArgsConstructor @Builder
@EqualsAndHashCode(of = "id")
public class Artist {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "ARTIST_ID")
  private Integer id;

  @Column(name = "NAME", length = 160, nullable = false)
  private String name;
}
```

Ejemplo: `Track` → `Album` / `Genre` / `MediaType` (N→1)

```
@Entity
@Table(name = "TRACK")
@Data @NoArgsConstructor @AllArgsConstructor @Builder
@EqualsAndHashCode(of = "id")
public class Track {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "TRACK_ID")
  private Integer id;

  @Column(name = "NAME", length = 200, nullable = false)
  private String name;

  @ManyToOne(fetch = FetchType.LAZY, optional = false)
  @JoinColumn(name = "ALBUM_ID", nullable = false)
  @ToString.Exclude
  private Album album;

  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "GENRE_ID")
  @ToString.Exclude
  private Genre genre;

  @ManyToOne(fetch = FetchType.LAZY, optional = false)
  @JoinColumn(name = "MEDIA_TYPE_ID", nullable = false)
  @ToString.Exclude
  private MediaType mediaType;

  @Column(name = "MILLISECONDS", nullable = false)
  private Integer milliseconds;

  @Column(name = "UNIT_PRICE", nullable = false)
  private java.math.BigDecimal unitPrice;
}
```

Mini-ejemplo (persistir respetando N→1)

```
em.getTransaction().begin();
Artist artist = Artist.builder().name("AC/DC").build();
em.persist(artist);
Album album = Album.builder().title("Back in Black").artist(artist).build();
em.persist(album);
MediaType mt = MediaType.builder().name("MPEG audio").build();
em.persist(mt);
Genre g = Genre.builder().name("Hard Rock").build();
em.persist(g);
Track t = Track.builder().name("Hells Bells").album(album).genre(g).mediaType(mt)
              .milliseconds(312000).unitPrice(new java.math.BigDecimal("0.99")).build();
em.persist(t);
em.getTransaction().commit();
```

Caso 3 - Uno-a-muchos (con cautela)

¿Qué pasa con las relaciones Uno→Muchos (1→N)?

El mapeo de relaciones 1→N en JPA refleja el lado inverso de una relación N→1: un objeto puede estar asociado a una **colección de muchos otros objetos**.
Si bien es una construcción válida, requiere precauciones:

- **Riesgo de carga masiva**: un `Artist` puede tener cientos de `Album`. Si accedemos a la colección sin cuidado, Hibernate intentará cargar todos los elementos, con el consiguiente impacto de memoria y consultas múltiples (problema conocido como *N+1 queries*).
- **Colecciones grandes**: no siempre tiene sentido traer todos los elementos asociados; muchas veces conviene una consulta específica con JPQL o Criteria para obtener solo lo necesario.
- **Complejidad en equals/hashCode**: incluir colecciones en estos métodos puede generar ciclos infinitos o comparaciones muy costosas.
- **Gestión de cascadas**: al tener colecciones, hay que ser cuidadoso con el uso de `cascade = ALL`, ya que operaciones de borrado o persistencia pueden propagarse sin control.

Buenas prácticas

- Mantener siempre **`fetch = LAZY`** en las colecciones.
- Usar `@ToString.Exclude` y `@EqualsAndHashCode(of = "id")` con Lombok para evitar ciclos.
- Evitar navegar directamente por colecciones grandes: en su lugar, escribir **consultas específicas** (`SELECT a FROM Album a WHERE a.artist.id = :id`).
- Modelar primero el lado N→1 y **agregar el 1→N solo si la navegación es realmente necesaria**.

> En síntesis: las relaciones 1→N son útiles para expresar la navegación desde un objeto padre hacia sus hijos, pero conllevan **más riesgos de rendimiento y complejidad**. Por eso en la cátedra recomendamos usarlas **con cautela** y siempre priorizar las N→1 como base.

Anotaciones clave para 1→N

- `@OneToMany(mappedBy = "...", fetch = FetchType.LAZY, cascade = ...)` define la **vista inversa** del N→1.
- **Riesgo a mitigar**: cargar colecciones grandes por accidente (N+1, explosión de selects).
- Recomendación de cátedra: **comenzar modelando solo el N→1** y agregar 1→N **solo si es necesario** para navegación.

Ejemplo básico: `Artist` con sus `Album`

```
@OneToMany(mappedBy = "artist", fetch = FetchType.LAZY)
@ToString.Exclude
private java.util.List<Album> albums = new java.util.ArrayList<>();
```

> Buenas prácticas: mantener `LAZY`, no usar colecciones en `equals/hashCode`, y preferir consultas JPQL específicas para traer listas.

Ejemplo completo con `Album` y `Track`

```
// Ejemplo completo: Album ↔ Track (1→N y N→1) con manejo de referencia circular

package utnfc.back.isi.jpa.domain;

import jakarta.persistence.*;
import lombok.*;
import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.List;

// ---------------- Entidad Album ----------------
@Entity
@Table(name = "ALBUM")
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
@EqualsAndHashCode(of = "id")
public class Album {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "ALBUM_ID")
  private Integer id;

  @Column(name = "TITLE", length = 200, nullable = false)
  private String title;

  @OneToMany(mappedBy = "album", fetch = FetchType.LAZY,
             cascade = CascadeType.PERSIST, orphanRemoval = false)
  @ToString.Exclude
  private List<Track> tracks = new ArrayList<>();

  public void addTrack(Track t) {
    tracks.add(t);
    t.setAlbum(this);
  }

  public void removeTrack(Track t) {
    tracks.remove(t);
    t.setAlbum(null);
  }
}

// ---------------- Entidad Track ----------------
@Entity
@Table(name = "TRACK")
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
@EqualsAndHashCode(of = "id")
public class Track {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "TRACK_ID")
  private Integer id;
```

```
  @Column(name = "NAME", length = 200, nullable = false)
  private String name;

  @ManyToOne(fetch = FetchType.LAZY, optional = false)
  @JoinColumn(name = "ALBUM_ID", nullable = false)
  @ToString.Exclude
  private Album album;

  @Column(name = "MILLISECONDS", nullable = false)
  private Integer milliseconds;

  @Column(name = "UNIT_PRICE", nullable = false, precision = 10, scale = 2)
  private BigDecimal unitPrice;
}

// ---------------- Persistencia mínima (H2) ----------------

EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

Album album = Album.builder().title("Back in Black").build();

Track t1 = Track.builder()
    .name("Hells Bells")
    .milliseconds(312_000)
    .unitPrice(new java.math.BigDecimal("0.99"))
    .build();

Track t2 = Track.builder()
    .name("Shoot to Thrill")
    .milliseconds(315_000)
    .unitPrice(new java.math.BigDecimal("0.99"))
    .build();

album.addTrack(t1);
album.addTrack(t2);

em.persist(album);
em.getTransaction().commit();
em.close();

// ---------------- Consulta básica con find ----------------
Album found = em.find(Album.class, album.getId());
System.out.println("Album: " + found.getTitle());
System.out.println("Tracks: " + found.getTracks().size()); // LAZY: puede disparar un select
```

Caso 4 - Muchos-a-muchos (informativo)

- En Chinook, `Playlist` ↔ `Track` se modela con tabla intermedia `PLAYLIST_TRACK`.
- En JPA, se puede mapear con `@ManyToMany` + `@JoinTable`, pero en la práctica **preferimos modelar la tabla intermedia como entidad** (p. ej., `PlaylistTrack`) para poder agregar atributos (orden, fecha, etc.).

Esquema informativo con `@ManyToMany`

```
@ManyToMany
@JoinTable(name = "PLAYLIST_TRACK",
  joinColumns = @JoinColumn(name = "PLAYLIST_ID"),
  inverseJoinColumns = @JoinColumn(name = "TRACK_ID")
)
@ToString.Exclude
private java.util.Set<Track> tracks = new java.util.HashSet<>();
```

> Recomendación: para los labs, **evitar ManyToMany directo** y usar entidad intermedia si se necesita manipular la relación.

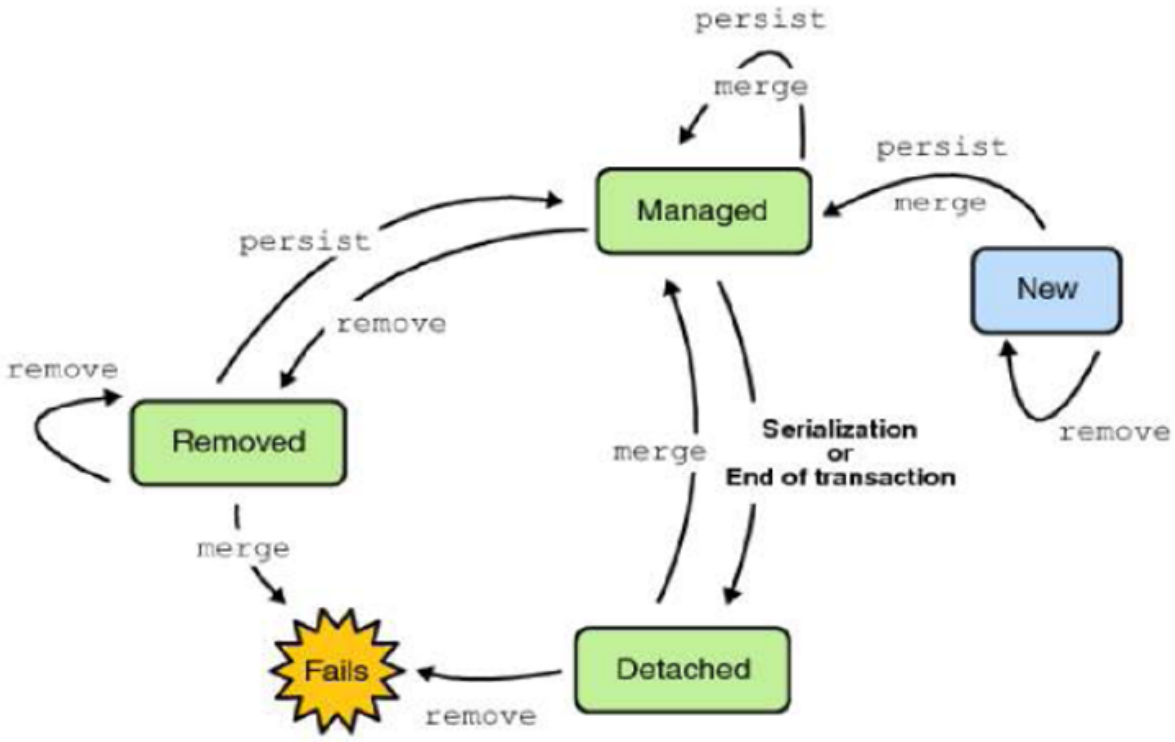
Notas de mapeo y validación

- Conviene alinear **nombres y tipos** con el esquema (todo **UPPER_SNAKE_CASE** si el script lo define así).
- Para valores monetarios, usar `BigDecimal` con precisión/escala adecuadas (`@Column(precision=, scale=)`).
- Usar `@NotNull`, `@Size`, etc. (Bean Validation) en los DTOs/entrada; en entidades solo lo necesario.
- Mantener `fetch = LAZY` por defecto en relaciones, y controlar la carga con JPQL/joins.

4. EntityManager y ciclo de vida de las entidades

El **EntityManager** es el corazón de JPA: gestiona las entidades y controla su estado dentro del *contexto de persistencia*. Una entidad puede atravesar distintos estados durante su vida útil, y conocerlos nos ayuda a evitar errores y entender los efectos de cada operación.

El EntityManager actúa como contenedor y proveedor de entidades gestionadas. Cuando una entidad entra en su persistence context, el EntityManager pasa a controlar su estado, sincronizar cambios y coordinar su persistencia en la base de datos. Esto significa que las entidades bajo su control siguen un ciclo de vida administrado.



Estados del ciclo de vida

- **New (Transient)**
  - Objeto recién creado en Java.
  - No está asociado a ningún contexto de persistencia.
  - No existe en la base de datos.
  - Operaciones válidas:
    - `persist()` → pasa a *Managed*.
    - `remove()` → no tiene efecto.
    - `merge()` → crea un nuevo objeto *Managed* con los mismos datos.
- **Managed (Persistente)**

    - Entidad dentro del *persistence context*.
    - Cualquier cambio en sus atributos será detectado y sincronizado con la BD al hacer `commit` o `flush()`.
    - Operaciones válidas:
        - `remove()` → pasa a *Removed*.
        - `detach()` / fin de la transacción → pasa a *Detached*.
        - `merge()` → mantiene estado *Managed*.
        - `persist()` → no tiene efecto (ya está gestionada).

- **Detached**
    - Entidad que fue *Managed* pero ya no pertenece al *persistence context* (porque cerramos el `EntityManager`, terminó la transacción, o se llamó a `detach()`).
    - Sus cambios **no se sincronizan automáticamente** con la BD.
    - Operaciones válidas:
        - `merge()` → crea/actualiza una copia *Managed* con los cambios.
        - `remove()` → provoca error (`IllegalArgumentException`) si no se vuelve a adjuntar.
        - `persist()` → vuelve a insertar (puede dar error de PK duplicada).

- **Removed**
    - Entidad marcada para eliminación en el *persistence context*.
    - Al hacer `commit`, se ejecuta `DELETE` en la base.
    - Operaciones válidas:
        - `persist()` → puede revertir la eliminación, vuelve a *Managed*.
        - `merge()` → vuelve a *Managed*.
        - `remove()` → no tiene efecto adicional.

Diagrama simplificado de transiciones

```
    persist
New ----------> Managed
|               |   |
|               |   | detach / commit end
v       merge   |   v
Managed <-------- Detached
  |                 |
  v      remove     |
Removed <-------------
```

Métodos principales del EntityManager

Método	Uso principal	Estado de entrada esperado	Resultado	Riesgos/Observaciones
`persist()`	Insertar una nueva entidad	NEW (Transient)	Pasa a Managed, se inserta al hacer commit/flush	Sobre *Managed* no hace nada, sobre *Detached* puede intentar reinserción (PK duplicada).
`merge()`	Reincorporar entidad detached o actualizar estado	DETACHED o NEW	Devuelve copia *Managed* con cambios aplicados	No actualiza el objeto original, sino una nueva instancia. Seguro en la mayoría de escenarios.

Método	Uso principal	Estado de entrada esperado	Resultado	Riesgos/Observaciones
`remove()`	Marcar para eliminación	MANAGED	Pasa a Removed, se ejecuta DELETE en commit	Falla si la entidad está DETACHED o NEW.
`detach()`	Desasociar del contexto	MANAGED	Entidad pasa a Detached	Cambios posteriores no se sincronizan.
`refresh()`	Sincronizar con BD descartando cambios	MANAGED	Sobrescribe el estado con datos actuales de BD	Cambios no confirmados se pierden.
`flush()`	Forzar sincronización inmediata	MANAGED	Ejecuta SQL pendiente sin cerrar transacción	Puede exponer errores antes del commit.

> [!Note] **Pensar el ciclo de vida como un grafo de estados** ayuda a comprender los efectos de cada método del `EntityManager`.

Contexto de persistencia y obtención del EntityManager

El contexto de persistencia es el conjunto de entidades gestionadas que mantiene el EntityManager en memoria mientras dura la transacción. El primer paso para trabajar con él es obtener una instancia de EntityManager desde el EntityManagerFactory (patrón Factory). En entornos de servidores Java EE/Jakarta EE o con Spring, normalmente el propio contenedor es quien administra y provee estas instancias de forma transparente. El persistence.xml establece la configuración de la unidad de persistencia, y es el puente que conecta la definición de la base de datos con el EntityManager.

Obtención del EntityManagerFactory (EMF)

Antes de obtener un EntityManager, debemos crear una instancia de EntityManagerFactory. Este se construye a partir del nombre de la unidad de persistencia definida en persistence.xml y actúa como el puente entre la configuración y la aplicación:

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("MiUnidad");
EntityManager em = emf.createEntityManager();
```

En aplicaciones de servidor (Java EE/Jakarta EE o Spring), normalmente el contenedor administra el EntityManagerFactory y provee directamente el EntityManager a través de inyección de dependencias (@PersistenceContext).

Ejemplo básico

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

Artist artist = new Artist();          // Estado: NEW
artist.setName("AC/DC");

em.persist(artist);                    // Estado: MANAGED
artist.setName("ACDC");                // Cambios se detectan

em.detach(artist);                     // Estado: DETACHED
artist.setName("AC-DC");               // Cambios NO se sincronizan

Artist merged = em.merge(artist);      // Crea copia MANAGED
em.remove(merged);                     // Estado: REMOVED

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

Ejemplo alternativo: `find` → actualizar → `detach`

Este flujo muestra cómo **modificar una entidad recuperada**, **persistir sus cambios** y luego **desasociarla** del contexto y **devolverla al llamador** con los cambios.

```
// Supongamos que existe un Artist con id conocido
Integer id = 1;

EntityManager em = emf.createEntityManager();
em.getTransaction().begin();

// 1) Recuperamos (MANAGED)
Artist a = em.find(Artist.class, id);
String nombreOriginal = a.getName();

// 2) Actualizamos mientras está MANAGED (JPA *podría* sincronizar en commit)
a.setName("Nombre Propuesto");

// 5) Commit → al estar en estado Managed, el cambio se sincroniza en la BD
//    Detecta todas las entidades actualizadas y realiza las modificaciones en la BD
em.getTransaction().commit();
// 6) Las entidades pasan a estado detached
em.close();

// 7) Retornamos el objeto fuera del contexto de persistencia
return a;

// ------------------------- Prueba -----------------------------------------

// Verificación: abrimos un nuevo EM y consultamos nuevamente
EntityManager em2 = emf.createEntityManager();
Artist verificado = em2.find(Artist.class, id);
System.out.println("DB ahora con: " + verificado.getName()); // imprime Nombre Propuesto
em2.close();
```

> Esto es comun en contextos donde necesitamos recibir los cambios desde otra capa de la aplicación realizar la persistencia de estos cambios y finalmente devolver a la otra capa la entidad modificada pero ya fuera del contexto de persistencia.

Comparativa de acciones tras un `find()` → modificar → commit

Acción aplicada a la entidad `a`	¿Se persiste el cambio?	SQL generado	Estado de `a` durante TX	Estado de `a` después del commit (cerrando EM)	Notas clave
Nada especial (solo commit)	✅ Sí	`UPDATE`	Managed	Detached	Caso normal: al ser *Managed*, sincroniza solo.
`em.flush()` antes de `commit`	✅ Sí	`UPDATE`	Managed	Detached	Fuerza sincronizar antes; detecta errores temprano.
`em.detach(a)` antes de `commit`	❌ No	—	Detached	Detached	Al sacarla del contexto, no se sincroniza.
`em.persist(a)`	❌ No (sin efecto)	—	Managed	Detached	`persist()` solo aplica a *NEW*. Sobre *Managed* no hace nada.
`em.merge(a)`	⚠ Sí, pero en copia	`UPDATE`	Retorno de `merge` = Managed	Retorno de `merge` = Detached	La instancia devuelta es la *Managed*, la original queda *Detached*.
`em.remove(a)`	❌ Se borra	`DELETE`	Removed	Detached (ya sin fila en BD)	Marca para borrar, no para actualizar.

Acción aplicada a la entidad `a`	¿Se persiste el cambio?	SQL generado	Estado de `a` durante TX	Estado de `a` después del commit (cerrando EM)	Notas clave
`em.clear()` (sobre EM completo)	❌ No	—	Todo Detached	Detached	Vacía el contexto, no se sincroniza salvo flush previo.
`em.refresh(a)`	❌ No (descarta cambios)	`SELECT`	Managed	Detached	Pisa los cambios en memoria con lo que está en la BD.

> [!TIP] **al cerrar el `EntityManager`, todas las entidades Managed pasan a Detached**.
> Si queremos seguir trabajando con ellas en otra transacción, debemos **recuperarlas otra vez** o hacer un `merge`.

Comparativa rápida de métodos del `EntityManager`

Método	¿Crea registro si no existe?	¿Actualiza si existe?	Estado requerido de la entidad pasada	Devuelve	Cuándo usar	Riesgos / notas
`persist(e)`	Sí (INSERT)	No	`NEW` (transient)	`void` (la entidad queda *Managed*)	Alta de entidades nuevas	Falla si la PK ya existe; sobre *Managed* no hace nada; sobre *Detached* intenta insertar (duplicado).
`merge(e)`	Sí (INSERT)	Sí (UPDATE)	`NEW` o `DETACHED`	**Nueva instancia *Managed***	Guardar cambios de *detached* o *upsert* manual	¡La instancia retornada es la *Managed*! La original sigue *Detached*. Puede crear inserciones si no existe.
`remove(e)`	No	Sí (DELETE)	`MANAGED`	`void`	Eliminar entidad gestionada	Lanza `IllegalArgumentException` si *Detached*; asegurar que esté *Managed* o hacer `em.remove(em.merge(e))`.
`detach(e)`	No	No	`MANAGED`	`void`	Sacar del contexto (evitar sincronización automática)	Cambios posteriores **no** se persisten salvo `merge`. Útil para ediciones offline/DTO.
`flush()`	N/A	N/A	N/A	`void`	Forzar sincronización pendiente a la BD dentro de la transacción	No cierra la transacción; puede revelar errores de constraints antes del `commit`.
`refresh(e)`	No	No (sobrescribe)	`MANAGED`	`void`	Descartar cambios en memoria y recargar desde BD	Pérdida de cambios no sincronizados; hace SELECT y pisa el estado actual.
`find(C,id)`	No	No	N/A	Instancia *Managed* o `null`	Recuperar entidad por PK	Respeta caché de 1er nivel; puede disparar proxy/lazy en relaciones.

Método	¿Crea registro si no existe?	¿Actualiza si existe?	Estado requerido de la entidad pasada	Devuelve	Cuándo usar	Riesgos / notas
`getReference(C,id)`	No	No	N/A	Proxy *Managed* (lazy)	Referencia perezosa cuando solo se necesita la PK	Acceso a campos no-PK puede disparar carga; puede lanzar `EntityNotFoundException` si no existe.

Operaciones CRUD con JPA (Create, Read, Update, Delete)

Las operaciones CRUD son las operaciones básicas de manipulación de datos en cualquier sistema de gestión de bases de datos. En JPA, estas operaciones se pueden realizar utilizando el `EntityManager`.
En base a los ejemplos anteriores podemos generar un ejemplo de cada una de las operaciones CRUD aprovechando las capacidades del EntityManager.
A continuación se describen estas operaciones en detalle:

Create (Crear)

Para crear una nueva entidad y guardarla en la base de datos, utilizamos el método `persist`.

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Usuario nuevoUsuario = new Usuario("nombre", "email@example.com");
em.persist(nuevoUsuario);
em.getTransaction().commit();
```

Retrieve (Obtener)

Para leer una entidad de la base de datos, utilizamos el método `find`.

```
EntityManager em = emf.createEntityManager();
Usuario usuarioExistente = em.find(Usuario.class, id);
```

Update (Actualizar)

Para actualizar una entidad ya existente, utilizamos el método `merge`.

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
usuarioExistente.setEmail("nuevo-email@example.com");
em.merge(usuarioExistente);
em.getTransaction().commit();
```

Delete (Eliminar)

Para eliminar una entidad de la base de datos, utilizamos el método `remove`.

```
EntityManager em = emf.createEntityManager();
em.getTransaction().begin();
Usuario usuarioAEliminar = em.find(Usuario.class, id);
```

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

Estas son las operaciones CRUD básicas en JPA. Al comprender estos fundamentos, estarás bien equipado para manejar la persistencia de datos en tus aplicaciones Java.

JPQL (Jakarta Persistence Query Language)

JPQL es el lenguaje de consultas orientado a entidades de JPA. Consulta **clases y sus relaciones** (modelo de objetos), no tablas. El proveedor (Hibernate) traduce JPQL a SQL del motor elegido.

Conceptos clave

- **Dominio de consulta**: clases anotadas con `@Entity` y sus relaciones.
- **Identificadores**: usamos **alias** para entidades (`from Album a`).
- **Expresiones de ruta**: navegación por relaciones (`a.artist.name`, `t.album.title`).
- **Proyecciones**: seleccionamos entidades completas, atributos o DTOs.
- **Parámetros**: `:named` o `?1` (recomendado `:named`).
- **Tipo-safe**: opcional con Criteria API (lo vemos al final como alternativa).

Sintaxis básica

```
select a from Album a
  where a.title like :title
  order by a.id
```

> - `select` puede omitirse si proyectamos la entidad completa: `from Album a`.
> - `where` usa operadores de Java/SQL (`=`, `<>`, `<`, `>`, `between`, `like`, `in`, `is null`).
> - `order by` admite rutas y dirección (`asc`/`desc`).

Joins

- **Inner join**: `select a from Album a join a.artist ar`.
- **Left join**: `select t from Track t left join t.genre g`.
- **Join con alias y filtro**: `select t from Track t join t.album a where a.id = :id`.
- **Fetch join** (carga asociada evitando N+1):

```
select distinct a
  from Album a
  left join fetch a.tracks
  where a.id = :id
```

> `fetch` trae la colección asociada en la misma consulta. Usar con cuidado en colecciones grandes.

Parámetros y paginado

```
List<Album> page = em.createQuery(
    "select a from Album a where a.title like :q order by a.id", Album.class)
  .setParameter("q", "%black%")
  .setFirstResult(0)      // offset
  .setMaxResults(10)      // page size
  .getResultList();
```

- `getSingleResult()` lanza `NoResultException` o `NonUniqueResultException`; preferir `getResultList()` y chequear vacío.

Proyecciones

- **Entidad completa**: `select a from Album a`.
- **Campos escalares**: `select a.id, a.title from Album a` → devuelve `List<Object[]>`.
- **DTO con constructor expression**:

```
select new utnfc.back.isi.jpa.dto.AlbumSummary(a.id, a.title, ar.name)
  from Album a join a.artist ar
  where a.title like :q
```

DTO ejemplo:

```
public record AlbumSummary(Integer id, String title, String artistName) { }
```

Agregaciones y agrupamientos

```
select ar.name, count(a)
  from Album a join a.artist ar
  group by ar.name
  having count(a) > 3
  order by count(a) desc
```

- Funciones comunes: `count`, `sum`, `avg`, `min`, `max`.
- `having` filtra sobre agregaciones.

Subconsultas

```
select t
  from Track t
  where t.unitPrice > (
    select avg(t2.unitPrice) from Track t2 where t2.genre = t.genre
  )
```

- Las subconsultas solo aparecen en `where`/`having` (no en `from`).

Funciones y operaciones útiles

- **Strings**: `lower`, `upper`, `concat`, `length`, `substring`, `trim`.
- **Numéricas**: `abs`, `mod`, `sqrt`.
- **Temporales**: `current_date`, `current_time`, `current_timestamp`.
- **Null-safe**: `coalesce(x, y)`, `nullif(x, y)`.
- **Case**: `case when ... then ... else ... end`.

Consultas de actualización/borrado en bloque

- **Update bulk**:

```
int n = em.createQuery(
  "update Track t set t.unitPrice = t.unitPrice * 1.1 where t.genre.name = :g")
 .setParameter("g", "Rock")
 .executeUpdate();
```

- **Delete bulk**:

```
int m = em.createQuery(
  "delete from Track t where t.album.id = :id")
 .setParameter("id", albumId)
 .executeUpdate();
```

> Son operaciones **bulk**: se ejecutan directamente en la base y **saltan el contexto de persistencia**. Tras usarlas, conviene `em.clear()` para evitar estados inconsistentes.

Named queries

- Declaración en la entidad:

```
@Entity
@NamedQuery(
  name = "Album.findByTitle",
  query = "select a from Album a where lower(a.title) like lower(:q)"
)
public class Album { ... }
```

- Uso:

```
List<Album> res = em.createNamedQuery("Album.findByTitle", Album.class)
  .setParameter("q", "%black%")
  .getResultList();
```

Ejemplos con Chinook (H2 en memoria)

- **Top 10 tracks por precio**:

```
select t
  from Track t
  order by t.unitPrice desc, t.id asc
```

- **Tracks de un álbum**:

```
  select t from Track t where t.album.id = :albumId order by t.id
```

- **Álbum con sus tracks (fetch)**:

```
  select distinct a from Album a left join fetch a.tracks where a.id = :id
```

- **Cantidad de tracks por género**:

```
select g.name, count(t)
  from Track t join t.genre g
  group by g.name
  order by count(t) desc
```

JPQL vs SQL nativo

- **JPQL** consulta **entidades** y entiende **relaciones**; es portable.
- **SQL nativo** consulta **tablas**; usar cuando necesitamos funciones específicas del motor o tuning fino.

```
List<Object[]> rows = em.createNativeQuery(
  "select a.album_id, a.title, count(t.track_id) as tracks " +
  "from album a left join track t on t.album_id = a.album_id " +
  "group by a.album_id, a.title order by tracks desc")
.getResultList();
```

Criteria API (mención)

- Alternativa **type-safe** para construir consultas en Java sin strings.
- Útil cuando generamos consultas dinámicas complejas.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Album> cq = cb.createQuery(Album.class);
Root<Album> a = cq.from(Album.class);
cq.select(a).where(cb.like(a.get("title"), cb.literal("%black%")));
List<Album> out = em.createQuery(cq).getResultList();
```

Buenas prácticas JPQL

- Mantener `LAZY` por defecto y usar `fetch join` solo en casos puntuales.
- Para colecciones grandes, paginar siempre (`setFirstResult` + `setMaxResults`).
- En DTOs, preferir **constructor expressions**.
- Evitar `select *` equivocado: en JPQL es `select e` (entidad completa).
- Tras **bulk update/delete**, hacer `em.clear()`.
- Validar `getSingleResult()` con cuidado; mejor `getResultList()` y controlar tamaño.

Anexo - Implementación de capa de acceso a datos con patrón Repositorio

Queda pendiente por ahora

Enlaces relacionados

- Introducción a JPA
- Java Persistence API (JPA)