

Bloque 6 – Actividad guiada: Patrón Repositorio con JPA

Stack: Java 21 + Maven, JPA (Hibernate), H2, Lombok, JUnit 5. **Paquete base:** `utnfc.isi.back.jpa` **DB:** Chinook sobre H2 (los scripts ya los aporta la cátedra) en `src/main/resources/sql/`.

Contexto didáctico En este bloque implementamos el *Patrón Repositorio* con **JPA puro** (sin Spring Data aún). La idea es que comprendamos la frontera de acceso a datos, separando *qué* queremos hacer (repositorio) de *cómo* se hace (JPA + Hibernate). Además, dejamos el proyecto listo para que, más adelante, podamos comparar con Spring Data.



Aclaración Inicial

Los componentes de código que se comparten a continuación como material para la actividad están a modo de ejemplificación y orientación guía de la actividad, los mismos no fueron probados al 100% puede que necesiten ajustes o correcciones para que todo quede coordinado.

0. Objetivos de aprendizaje

- Entender el **Patrón Repositorio** y su rol como frontera de persistencia.
- Diseñar un **Repositorio Base genérico** y **Repositorios específicos** por entidad.
- Usar **JPA/Hibernate** con **H2** para CRUD y consultas con **JPQL**.
- Aplicar **Lombok** en entidades para reducir boilerplate.
- Exponer **paginación simple** (offset/limit) desde el repositorio base.
- Verificar con **CLI de demostración** y **tests JUnit**.

1. Preparación del proyecto (Maven)

1.1 Estructura inicial

```

/ (raíz)
├─ pom.xml
├─ src
│   └─ main
│       └─ java
│           └─ utnfc
│               └─ isi
│                   └─ back
│                       └─ jpa
│                           └─ app
│                               └─ Main.java
│                           └─ config
│                               └─ LocalEntityManagerProvider.java
│                               └─ DataInitializer.java (punto de integración con
los scripts)
│                                   └─ domain
│                                       └─ Artist.java
│                                       └─ Album.java
│                                       └─ Track.java
│                                   └─ repository
│                                       └─ base
│                                           └─ BaseRepository.java
│                                           └─ JpaBaseRepository.java
│                                           └─ PageRequest.java
│                                       └─ ArtistRepository.java
│                                       └─ AlbumRepository.java
│                                       └─ TrackRepository.java
│                                   └─ service
│                                       └─ CatalogService.java
│
│   └─ resources
│       └─ META-INF
│           └─ persistence.xml
│       └─ sql
│           └─ 01_chinook_tables.sql
│           └─ 02_chinook_data.sql
│           └─ 03_chinook_constraints_indexes.sql
│           └─ 03_chinook_sequences.sql
│
└─ test
    └─ java
        └─ utnfc
            └─ isi
                └─ back
                    └─ jpa
                        └─ repository
                            └─ ArtistRepositoryTest.java

```

1.2 pom.xml (dependencias clave)

```

<project>
  <modelVersion>4.0.0</modelVersion>

```

```
<groupId>utnfc.isi.back</groupId>
<artifactId>bloque6-jpa-repository</artifactId>
<version>1.0.0</version>
<properties>
  <maven.compiler.source>21</maven.compiler.source>
  <maven.compiler.target>21</maven.compiler.target>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <junit.jupiter.version>5.10.2</junit.jupiter.version>
  <hibernate.version>6.5.2.Final</hibernate.version>
</properties>

<dependencies>
  <!-- JPA + Hibernate -->
  <dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>${hibernate.version}</version>
  </dependency>

  <!-- H2 DB para dev/test -->
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>

  <!-- Lombok -->
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.32</version>
    <scope>provided</scope>
  </dependency>

  <!-- JUnit 5 -->
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>${junit.jupiter.version}</version>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.11.0</version>
      <configuration>
        <release>21</release>
      </configuration>
    </plugin>
    <plugin>
```

```

    <groupId>org.codehaus.mojo</groupId>
    <artifactId>exec-maven-plugin</artifactId>
    <version>3.3.0</version>
    <configuration>
        <mainClass>utnfc.isi.back.jpa.app.Main</mainClass>
    </configuration>
</plugin>
</plugins>
</build>
</project>

```

2. Configuración de JPA

2.1 META-INF/persistence.xml

- **Unidad:** chinookPU
- **Conexión:** H2 (file o memoria). Para la actividad usamos memoria.
- **DDL:** none (los scripts de Chinook ya generan esquema y datos).

```

<persistence xmlns="https://jakarta.ee/xml/ns/persistence" version="3.0">
  <persistence-unit name="chinookPU">
    <properties>
      <property name="jakarta.persistence.jdbc.driver"
value="org.h2.Driver"/>
      <property name="jakarta.persistence.jdbc.url"
value="jdbc:h2:mem:chinook;DB_CLOSE_DELAY=-1;MODE=PostgreSQL"/>
      <property name="jakarta.persistence.jdbc.user" value="sa"/>
      <property name="jakarta.persistence.jdbc.password" value=""/>

      <!-- Hibernate -->
      <property name="hibernate.hbm2ddl.auto" value="none"/>
      <property name="hibernate.show_sql" value="false"/>
      <property name="hibernate.format_sql" value="true"/>
    </properties>
  </persistence-unit>
</persistence>

```

2.2 LocalEntityManagerProvider

Propósito: Centralizar la creación del `EntityManagerFactory` y entregar `EntityManager` locales por hilo. Es nuestro *contexto mínimo*.

```

package utnfc.isi.back.jpa.config;

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;

```

```

public final class LocalEntityManagerProvider {
    private static final String PU_NAME = "chinookPU";
    private static final EntityManagerFactory emf = buildEmf();

    private static final ThreadLocal<EntityManager> current = new
    ThreadLocal<>();

    private LocalEntityManagerProvider() {}

    private static EntityManagerFactory buildEmf() {
        // 1) Crear EMF una sola vez en la app
        return Persistence.createEntityManagerFactory(PU_NAME);
    }

    public static EntityManager em() {
        // 2) Un EM por hilo cuando se pida, reutilizable dentro del hilo
        EntityManager em = current.get();
        if (em == null || !em.isOpen()) {
            em = emf.createEntityManager();
            current.set(em);
        }
        return em;
    }

    public static void closeCurrent() {
        EntityManager em = current.get();
        if (em != null) {
            em.close();
            current.remove();
        }
    }

    public static void shutdown() {
        // 3) Cierre ordenado del EMF al terminar la app/tests
        if (emf.isOpen()) emf.close();
    }
}

```

Nota: Gestionamos transacciones manualmente en los repositorios/servicios (begin/commit/rollback).

2.3 **DataInitializer** (punto de integración con scripts)

[!Note] Ver Ejemplos 1 y dos del Apunte 13 - ORM con JPA Allí encontrará también los scripts de inicialización de la base de datos que se mencionan en la estructura del proyecto y un ejemplo de implementación de la clase de inicialización.

Propósito: Ejecutar los 4 scripts `sql/*.sql` antes de usar los repositorios. La cátedra provee los scripts; aquí se documenta el lugar donde se invoca su carga.

Pseudocódigo (implementación real queda en los ejemplos de JPA):

```
class DataInitializer {  
    void init() {  
        // 1) Abrir conexión JDBC a la URL de H2 usada por JPA  
        // 2) Leer y ejecutar en orden 01-schema.sql, 02-constraints.sql, 03-  
views.sql, 04-data.sql  
        // 3) Cerrar conexión  
    }  
}
```

3. Dominio y entidades (con Lombok)

Alcance didáctico: *Artist*, *Album*, *Track* (subset de Chinook). **Reglas:** Relaciones básicas, sin cascadas agresivas (para controlar mejor los ejemplos).

Buenas prácticas Lombok: evitar `@ToString` sobre colecciones bidireccionales; definir `equals/hashCode` por `id`.

3.1 *Artist*

```
package utnfc.isi.back.jpa.domain;  
  
import jakarta.persistence.*;  
import lombok.*;  
  
@Entity  
@Table(name = "Artist")  
@Getter @Setter  
@NoArgsConstructor @AllArgsConstructor @Builder  
@EqualsAndHashCode(of = "artistId")  
public class Artist {  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    @Column(name = "ArtistId")  
    private Long artistId;  
  
    @Column(name = "Name", nullable = false)  
    private String name;  
}
```

3.2 *Album*

```
@Entity  
@Table(name = "Album")  
@Getter @Setter  
@NoArgsConstructor @AllArgsConstructor @Builder  
@EqualsAndHashCode(of = "albumId")  
public class Album {
```

```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "AlbumId")
private Long albumId;

@ManyToOne(optional = false, fetch = FetchType.LAZY)
@JoinColumn(name = "ArtistId")
private Artist artist;

@Column(name = "Title", nullable = false)
private String title;
}

```

3.3 Track

```

@Entity
@Table(name = "Track")
@Getter @Setter
@NoArgsConstructor @AllArgsConstructor @Builder
@EqualsAndHashCode(of = "trackId")
public class Track {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "TrackId")
    private Long trackId;

    @ManyToOne(optional = false, fetch = FetchType.LAZY)
    @JoinColumn(name = "AlbumId")
    private Album album;

    @Column(name = "Name", nullable = false)
    private String name;

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

```

Checklist entidad: nombres de columnas según Chinook; **IDENTITY** para H2; evitar cargar relaciones pesadas en `toString()`.

4. Repositorio Base genérico

Idea: Definir una interfaz *genérica* y una implementación JPA única que reutilicemos en los repos específicos.

4.1 BaseRepository<T, ID>

```
package utnfc.isi.back.jpa.repository.base;

import java.util.*;

public interface BaseRepository<T, ID> {
    T save(T entity);
    Optional<T> findById(ID id);
    List<T> findAll();
    List<T> findAll(int offset, int limit); // paginación simple
    void delete(T entity);
    long count();
    boolean existsById(ID id);
}
```

4.2 PageRequest (utilidad simple)

```
package utnfc.isi.back.jpa.repository.base;

public record PageRequest(int offset, int limit) {
    public PageRequest {
        if (offset < 0 || limit <= 0) throw new
        IllegalArgumentException("Valores de paginación inválidos");
    }
}
```

4.3 JpaBaseRepository<T, ID>

Puntos didácticos:

- Resolver `Class<T>` vía inyección en el constructor.
- Gestionar **transacciones locales**: `begin/commit/rollback` por operación *mutante*.
- Reutilizar JPQL para `findAll`.

```
package utnfc.isi.back.jpa.repository.base;

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityTransaction;
import java.util.*;

public class JpaBaseRepository<T, ID> implements BaseRepository<T, ID> {
    private final Class<T> entityClass;
    private final EntityManager em; // provisto por
    LocalEntityManagerProvider

    public JpaBaseRepository(Class<T> entityClass, EntityManager em) {
        this.entityClass = entityClass;
        this.em = em;
    }
}
```



```
@Override
public T save(T entity) {
    EntityTransaction tx = em.getTransaction();
    try {
        tx.begin();
        T merged = em.merge(entity); // sirve para persistir o actualizar
        tx.commit();
        return merged;
    } catch (RuntimeException e) {
        if (tx.isActive()) tx.rollback();
        throw e;
    }
}

@Override
public Optional<T> findById(ID id) {
    return Optional.ofNullable(em.find(entityClass, id));
}

@Override
public List<T> findAll() {
    String ql = "select e from " + entityClass.getSimpleName() + " e";
    return em.createQuery(ql, entityClass).getResultList();
}

@Override
public List<T> findAll(int offset, int limit) {
    String ql = "select e from " + entityClass.getSimpleName() + " e";
    return em.createQuery(ql, entityClass)
        .setFirstResult(offset)
        .setMaxResults(limit)
        .getResultList();
}

@Override
public void delete(T entity) {
    EntityTransaction tx = em.getTransaction();
    try {
        tx.begin();
        T managed = entity;
        if (!em.contains(entity)) {
            managed = em.merge(entity);
        }
        em.remove(managed);
        tx.commit();
    } catch (RuntimeException e) {
        if (tx.isActive()) tx.rollback();
        throw e;
    }
}

@Override
public long count() {
```

```

        String ql = "select count(e) from " + entityClass.getSimpleName() + "
e";
        return em.createQuery(ql, Long.class).getSingleResult();
    }

    @Override
    public boolean existsById(ID id) {
        return findById(id).isPresent();
    }
}

```

Fundamento: encapsulamos la *mecánica JPA* (em, transacciones, JPQL) en una implementación base reutilizable.

5. Repositorios específicos (JPQL derivada y filtros)

5.1 ArtistRepository

```

package utnfc.isi.back.jpa.repository;

import utnfc.isi.back.jpa.domain.Artist;
import utnfc.isi.back.jpa.repository.base.BaseRepository;
import java.util.List;

public interface ArtistRepository extends BaseRepository<Artist, Long> {
    List<Artist> findByNameContainsIgnoreCase(String q, int offset, int
limit);
}

```

Implementación (JpaArtistRepository):

```

public class JpaArtistRepository extends JpaBaseRepository<Artist, Long>
implements ArtistRepository {
    public JpaArtistRepository(EntityManager em) { super(Artist.class, em);
}

    @Override
    public List<Artist> findByNameContainsIgnoreCase(String q, int offset,
int limit) {
        String jpql = "select a from Artist a where lower(a.name) like
lower(:q) order by a.name";
        return em().createQuery(jpql, Artist.class)
            .setParameter("q", "%" + q + "%")
            .setFirstResult(offset)
            .setMaxResults(limit)
            .getResultList();
    }
}

```

```
private EntityManager em() { /* acceso protegido al EM (ver nota) */
return null; }
}
```

Nota: Para acceder al `EntityManager` desde la subclase, podemos (a) cambiar `em` a `protected` en `JpaBaseRepository`, o (b) proveer un `protected EntityManager em()` en la base. Dejar **(a)** para simplificar en clase.

5.2 AlbumRepository

```
public interface AlbumRepository extends BaseRepository<Album, Long> {
    List<Album> findByArtistId(Long artistId, int offset, int limit);
}
```

Implementación (`JpaAlbumRepository`):

```
public class JpaAlbumRepository extends JpaBaseRepository<Album, Long>
implements AlbumRepository {
    public JpaAlbumRepository(EntityManager em) { super(Album.class, em); }

    @Override
    public List<Album> findByArtistId(Long artistId, int offset, int limit)
    {
        String jpql = "select a from Album a where a.artist.artistId =
:artistId order by a.title";
        return em.createQuery(jpql, Album.class)
            .setParameter("artistId", artistId)
            .setFirstResult(offset)
            .setMaxResults(limit)
            .getResultList();
    }
}
```

5.3 TrackRepository

```
public interface TrackRepository extends BaseRepository<Track, Long> {
    List<Track> findByAlbumId(Long albumId, int offset, int limit);
    List<Track> findByNameContainsIgnoreCase(String q, int offset, int
limit);
}
```

Implementación (`JpaTrackRepository`): JPQL similar a los anteriores.

6. Servicio de aplicación (uso transaccional)

Idea: Orquestar operaciones que tocan varias entidades con **transacciones claras**.

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

import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityTransaction;
import utnfc.isi.back.jpa.domain.*;
import utnfc.isi.back.jpa.repository.*;

public class CatalogService {
    private final EntityManager em;
    private final ArtistRepository artists;
    private final AlbumRepository albums;
    private final TrackRepository tracks;

    public CatalogService(EntityManager em,
                          ArtistRepository artists,
                          AlbumRepository albums,
                          TrackRepository tracks) {

        this.em = em;
        this.artists = artists;
        this.albums = albums;
        this.tracks = tracks;
    }

    public Album createAlbumForArtist(Long artistId, String title) {
        EntityTransaction tx = em.getTransaction();
        try {
            tx.begin();
            var artist = artists.findById(artistId).orElseThrow();
            var album = Album.builder().artist(artist).title(title).build();
            Album saved = albums.save(album);
            tx.commit();
            return saved;
        } catch (RuntimeException e) {
            if (tx.isActive()) tx.rollback();
            throw e;
        }
    }
}
```

Fundamento: mantenemos las reglas de negocio y la *unidad transaccional* en el servicio.

7. CLI de demostración (Main)

Objetivo: permitir probar rápidamente sin tests, usando `System.out.println`.

Pseudocódigo:

```

public class Main {
    public static void main(String[] args) {
        // 0) Init DB (scripts) -> DataInitializer.init();
        var em = LocalEntityManagerProvider.em();

        var artistRepo = new JpaArtistRepository(em);
        var albumRepo = new JpaAlbumRepository(em);
        var trackRepo = new JpaTrackRepository(em);

        var service = new CatalogService(em, artistRepo, albumRepo,
        trackRepo);

        // 1) Listar primeros 10 artistas
        artistRepo.findAll(0, 10).forEach(a ->
        System.out.println(a.getName()));

        // 2) Buscar por nombre
        artistRepo.findByNameContainsIgnoreCase("queen", 0, 5)
            .forEach(a -> System.out.println("Match: " + a.getName()));

        // 3) Crear álbum para un artista existente
        var newAlbum = service.createAlbumForArtist(artistId(1L), "Demo
        Album");
        System.out.println("Nuevo álbum: " + newAlbum.getTitle());

        LocalEntityManagerProvider.closeCurrent();
        LocalEntityManagerProvider.shutdown();
    }
}

```

8. Tests JUnit (slice básico de repositorio)

ArtistRepositoryTest – ideas de casos:

- `findAll()` devuelve resultados (> 0) tras la init de Chinook.
- `findByNameContainsIgnoreCase()` filtra correctamente.
- `save()` persiste y asigna `id`.
- `delete()` elimina.

Esqueleto:

```

class ArtistRepositoryTest {
    @BeforeAll
    static void setUpAll() {
        // DataInitializer.init(); // ejecuta los 4 scripts
    }

    @BeforeEach
    void setUp() {

```

```
    em = LocalEntityManagerProvider.em();
    repo = new JpaArtistRepository(em);
}

@AfterEach
void tearDown() {
    LocalEntityManagerProvider.closeCurrent();
}

@AfterAll
static void shutdownAll() {
    LocalEntityManagerProvider.shutdown();
}

@Test
void findAll_returnsData() {
    var list = repo.findAll();
    assertFalse(list.isEmpty());
}
}
```

9. Paginación simple: criterios y contrato

- **Contrato:** `findAll(int offset, int limit)` y variantes en repos específicos.
- **Regla:** `offset >= 0` y `limit > 0`.
- **Uso:** UI o CLI decide `offset/limit`; repos usan `setFirstResult/setMaxResults`.

Más adelante, cuando veamos Spring Data, mapearemos este contrato a `Pageable/Page`.

10. Actividad paso a paso para los alumnos

Paso 1 – Crear proyecto y POM (15')

1. Generar proyecto Maven vacío (Java 21).
2. Agregar dependencias (Hibernate, H2, Lombok, JUnit).
3. Verificar `mvn -q -DskipTests package`.

Criterio de aceptación: compila sin tests.

Paso 2 – Configurar JPA (20')

1. Crear `META-INF/persistence.xml` con `chinookPU` y H2 en memoria.
2. Implementar `LocalEntityManagerProvider`.

Criterio: EMF crea sin errores.

Paso 3 – Integrar inicialización de Chinook (15')

1. Ubicar los 4 scripts en `src/main/resources/sql/`.
2. Implementar `DataInitializer.init()` (o usar la versión provista por la cátedra).
3. Invocar `init()` al inicio del `Main` y en `@BeforeAll` de los tests.

Criterio: las tablas y datos están disponibles (una consulta simple devuelve filas).

Paso 4 – Modelar entidades con Lombok (25')

1. Implementar `Artist`, `Album`, `Track` con anotaciones JPA.
2. Agregar Lombok (`@Getter`, `@Setter`, `@Builder`, etc.).
3. Validar nombres de columnas según Chinook.

Criterio: `em.find(Artist.class, 1L)` retorna datos.

Paso 5 – Repositorio base genérico (30')

1. Crear `BaseRepository<T, ID>`.
2. Implementar `JpaBaseRepository<T, ID>` con `merge`, `find`, `JPQL`, `delete` y `count`.
3. Agregar `findAll(offset, limit)`.

Criterio: tests básicos del base repo pasan.

Paso 6 – Repos específicos y filtros (30')

1. `ArtistRepository` con `findByNameContainsIgnoreCase`.
2. `AlbumRepository` por `artistId`.
3. `TrackRepository` por `albumId` y por nombre.

Criterio: consultas JPQL devuelven resultados coherentes.

Paso 7 – Servicio y transacciones (20')

1. `CatalogService.createAlbumForArtist(artistId, title)`.
2. Manejar begin/commit/rollback sobre `em.getTransaction()`.

Criterio: crear un álbum y verificarlo luego por `findAll`.

Paso 8 – CLI de demostración (15')

1. `Main` que liste, busque y cree.
2. Ejecutar con `mvn -q exec:java`.

Criterio: salida por consola con resultados esperados.

Paso 9 – Tests JUnit (30')

1. Tests de `ArtistRepository`.

2. Tests mínimos de servicio.

Criterio: suite verde.

12. Próximos pasos (teaser)

- Agregar **criterios de ordenamiento** opcionales en `findAll`.
 - Implementar **criterios compuestos** (ej: filtros múltiples en Track).
 - Comparativa con **Spring Data JPA** (derivación de métodos, `Pageable`, `@Transactional`).
-

Apéndice A – Pseudocódigos de soporte

A.1 JPQL genérico en repos base

```
ql = "select e from {Entity} e"
query = em.createQuery(ql, entityClass)
if (paginar) {
    query.setFirstResult(offset)
    query.setMaxResults(limit)
}
return query.getResultList()
```

A.2 Transacciones en servicio

```
em = LocalEntityManagerProvider.em()
tx = em.getTransaction()
try:
    tx.begin()
    ... (operaciones sobre repos)
    tx.commit()
except:
    if tx.isActive(): tx.rollback()
    throw
```

A.3 Búsquedas con `like`

```
jpql = "select a from Artist a where lower(a.name) like lower(:q) order by a.name"
param q = "%" + criterio + "%"
```

Apéndice B – Resumen final

B.1. Preguntas guía por paso

- **Paso 1 – POM / Build**

- ¿Qué rol cumple cada dependencia (Hibernate, H2, Lombok, JUnit)?
- ¿Por qué usamos `exec-maven-plugin` y cómo ayuda a reproducir la ejecución?

- **Paso 2 – JPA / EMF**

- ¿Por qué el `EntityManagerFactory` es único (singleton) y el `EntityManager` es local por hilo?
- ¿Dónde conviene iniciar/terminar la transacción y por qué?

- **Paso 3 – Inicialización Chinook**

- ¿Por qué el orden `01-schema` → `02-constraints` → `03-views` → `04-data`?
- ¿Qué implican `hbm2ddl.auto=none` y usar la **misma URL JDBC** para JPA y el cargador?

- **Paso 4 – Entidades con Lombok**

- ¿Qué ventajas y riesgos trae `@Builder` en entidades JPA?
- ¿Por qué `equals/hashCode` sólo por `id`? ¿Qué evitar en `toString`?

- **Paso 5 – Repositorio Base**

- ¿Cuándo usar `persist` vs `merge`? ¿Por qué `merge` sirve para insertar y actualizar?
- ¿Qué responsabilidad tiene el repositorio respecto a *transacciones*?

- **Paso 6 – Repos específicos / JPQL**

- ¿Por qué usar `lower(...)` `like :q` y ordenar por un campo (`order by`)?
- ¿Cuándo conviene limitar resultados (`offset/limit`) aunque sea una demo?

- **Paso 7 – Servicio**

- ¿Qué define la *unidad transaccional* en un caso de uso? ¿Qué hacer ante una excepción?

- **Paso 8 – CLI**

- ¿Cómo validar rápidamente que los repos funcionan sin escribir tests (casos mínimos)?

- **Paso 9 – Tests**

- ¿Qué *fixtures* necesita el test para ser estable? ¿Cómo aislar casos *green* y *errores*?

B.2. Errores frecuentes a vigilar

- **Transacciones:** olvidar `begin/commit` en operaciones mutantes o no hacer `rollback` en excepciones.
- **Ciclo de vida del EM:** reusar un `EntityManager` cerrado o no cerrarlo en `@AfterEach/CLI`.
- **Inicialización:** no ejecutar los 4 scripts antes de consultar datos (tests/CLI fallan en `findAll`).
- **Relaciones:** poner `cascade = ALL` donde no corresponde o dejar `ManyToOne` en *EAGER* (riesgo N+1). Usar `fetch = LAZY` conscientemente.

- **JPQL:** referenciar *nombres de propiedades* de la entidad (no nombres de columnas). Ej.: `a.artist.artistId`, no `ArtistId` suelto.
- **Lombok:** incluir colecciones LAZY en `@ToString` o en `@EqualsAndHashCode` (recursión/performance).
- **Paginación:** permitir `offset < 0` o `limit <= 0`. Definir contrato y validar.
- **Consistencia de URL:** usar URL/credenciales distintas entre el cargador JDBC y JPA (terminás con dos bases distintas).
- **Ordenamiento:** omitir `order by` en búsquedas textuales y sorprenderse por resultados inestables.
- **Modo H2:** olvidar `MODE=PostgreSQL` (o el modo elegido) y toparse con diferencias de tipos/escape.