Universidad ICESI

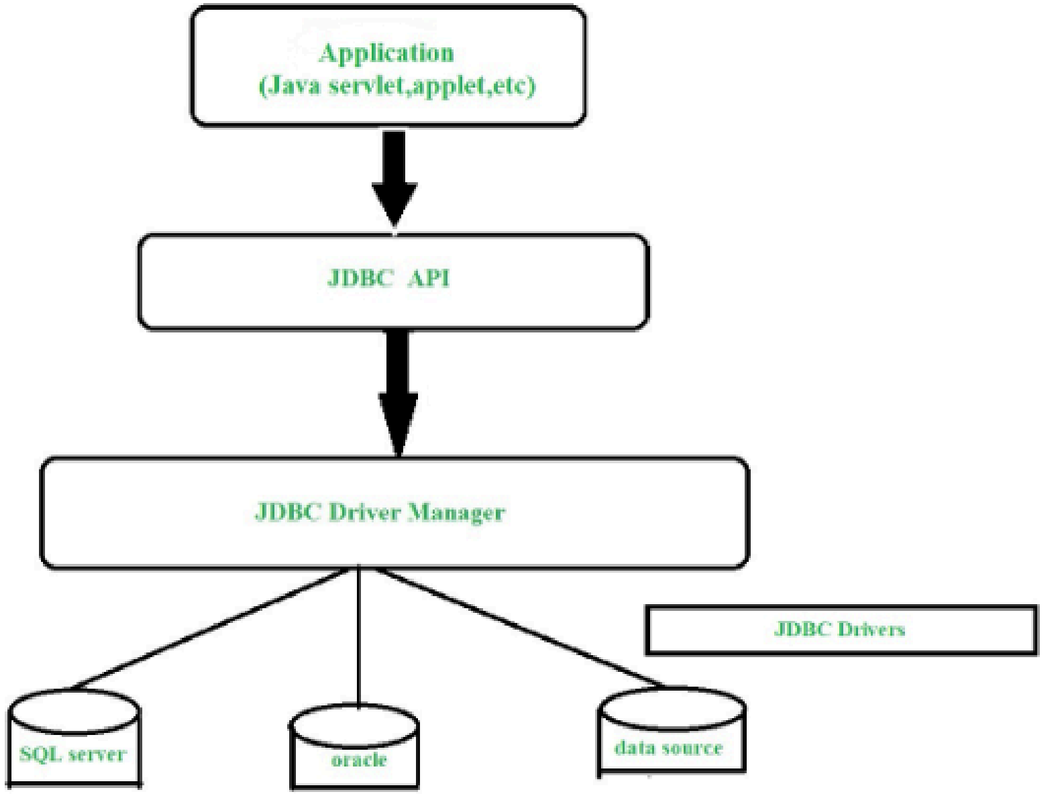
45 años

Spring Data

Computación en Internet II
2025-1

JDBC

JDBC es una API de Java que permite a las aplicaciones conectarse e interactuar con bases de datos mediante SQL.



JDBC

https://www.geeksforgeeks.org/jdbc-tutorial/

```java
import java.sql.*;

// Main class to illustrate demo of JDBC
class GFG {

    // Main driver method
    public static void main(String[] args) throws Exception
    {

        // Loading and registering drivers
        // Optional from JDBC version 4.0
        Class.forName("oracle.jdbc.OracleDriver");

        // Step 2:Establishing a connection
        Connection con = DriverManager(
            "jdbc:oracle:thin:@localhost:1521:XE",
            "username", "password");

        // Step 3: Creating statement
        Statement st = con.createStatement();

        // Step 4: Executing the query and storing the
        // result
        ResultSet rs = st.executeQuery(
            "select * from Students where Marks >= 70");

        // Step 5: Processing the results
        while (rs.next()) {
            System.out.println(rs.getString("students"));
            System.out.println(rs.getInt("marks"));
        }

        // Step 6: Closing the connections
        // using close() method to release memory resources
        con.close();

      // Display message for successful execution of program
      System.out.println("Steps in Setting Up of JDBC");
    }
}
```

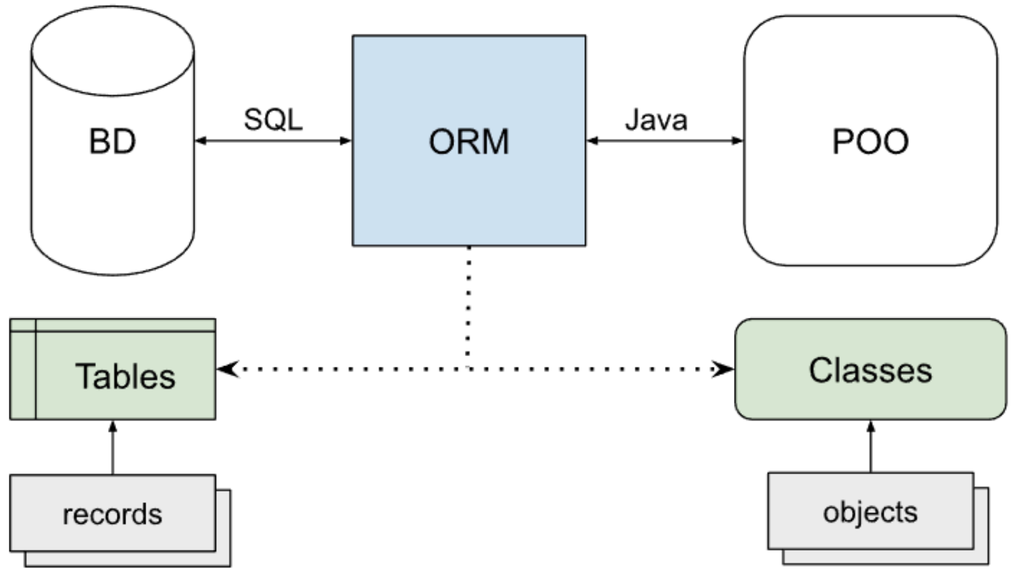
https://www.geeksforgeeks.org/jdbc-tutorial/

📌 Desventajas de usar JDBC directamente:

- Código repetitivo : Se deben escribir manualmente las consultas SQL y el manejo de conexiones.
- Manejo manual de transacciones: Debemos asegurarnos de cerrar conexiones y manejar commits/rollbacks.
- Fuerte acoplamiento con la base de datos: El código SQL está incrustado en la aplicación, dificultando cambios de motor de base de datos.
- Solución: Usar ORM (Object-Relational Mapping)
- ORM permite trabajar con bases de datos a través de objetos, eliminando la necesidad de escribir SQL directamente.

ORM (Object-Relational Mapping)

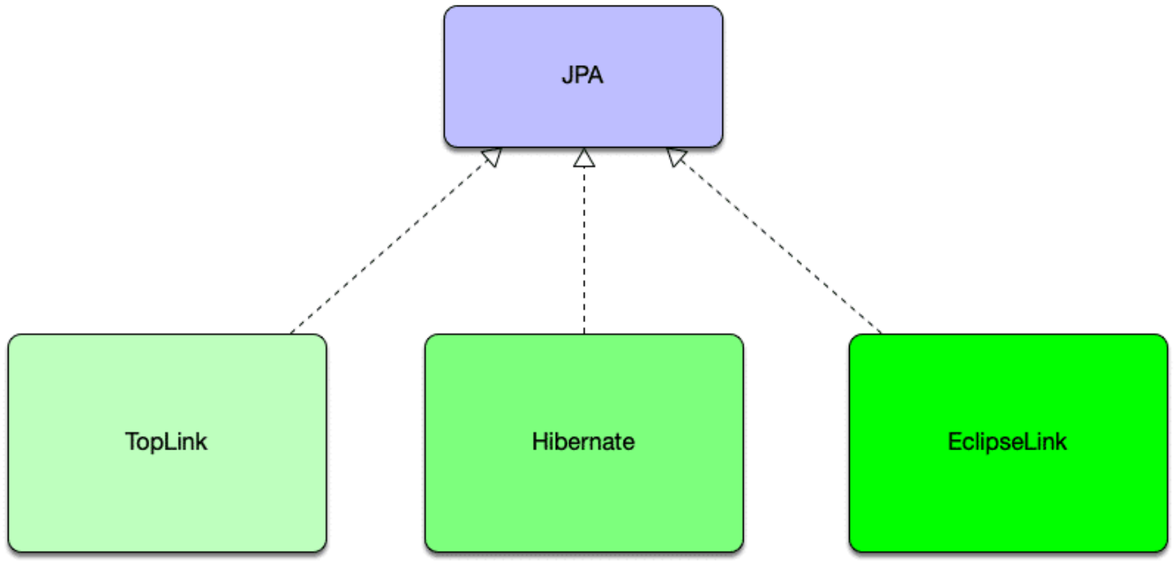
El mapeo objeto-relacional (ORM) es una técnica que permite interactuar con bases de datos relacionales utilizando objetos en lugar de sentencias SQL directas.



ORM

JPA

JPA (Jakarta Persistence API) es una especificación de Java que define una forma estándar de trabajar con bases de datos relacionales mediante el mapeo objeto-relacional (ORM). No es una implementación, sino un conjunto de reglas que deben seguir las librerías que lo implementan.



🔹 Ventajas de usar JPA

✅ Abstracción del SQL: No es necesario escribir consultas SQL manualmente para operaciones básicas.
✅ Portabilidad: Puede trabajar con distintos motores de bases de datos sin cambiar código.
✅ Optimización automática: Uso de caché, lazy loading, y manejo eficiente de consultas.
✅ Facilidad de mantenimiento: Código más limpio y orientado a objetos.

JPA

📌 ¿Qué es Association Mapping?

🔹 En bases de datos relacionales, las entidades están relacionadas mediante claves foráneas.
🔹 JPA permiten mapear estas relaciones entre objetos Java.
🔹 Se pueden definir asociaciones unidireccionales o bidireccionales.

📌 Tipos de relaciones en JPA:

✔️ @OneToOne → Relación uno a uno.
✔️ @OneToMany → Relación uno a muchos.
✔️ @ManyToOne → Relación muchos a uno.
✔️ @ManyToMany → Relación muchos a muchos.

JPA

https://www.arquitecturajava.com/jpa-vs-hibernate/

Annotations	Description
@Entity	Specifies that the class is an entity and is mapped to a database table.
@Table	Specifies the details of the table to which the entity is mapped.
@Id	Specifies the primary key attribute of the entity.
@GeneratedValue	Specifies the generation strategy for the values of the primary key.
@Column	Specifies the details of the column to which an attribute is mapped.
@OneToOne	Specifies a one-to-one relationship between two entities.
@OneToMany	Specifies a one-to-many relationship between two entities.
@ManyToOne	Specifies a many-to-one relationship between two entities.
@ManyToMany	Specifies a many-to-many relationship between two entities.

JPA

https://medium.com/@lahiruchandika/hibernate-jpa-annotations-2bbdad6cf456

Annotations	Description
@JoinColumn	Specifies the details of the join column in a relationship.
@Transient	Specifies that an attribute should not be persisted to the database.
@Temporal	Specifies the mapping of a temporal (date or time) attribute.
@Lob	Specifies the mapping of a large object (CLOB or BLOB) attribute.

JPA

https://medium.com/@lahiruchandika/hibernate-jpa-annotations-2bbdad6cf456

Universidad ICESI

45 años

```java
@Entity
public class Department {
    @Id @GeneratedValue
    private Long id;
    private String name;

    @OneToMany(mappedBy = "department") // Un departamento tiene muchos empleados
    private List<Employee> employees;
}

@Entity
public class Employee {
    @Id @GeneratedValue
    private Long id;
    private String name;

    @ManyToOne
    @JoinColumn(name = "department_id") // Clave foránea en la tabla employee
    private Department department;
}
```

JPA

Universidad ICESI

45 años

```java
@Entity
public class Student {
    @Id @GeneratedValue
    private Long id;
    private String name;

    @ManyToMany
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    private List<Course> courses;
}

@Entity
public class Course {
    @Id @GeneratedValue
    private Long id;
    private String title;

    @ManyToMany(mappedBy = "courses")
    private List<Student> students;
}
```

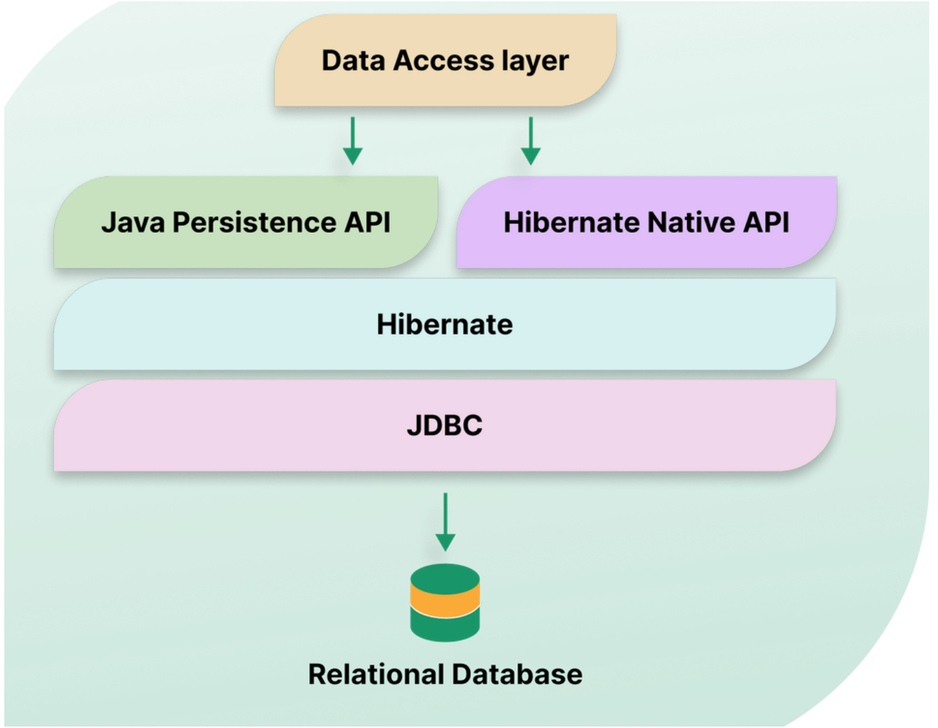
JPA

Hibernate

Hibernate es un framework de mapeo objeto-relacional (ORM) para Java que facilita la persistencia de datos en bases de datos relacionales. Proporciona una abstracción sobre JDBC y permite trabajar con bases de datos utilizando objetos de Java sin escribir SQL manualmente.

✅ Ventajas de Hibernate:

- Independencia de base de datos: Funciona con MySQL, PostgreSQL, Oracle, etc.
- Automatiza las consultas SQL: Se genera SQL de manera transparente.
- Manejo de transacciones: Soporta transacciones ACID de forma nativa.
- Optimización de consultas: Usa caché y lazy loading para mejorar el rendimiento.
- Soporte para relaciones: Manejo de asociaciones como @OneToMany, @ManyToOne, @ManyToMany.



Hibernate

https://www.arquitecturajava.com/jpa-vs-hibernate/

La misión de Spring Data es proporcionar un modelo de programación familiar y coherente basado en Spring para el acceso a datos, manteniendo al mismo tiempo las características especiales del sistema de almacenamiento subyacente.

Spring Data

Facilita el uso de tecnologías de acceso a datos, bases de datos relacionales y no relacionales, frameworks de procesamiento distribuido (map-reduce) y servicios de datos en la nube.

Descripción

Spring Data es un proyecto paraguas que agrupa múltiples subproyectos, cada uno específico para una base de datos determinada. Estos proyectos se desarrollan en colaboración con diversas empresas y desarrolladores que impulsan estas innovadoras tecnologías.

Descripción

- Potentes abstracciones de repositorio y mapeo de objetos personalizados.
- Derivación dinámica de consultas a partir de los nombres de los métodos del repositorio.
- Clases base de dominio con propiedades básicas predefinidas.
- Soporte para auditoría transparente (fecha de creación y última modificación).

Características

- Posibilidad de integrar código de repositorio personalizado.
- Fácil integración con Spring mediante JavaConfig y espacios de nombres XML personalizados.
- Integración avanzada con controladores de Spring MVC.
- Soporte experimental para persistencia en múltiples almacenes de datos.

Características

Universidad ICESI

45 años

Módulos

- Spring Data Commons - Conceptos centrales de Spring que sustentan cada módulo de Spring Data.
- Spring Data JDBC - Soporte de repositorios Spring Data para JDBC.
- Spring Data JPA - Soporte de repositorios Spring Data para JPA.
- Spring Data LDAP - Soporte de repositorios Spring Data para Spring LDAP.
- Spring Data MongoDB - Soporte basado en Spring para objetos-documento y repositorios en MongoDB.

Universidad ICESI

45 años

Extras

- Spring Data R2DBC - Soporte de repositorios Spring Data para R2DBC.
- Spring Data KeyValue - Repositorios basados en mapas y SPIs para construir módulos de Spring Data para almacenes clave-valor.
- Spring Data Redis - Configuración y acceso sencillo a Redis desde aplicaciones Spring.
- Spring Data REST - Expone repositorios de Spring Data como recursos RESTful impulsados por hipermedios.
- Spring Data para Apache Cassandra - Configuración y acceso fácil a Apache Cassandra para aplicaciones Spring de gran escala y alta disponibilidad.
- Spring Data para Apache Geode - Configuración y acceso fácil a Apache Geode para aplicaciones Spring de baja latencia y alta consistencia.

El proyecto Spring Data Commons aplica los conceptos fundamentales de Spring para el desarrollo de soluciones que utilizan múltiples bases de datos, tanto relacionales como no relacionales.

Objetivos principales:

- ✓ Potentes abstracciones de repositorio y mapeo de objetos personalizados
- ✓ Soporte para persistencia en múltiples almacenes de datos
- ✓ Generación dinámica de consultas a partir de nombres de métodos
- ✓ Clases base de dominio con propiedades básicas predefinidas
- ✓ Soporte para auditoría transparente (fecha de creación y última modificación)
- ✓ Posibilidad de integrar código de repositorio personalizado
- ✓ Fácil integración con Spring mediante namespaces personalizados

Spring Data Commons

https://docs.spring.io/spring-data/commons/reference/repositories/core-concepts.html

El objetivo de la abstracción de repositorios en Spring Data es reducir significativamente la cantidad de código repetitivo necesario para implementar capas de acceso a datos a diversas bases de datos.

La interfaz central en la abstracción de repositorios de Spring Data es **Repository**.

✓ Define la clase de dominio a gestionar y el tipo de su identificador como argumentos genéricos.

✓ Actúa como una interfaz marcador, ayudando a identificar otras interfaces que la extienden.

✓ No proporciona métodos por sí misma, pero sirve como base para otras interfaces más especializadas como **CrudRepository**, **PagingAndSortingRepository** y **JpaRepository**.

CrudRepository Interface

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {

  <S extends T> S save(S entity);      ❶

  Optional<T> findById(ID primaryKey); ❷

  Iterable<T> findAll();               ❸

  long count();                        ❹

  void delete(T entity);               ❺

  boolean existsById(ID primaryKey);   ❻

  // … more functionality omitted.
}
```

Query Methods

El proxy de repositorio en Spring Data tiene dos formas de generar una:

1️⃣ Derivando la consulta directamente del nombre del método, Spring analiza el nombre del método y genera automáticamente la consulta correspondiente.

2️⃣ Usando una consulta definida manualmente, Puedes escribir la consulta explícitamente con JPQL, SQL nativo o anotaciones como **@Query.**

Query Methods

Keyword	Sample	JPQL snippet
`Distinct`	`findDistinctByLastnameAndFirstname`	`select distinct … where x.lastname = ?1 and x.firstname = ?2`
`And`	`findByLastnameAndFirstname`	`… where x.lastname = ?1 and x.firstname = ?2`
`Or`	`findByLastnameOrFirstname`	`… where x.lastname = ?1 or x.firstname = ?2`
`Is`, `Equals`	`findByFirstname`, `findByFirstnameIs`, `findByFirstnameEquals`	`… where x.firstname = ?1` (or `… where x.firstname IS NULL` if the argument is `null`)
`Between`	`findByStartDateBetween`	`… where x.startDate between ?1 and ?2`
`LessThan`	`findByAgeLessThan`	`… where x.age < ?1`
`LessThanEqual`	`findByAgeLessThanEqual`	`… where x.age <= ?1`
`GreaterThan`	`findByAgeGreaterThan`	`… where x.age > ?1`
`GreaterThanEqual`	`findByAgeGreaterThanEqual`	`… where x.age >= ?1`

Query Methods

`After`	`findByStartDateAfter`	`… where x.startDate > ?1`
`Before`	`findByStartDateBefore`	`… where x.startDate < ?1`
`IsNull`, `Null`	`findByAge(Is)Null`	`… where x.age is null`
`IsNotNull`, `NotNull`	`findByAge(Is)NotNull`	`… where x.age is not null`
`Like`	`findByFirstnameLike`	`… where x.firstname like ?1`
`NotLike`	`findByFirstnameNotLike`	`… where x.firstname not like ?1`
`StartingWith`	`findByFirstnameStartingWith`	`… where x.firstname like ?1` (parameter bound with appended `%`)
`EndingWith`	`findByFirstnameEndingWith`	`… where x.firstname like ?1` (parameter bound with prepended `%`)

Query Methods

`Containing`	`findByFirstnameContaining`	`… where x.firstname like ?1` (parameter bound wrapped in `%`)
`OrderBy`	`findByAgeOrderByLastnameDesc`	`… where x.age = ?1 order by x.lastname desc`
`Not`	`findByLastnameNot`	`… where x.lastname <> ?1`
`In`	`findByAgeIn(Collection<Age> ages)`	`… where x.age in ?1`
`NotIn`	`findByAgeNotIn(Collection<Age> ages)`	`… where x.age not in ?1`
`True`	`findByActiveTrue()`	`… where x.active = true`
`False`	`findByActiveFalse()`	`… where x.active = false`
`IgnoreCase`	`findByFirstnameIgnoreCase`	`… where UPPER(x.firstname) = UPPER(?1)`

```
Page<User> findByLastname(String lastname, Pageable pageable);

Slice<User> findByLastname(String lastname, Pageable pageable);

List<User> findByLastname(String lastname, Sort sort);

List<User> findByLastname(String lastname, Sort sort, Limit limit);

List<User> findByLastname(String lastname, Pageable pageable);
```

Declare a query method by using @Query

```
interface UserRepository extends CrudRepository<User, Long> {

  @Query("select firstName, lastName from User u where u.emailAddress = :email")
  User findByEmailAddress(@Param("email") String email);
}
```

Universidad ICESI

Transacciones

El concepto ACID es un conjunto de propiedades que garantizan la confiabilidad de las transacciones en bases de datos. Se utiliza para asegurar la integridad y consistencia de los datos, incluso en caso de fallos o accesos concurrentes.

- **Atomicidad (Atomicity):** Una transacción es atómica, lo que significa que se ejecuta completamente o no se ejecuta en absoluto. Si todas las operaciones dentro de una transacción se completan, los cambios se guardan. Si alguna operación falla, se revierte todo con un ROLLBACK.
- **Consistencia (Consistency):** Una transacción debe llevar la base de datos de un estado válido a otro válido, respetando reglas de integridad. No se deben violar restricciones de clave primaria, claves foráneas, o reglas de negocio y si una transacción deja la base de datos en un estado inválido, se revierte.
- **Aislamiento (Isolation):** Asegura que múltiples transacciones ejecutándose al mismo tiempo no interfieran entre sí. Los problemas que resuelve son: Lecturas sucias, Leer datos que aún no han sido confirmados. Lecturas no repetibles, leer datos que cambian durante una transacción. Lectura fantasma, nuevas filas aparecen en consultas repetidas dentro de una transacción.
- **Durabilidad (Durability):** Una vez que una transacción ha sido confirmada (COMMIT), sus cambios se guardan de forma permanente, incluso si ocurre un fallo del sistema.

ACID

Los métodos de las instancias de CrudRepository son transaccionales por defecto.

🔹 Para operaciones de lectura (findBy, count, exists, etc.)
Se configura con @Transactional(readOnly = true), lo que optimiza el rendimiento evitando bloqueos innecesarios en la base de datos.

🔹 Para operaciones de escritura (save, delete, update, etc.)
Se usa @Transactional sin parámetros, aplicando la configuración de transacciones por defecto de Spring.

```
interface UserRepository extends CrudRepository<User, Long> {

  @Override
  @Transactional(timeout = 10)
  List<User> findAll();

  // Further query method declarations
}
```

Si necesitas cambiar la configuración de transacciones para un método específico, solo tienes que redeclararlo en la interfaz del repositorio con la anotación @Transactional personalizada.

Además de configurar transacciones a nivel de repositorio, puedes manejar límites transaccionales a través de una clase de servicio. Esto es útil cuando una operación involucra múltiples repositorios o lógica de negocio más compleja.

✓ Agrupa múltiples operaciones en una sola transacción.

✓ Permite rollback automático en caso de error.

✓ Evita la repetición de lógica transaccional en cada repositorio.

💡 Recomendado: Siempre que haya lógica de negocio que involucre más de un Repository, usa un servicio (@Service) con @Transactional en lugar de anotar cada método en el repositorio.

```java
@Service
public class UserManagementImpl implements UserManagement {

  private final UserRepository userRepository;
  private final RoleRepository roleRepository;

  UserManagementImpl(UserRepository userRepository,
    RoleRepository roleRepository) {
    this.userRepository = userRepository;
    this.roleRepository = roleRepository;
  }

  @Transactional
  public void addRoleToAllUsers(String roleName) {

    Role role = roleRepository.findByName(roleName);

    for (User user : userRepository.findAll()) {
      user.addRole(role);
      userRepository.save(user);
    }
}
```