

Spring Data

Spring Data

- Spring Data es un proyecto general de código abierto que proporciona una abstracción común para interactuar con un “almacenamiento de datos” conservando las características especiales de su modelo de base de datos.
- Hay más de una docena de proyectos “oficiales” de Spring Data que cubren variedad de base de datos populares, incluidos los modelos de datos relacional y no SQL.
- Por ejemplo, MongoDB, Neo4j, Redis, Elasticsearch, Cassandra, MySQL, PostgreSQL, entre otros.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
  </dependency>
</dependencies>
```

Spring Data - Características

- Soporte para las persistencias en diferentes bases de datos
- Abstracción de mapeo de objetos personalizados y basadas en repository
- Consultas dinámicas basadas en nombres de métodos

Spring Data - Características

- Soporte para las persistencias en diferentes bases de datos
- Abstracción de mapeo de objetos personalizados y basadas en repository
- Consultas dinámicas basadas en nombres de métodos

Configurar Datasource para MySQL

```
<dependency>  
    <groupId>com.mysql</groupId>  
    <artifactId>mysql-connector-j</artifactId>  
    <scope>runtime</scope>  
</dependency>
```

```
spring.datasource.url=jdbc:mysql://localhost:3306/test  
spring.datasource.username= root  
spring.datasource.password=  
spring.datasource.driver-class-name= com.mysql.cj.jdbc.Driver  
spring.jpa.database-platform = org.hibernate.dialect.MySQL8Dialect  
spring.jpa.show-sql=true
```

Repositorio

- La forma principal para acceder a datos en una aplicación Spring Data es a través de repositorios
- Spring posee la anotación, `@Repository`, que, además de marcar una clase como `@Component`, le indica a Spring que el bean puede generar excepciones específicas de tecnología de persistencia, así se unifican distintas excepciones (por ej violaciones de restricciones) en una única.
- SpringData va más allá y proporciona implementaciones de interfaces de acuerdo con las convenciones y que pueden ser extendidas por el usuario.

Qué es un ORM?

- ORM significa Object-Relational Mapping.
- Es una técnica que permite mapear objetos de una aplicación con tablas de una base de datos relacional.
- El ORM automatiza la conversión de datos entre los objetos del código (POJOs) y las tablas de la base de datos.
- Facilita el manejo de bases de datos desde el código sin necesidad de escribir SQL manual.
- Proporciona un enfoque orientado a objetos para interactuar con la base de datos.
- Los ORMs suelen gestionar operaciones como CRUD (Create, Read, Update, Delete) de forma automática.

Hibernate

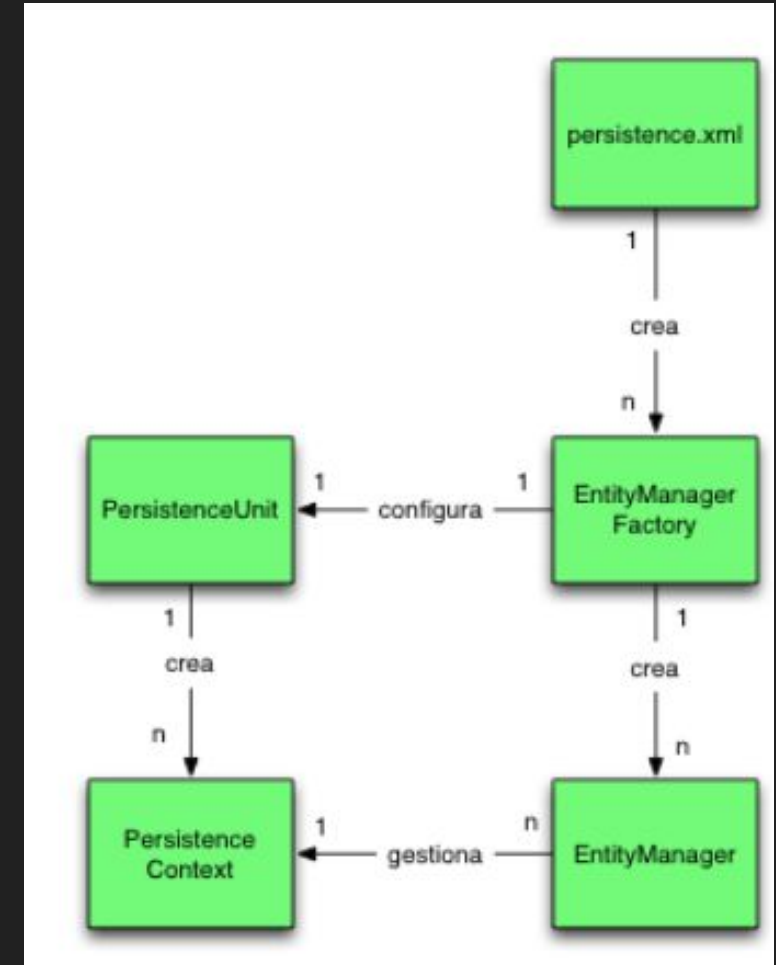


Hibernate es un framework de mapeo objeto-relacional (ORM) que facilita la interacción entre la aplicación Java y la base de datos. Hibernate permite mapear clases Java a tablas en la base de datos, lo que reduce el código SQL que debes escribir.

Ejemplo: En una aplicación Spring Boot, puedes usar Hibernate para gestionar la persistencia de datos. Spring Data JPA, que integra Hibernate, es una extensión popular para esto.

EntityManager

- En JPA el elemento central que administra todo el trabajo entre el modelo de objetos y una base de datos relacional, se llama EntityManager y se encuentra definido por la interface `jakarta.persistence.EntityManager` (antes Javax)
- Realiza operaciones CRUD o consultas complejas sobre bases de datos
- Primero debemos obtener un elemento EntityManagerFactory, el cual lee el archivo de configuración apropiado y nos retorna una instancia del EntityManager que maneja una unidad de persistencia específica




EntityManager en Spring

```
@Service
public class UnidadServiceImp implements UnidadService {

    @Autowired
    EntityManager em;

    @Override
    @Transactional
    public Unidad guardar(Unidad u) {
        em.persist(u);
        em.flush();
        em.refresh(u);
        return u;
    }
}
```



Marcamos a Hibernate para que automáticamente inicie una transacción con el método y la "comitee" al finalizar

Spring se encarga de crear por nosotros el entityManager.

JPA Repository

- Cada tabla, en un sentido amplio se corresponde con una clase.
- Se basa en clases simples (POJO) y en almacenar su “estado” en una DB relacional.
- El estado de un objeto, según se define en la POO, es el valor de todos los atributos que tiene una instancia en un momento dado
- Cuando hablamos de mapeo de objetos a bases de datos relacionales, objetos persistentes, o consulta de objetos, deberíamos usar el término “entidad”
- Entidades son objetos que temporalmente, y en un lapso breve, se encuentran en memoria, pero persistentemente almacenan su estado en una base de datos relacional

JPA Repository

- Cuando Spring Boot ejecuta su configuración automática y descubre que tiene Spring Data JPA, configura un datasource datos por defecto (si no hay ninguno definido) y el proveedor JPA (por defecto usa Hibernate). Luego habilita los repositorios (usando la configuración de `@EnableJpaRepositories`).

- ¿Qué métodos tenemos disponibles?

- `S save(S entity) / saveAll(Iterable<S> entities)`
- `Optional<T> findById(ID id)`
- `Iterable<T> findAll() / findAllById(Iterable<ID> ids)`
- `Page<T> findAll(Pageable pageable)`
- `Optional<S> findOne(Example<S> example)`
- `delete(T entity) / deleteAll() deleteAll(Iterable) /deleteById(id)`
- `boolean existsById(ID id)`

¿Cómo crear un JPA Repository?

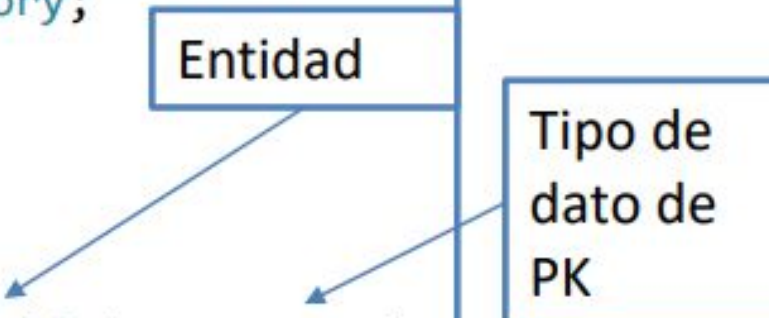
```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import dan.tp2021.productos.domain.Unidad;

@Repository
public interface UnidadRepository extends JpaRepository<Unidad, Integer>{

}
```

Entidad



Tipo de
dato de
PK

¿Cómo definir una entidad?

- Para que cualquier clase sea reconocida como entidad tiene que cumplir 2 condiciones: Tener la anotación `@Entity` y tener un atributo definido como clave primaria (`@Id`)

```
import javax.persistence.Entity;
import javax.persistence.Id;

/**
 * Clase que permite representar una unidad de medida para un material.
 * Puede ser m2, m3, litros, piezas, unidades, kilos, bolsas.
 * @author MD
 */
@Entity
public class Unidad {

    @Id
    private Integer id;
    private String descripcion;
```

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.UniqueConstraint;
```

@Table especifica nombre de tabla donde la entidad es almacenada. Por defecto es el nombre de la Clase.

@UniqueConstraint para indicar una restricción de unicidad

```
@Entity
@Table(name="PRD_UNIDAD", schema="MS_PRD",
       uniqueConstraints = {@UniqueConstraint(columnNames={"ID_UNIDAD", "DESCRIPCION"})})
public class Unidad {
```

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

Para indicar que el ID lo generará la base de datos se utiliza `@GeneratedValue` y se debe describir cual será la estrategia de generación

@Column define el nombre de la columna de la tabla donde es almacenado el atributo. Por defecto es igual al nombre del campo de la tabla.

Estrategias de generación de ID

- Autonumérico → GenerationType.IDENTITY
- Secuencias → GenerationType.SEQUENCE
- Tabla de ID → GenerationType.TABLE
- AUTO: permite que proveedor de persistencia defina la mejor forma de generar las claves primarias.
(alguna de las 3 anteriores)

```
@SequenceGenerator(name="Unidad_Gen", sequenceName="MS_UNIDAD_SEQ")  
@Id @GeneratedValue(generator="Unidad_Gen")  
private int getId;
```

Atributos de la anotación column


- `name()` especifica el nombre de la columna, Por defecto es el nombre de la propiedad de la clase.
- `table()` es utilizado cuando se realiza mapeo de una clase, en más de una tabla.
- `unique()` y `nullable()` define las restricciones que se implementarán sobre la columna, `unique` agrega una clave única sobre la columna, y `nullable` permite asignar valores nulos.
- `insertable()` y `updatable()` indican si la columna debe ser incluida en los SQL de `INSERT` o `UPDATE` generados por el Entity Manager.
- `@Column(nullable = false)` es usado para indicarle a la base de datos que no acepte persistir datos que no cumplan con esa restricción

OneToOne unidireccional

- Definimos el mapeo y la columna de union

```
public class Clase1 {  
    private Integer id;  
    private String desc;  
    @OneToOne  
    @JoinColumn(name = "CLASE2_ID")  
    private Clase2 clase2;  
}
```

CLASE1		
ID	DESC	CLASE2_ID
1	A	2
2	B	3



```
public class Clase2 {  
    private Integer id;  
    private String desc;  
}
```

CLASE2	
ID	DESC
1	XZ
2	JK
3	NY

OneToOne bidireccional

- Cuando el mapeo es bidireccional debemos elegir en que tabla poner la columna de UNION

```
public class Clase1 {  
    private Integer id;  
    private String desc;  
    @OneToOne(mappedBy="varClase1")  
    private Clase2 clase2;  
}
```

```
public class Clase2 {  
    private Integer id;  
    private String desc;  
    @OneToOne  
    @JoinColumn(name = "CLASE1_ID")  
    private Clase1 varClase1;  
}
```

CLASE1	
ID	DESC
1	A
2	B
3	C

CLASE2		
ID	DESC	CLASE1_ID
1	XZ	1
2	JK	2
3	NY	3

UK

La tabla que tiene la FK es la "dueña" de la relación.

La tabla que no es dueña, indica que la dueña es la otra usando el atributo "mappedBy".

ManyToOne unidireccional

ManyToOne UniDireccional

```
public class Clase1 {  
    private Integer id;  
    private String desc;  
    @ManyToOne  
    @JoinColumn(name = "CLASE2_ID")  
    private Clase2 clase2;  
}
```

CLASE1		
ID	DESC	CLASE2_ID
1	A	2
2	B	3
3	C	2

```
public class Clase2 {  
    private Integer id;  
    private String desc;  
}
```

CLASE2	
ID	DESC
1	XZ
2	JK
3	NY

La relación es UNIDIRECCIONAL.

No hay UK, distintas filas de la Clase1 pueden tener relación con la misma fila de Clase2

La tabla que no es dueña, indica que la dueña es la otra usando el atributo "mappedBy".

OneToMany unidireccional

OneToMany Unidireccional

```
public class Clase1 {  
    private Integer id;  
    private String desc;  
}
```

CLASE1		
ID	DESC	CLASE2_ID
1	A	2
2	B	3
3	C	2

```
public class Clase2 {  
    private Integer id;  
    private String desc;  
    @OneToMany(mappedBy="clase2")  
    @JoinColumn(name = "CLASE2_ID",  
        referencedColumnName="ID")  
    private List<Clase1> lista;  
}
```

CLASE2	
ID	DESC
1	XZ
2	JK
3	NY

Le indicamos cual es la columna en la tabla del lado **MUCHOS** que guarda la FK y a que columna de la propia tabla referencia

A nivel tablas no hay cambios.
El dueño de la relación es Clase2. La relación se va a guardar (se guarda la CLAVE) cuando se guarde la tabla dueña. EN este caso se actualiza Clase2 y Clase1.

ManyToOne / OneToMany bidireccional

ManyToOne Bidireccional (→ OneToMany)

```
public class Clase1 {  
    private Integer id;  
    private String desc;  
    @ManyToOne  
    @JoinColumn(name = "CLASE2_ID")  
    private Clase2 clase2;  
}
```

CLASE1		
ID	DESC	CLASE2_ID
1	A	2
2	B	3
3	C	2

```
public class Clase2 {  
    private Integer id;  
    private String desc;  
    @OneToMany(mappedBy="clase2")  
    private List<Clase1> lista;  
}
```

CLASE2	
ID	DESC
1	XZ
2	JK
3	NY

La relación BIDIRECCIONAL,
indica que un lado tiene solo
una fila y el otro muchas.

A nivel tablas no hay cambios.

El dueño de la relación es Clase1. La relación se va a guardar (se guarda la CLAVE)
cuando se guarde la tabla dueña.

Clase2 se denomina "ladoInverso"

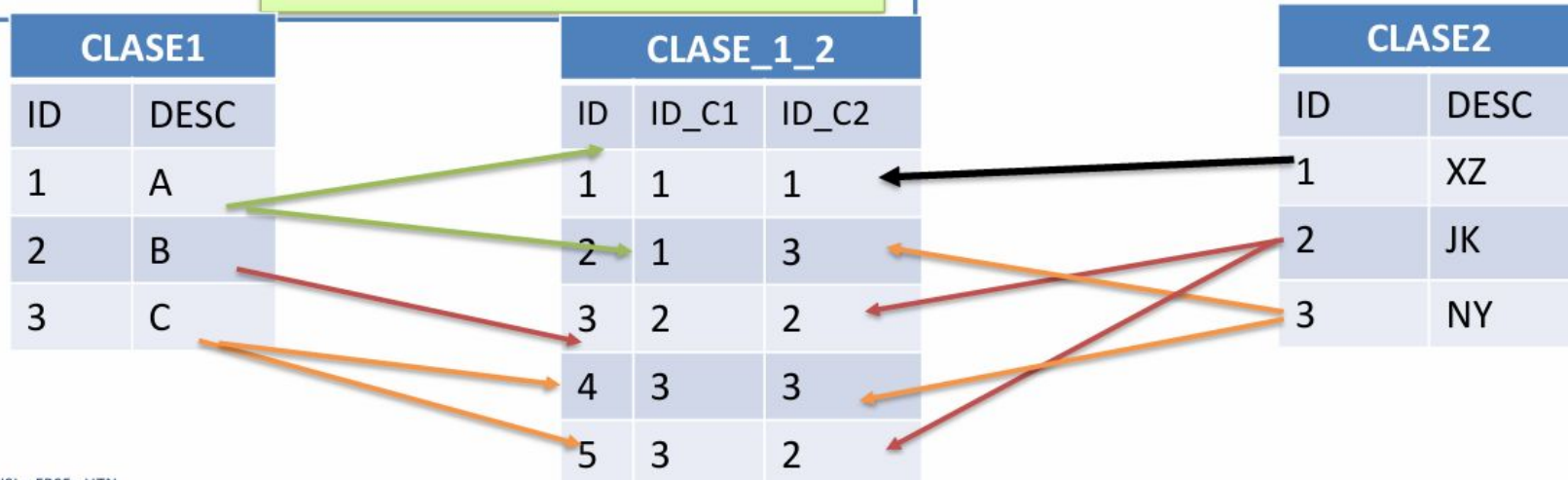
ManyToMany

Muchos a Muchos

```
public class Clase1 {  
    private Integer id;  
    private String desc;  
    @ManyToMany  
    @JoinTable(  
        name = "CLASE_1_2",  
        joinColumns = @JoinColumn(name = "ID_C1"),  
        inverseJoinColumns = @JoinColumn(name = "ID_C2"))  
    private List<Clase2> lista2;  
}
```

Necesitamos una tabla de union

```
public class Clase2 {  
    private Integer id;  
    private String desc;  
    @ManyToMany(mappedBy = "lista2")  
    private List<Clase1> lista1;  
}
```



Eager / Lazy

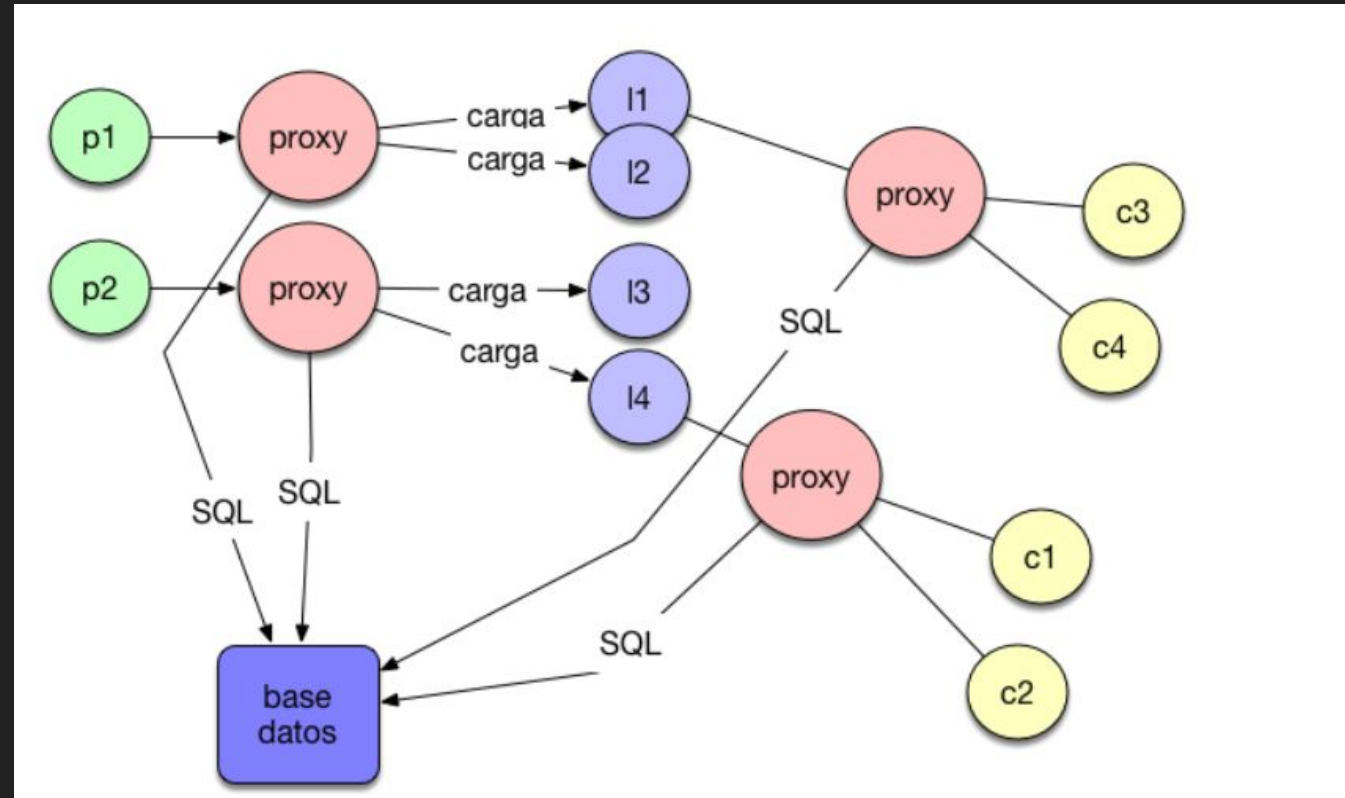
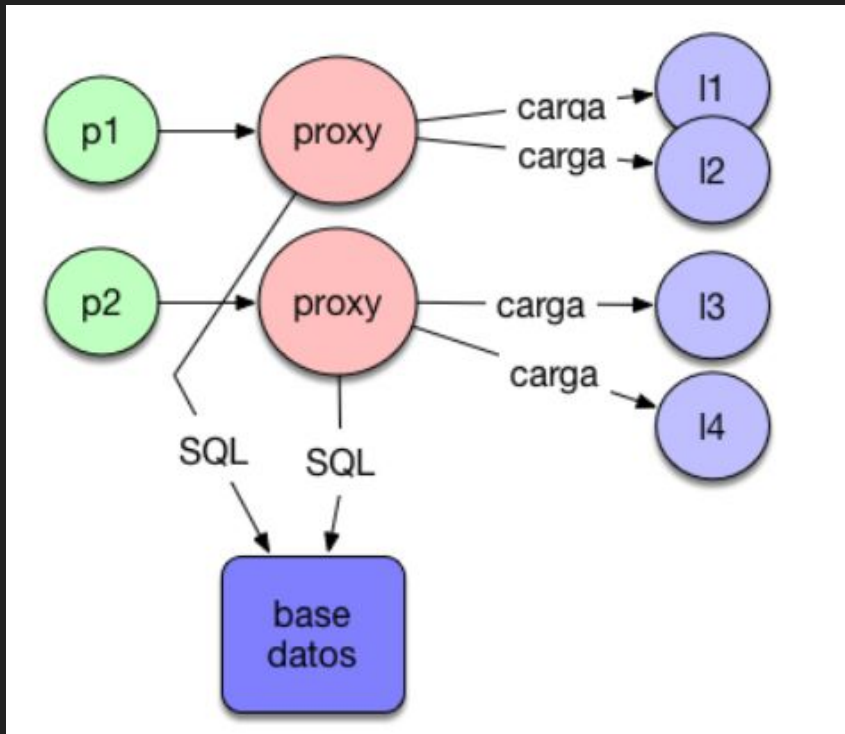
- Todos los atributos de una ENTIDAD tienen por defecto una anotación “@Basic” la cual tiene un atributo “fetch” que puede tomar dos valores posibles (eager o lazy), configurado con el valor “Eager”.
- En tanto para las relaciones la estrategia depende del tipo de relación.

- @One**ToMany** por defecto es `javax.persistence.FetchType.LAZY`
- @One**ToOne** por defecto es `javax.persistence.FetchType.EAGER`
- @Many**ToMany** por defecto es `javax.persistence.FetchType.LAZY`
- @Many**ToOne** por defecto es `javax.persistence.FetchType.EAGER`

Eager / Lazy

<https://www.arquitecturajava.com/jpa-lazy-fetching-proxies-y-rendimiento/>

- Cuando la carga es lazy, hibernate carga un objeto “proxy” de la entidad
- Es un objeto que tiene los mismos métodos pero no tiene datos, sino que cuando un método getter es invocado entonces el objeto proxy se encarga de buscar el resultado en la base de datos y retornarlos.



Eager / Lazy

- Cuando la carga es Eager se carga el objeto real de la base de datos.
- ¿Qué ocurre si una cadena de relaciones ManyToOne se expande con una profundidad 6?. Tengo que traer los 6 objetos reales a memoria y sus relaciones.
- Con `hibernate.max_fetch_deph` podemos setearle la profundidad máxima para el fetch para que a partir de esa profundidad traiga objetos proxis. No se recomienda ir más allá de 3 y si le colocamos 0 deshabilitamos la función.

En la nube debemos **minimizar todos los viajes a la base de datos.**

Soluciones:

- **controlar las relaciones**
- **Desnormalizar los esquemas**
- **Usar ORM para operaciones basicas y SQL Plano para relaciones complejas**
- **Usar otras bases de datos NO-SQL para algunas operaciones.**

Consultas derivadas

- Con lo que hemos visto hasta ahora es fácil derivar cuál debería ser la consulta SQL con solo mirar el nombre del método o atributo correspondiente en nuestro código.
- Spring Data JPA aprovecha esta idea en forma de una convención de nomenclatura de métodos
- Las consultas derivadas tienen dos partes, la **introducción** y el **criterio**.
- Por ejemplo **findByNombre**(String nombre)

Consultas derivadas ejemplos

Keyword	Sample
Distinct	<code>findDistinctBy</code> Lastname <code>And</code> Firstname
And	<code>findBy</code> Lastname <code>And</code> Firstname
Or	<code>findBy</code> Lastname <code>Or</code> Firstname
Is, Equals	<code>findBy</code> Firstname, <code>findBy</code> Firstname <code>Is</code> , <code>findBy</code> Firstname <code>Equals</code>
Between	<code>findBy</code> StartDate <code>Between</code>
LessThan	<code>findBy</code> Age <code>LessThan</code>
LessThanEqual	<code>findBy</code> Age <code>LessThanEqual</code>
GreaterThan	<code>findBy</code> Age <code>GreaterThan</code>
GreaterThanEqual	<code>findBy</code> Age <code>GreaterThanEqual</code>

Keyword	Sample
After	<code>findBy</code> StartDate <code>After</code>
Before	<code>findBy</code> StartDate <code>Before</code>
IsNull, Null	<code>findBy</code> Age <code>(Is) Null</code>
IsNotNull, NotNull	<code>findBy</code> Age <code>(Is) NotNull</code>
Like	<code>findBy</code> Firstname <code>Like</code>
NotLike	<code>findBy</code> Firstname <code>NotLike</code>
StartingWith	<code>findBy</code> Firstname <code>StartingWith</code>
EndingWith	<code>findBy</code> Firstname <code>EndingWith</code>
Containing	<code>findBy</code> Firstname <code>Containing</code>

Consultas derivadas ejemplos

Keyword	Sample
OrderBy	findBy Age OrderBy LastName Desc
Not	findBy LastName Not
In	findBy Age In (Collection <Age> ages)
NotIn	findBy Age NotIn (Collection <Age> ages)
True	findBy Active True ()
False	findBy Active False ()
IgnoreCase	findBy Firstname IgnoreCase

Keyword	Description
find...By, read...By, get...By, query...By, search...By, stream...By	General query method returning typically the repository type, a Collection or Streamable subtype or a result wrapper such as Page, GeoResults or any other store-specific result wrapper. Can be used as findBy..., findMyDomainTypeBy... or in combination with additional keywords.
exists...By	Exists projection, returning typically a boolean result.
count...By	Count projection returning a numeric result.
delete...By, remove...By	Delete query method returning either no result (void) or the delete count.
...First<number>..., ...Top<number>...	Limit the query results to the first <number> of results. This keyword can occur in any place of the subject between find (and the other keywords) and by. findFirst3ByAge() findTop3ByAge()
...Distinct...	Use a distinct query to return only unique results. Consult the store-specific documentation whether that feature is supported. This keyword can occur in any place of the subject between find (and the other keywords) and by.
OrderBy...	Specify a static sorting order followed by the property path and direction (e. g. OrderByFirstnameAscLastNameDesc).

Consulta JPQL

- Java Persistence Query Language (JPQL), es un lenguaje de consultas portable diseñado para combinar la sintaxis simple y la potencia semántica de SQL, con la expresividad de las expresiones orientadas a objetos.
- Las consultas escritas con este lenguaje pueden ser portables y compiladas a SQL para ser ejecutadas en la mayoría de los motores de base de datos relacionales.
- Ejemplos:
 - `SELECT c FROM Cliente C`
- Las “path expressions” permiten navegar a través de las relaciones de las entidades.
- Por ejemplo:
 - `SELECT e.usuario.tipoUsuario FROM Cliente e`
 - `SELECT c.obras FROM Cliente c`

Consulta JPQL JOIN

- Si queremos realizar un join entre dos tablas, empleado y telefono, por ejemplo, podemos hacer lo siguiente:
 - `SELECT t FROM Empleado e JOIN e.telefonos t`
- Las condiciones del JOIN se definen en el mapeo de la relación entre las entidades, no se necesitan criterios adicionales
- Siempre que utilizamos “path expression” se produce un JOIN implicito, por ejemplo, podemos hacer la consulta anterior de la siguiente forma
 - `SELECT e.telefonos FROM Empleado e`
- JPQL soporta OUTER JOIN, LEFT JOIN y RIGHT JOIN

JPQL WHERE AND HAVING

- JPQL soporta todos los operadores de SQL, como BETWEEN, LIKE, IN, NOT IN, etc
- JPQL agrega los operadores IS EMPTY y MEMBER OF
 - IS EMPTY nos permite saber si una colección está vacía
 - MEMBER OF nos permite saber si un elemento se encuentra en una colección
 - Por ejemplo
 - SELECT e FROM Empleados e WHERE e.proyectos IS EMPTY
 - SELECT e FROM Empleados e WHERE :p MEMBER OF e.proyectos

JPQL WHERE AND HAVING

- JPQL nos provee además una serie de funciones que se pueden utilizar en las cláusulas WHERE y HAVING
- La cláusula SIZE genera una subconsulta con un count

- ABS(number)
- MOD(number1, number2)
- SQRT(number)
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- LENGTH(string)
- LOWER(string)
- UPPER(string)
- SIZE(collection)

- SUBSTRING(string, start, end)
- TRIM([[LEADING|TRAILING|BOTH] [char] FROM] string)
- CONCAT(string1, string2)
- LOCATE(string1, string2 [, start])

Implementar JPQL en Spring

- Podemos utilizar JPQL mediante la anotación @Query

```
@Query("select u from User u where u.firstname like %?1")  
List<User> findByFirstnameEndsWith(String firstname);
```

- @Query nos permite también ejecutar consultas nativas

```
@Query(value = "SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?1", nativeQuery = true)  
User findByEmailAddress(String emailAddress);
```

Transacciones

- Grupo de operaciones que deben ser realizadas como una unidad
- Estas operaciones pueden ser sincrónicas o asincrónicas, y pueden involucrar varias tareas.
- Por ejemplo en una aplicación que transfiere dinero de una cuenta A a una cuenta B, desde la perspectiva de una aplicación externa, en ningún momento debe ser posible realizar una consulta donde el monto transferido pueda ser visto en ambas cuentas al mismo tiempo (o en ninguna)
- Solo cuando ambas operaciones, de debitar de una cuenta y acreditar en la otra, han sido finalizadas, los cambios deberán poder ser visibles desde otro contexto de la aplicación.

Transacciones

```
@Service
public class CuentaService {

    @Autowired
    private CuentaRepository cuentaRepository;

    @Transactional
    public void transferirFondos(Long cuentaOrigenId, Long cuentaDestinoId, Double monto) {
        // 1. Retirar dinero de la cuenta de origen
        Cuenta cuentaOrigen = cuentaRepository.findById(cuentaOrigenId).orElseThrow(() -> new IllegalArgumentException("Cuenta origen no encontrada"));
        cuentaOrigen.retirar(monto);

        // 2. Depositar dinero en la cuenta de destino
        Cuenta cuentaDestino = cuentaRepository.findById(cuentaDestinoId).orElseThrow(() -> new IllegalArgumentException("Cuenta destino no encontrada"));
        cuentaDestino.depositar(monto);

        // 3. Guardar ambas cuentas
        cuentaRepository.save(cuentaOrigen);
        cuentaRepository.save(cuentaDestino);
    }
}
```

Transacciones

- `@Transactional` tiene un modificador que es `readOnly`.
- Se utiliza cuando sabemos que las operaciones son de solo lectura y no modificaremos los resultados ya que nos permite optimizar el rendimiento de las consultas, debido a que Hibernate evita varias comprobaciones a la hora de ejecutar la consulta.

Transacciones

```
@Repository
@Transactional(readonly = true)
public interface MaterialRepository extends JpaRepository<Material, Integer> {

    @Query("update PRD_MATERIAL P "
+ " set P.PRECIO = P.PRECIO * (1+ :porc ) "
+ " where STOCK_ACTUAL >? :stkBase")
    @Modifying
    @Transactional(readonly = false)
    int actualizarPrecios(@Param("porc") Double porcentaje,
        @Param("stkBase") Integer stockBase);

    List<Material> findByPrecioGreatherThan(Double precio,Sort sort);
    List<Material> findByStockActual(Integer stockActual);
    List<Material> findByStockActualBetween(Integer minimo,Integer maximo);
    Page<Material> findByStockActualBetween(Integer minimo,Integer maximo,
        PageRequest page);
}
```