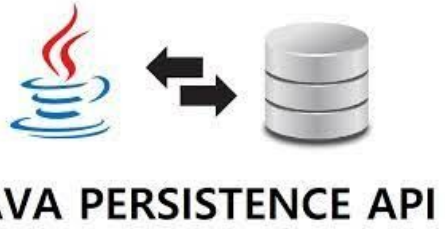


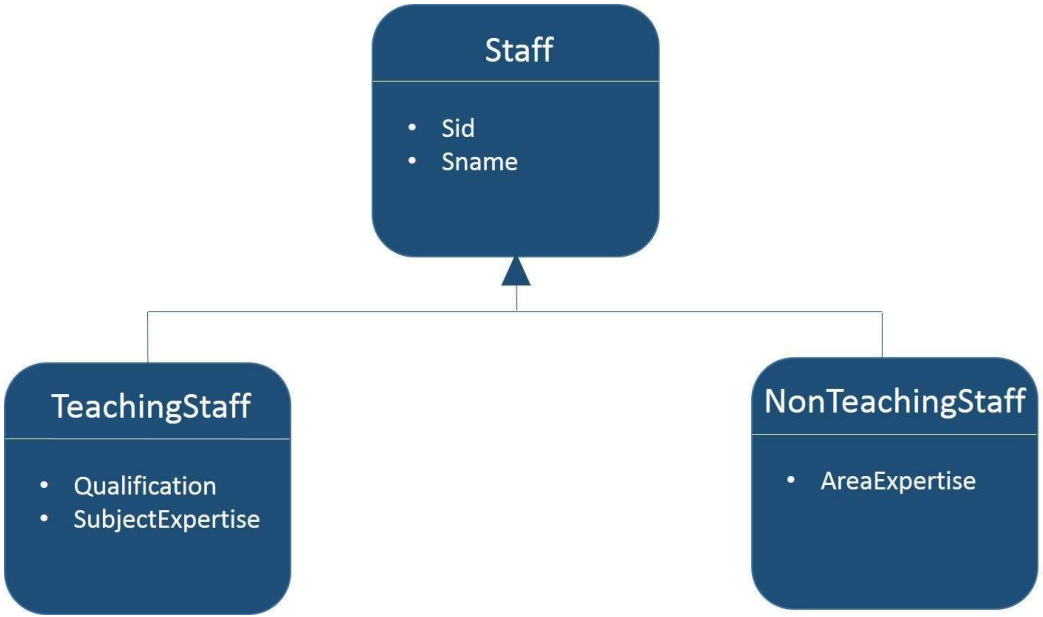
JPA - Relaciones avanzadas

UNJu - Programación Orientada a Objetos

Ing. José Zapana

Estrategias de herencia

- JPA admiten tres tipos de estrategias de herencia: SINGLE_TABLE, JOINED_TABLE y TABLE_PER_CONCRETE_CLASS.



Estrategia Single Table

- Mesa única estrategia toma todos los campos de las clases (clases tanto super y sub) y mapa abajo en una sola tabla conocida como estrategia **SINGLE_TABLE**. Aquí el valor discriminador desempeña un papel clave en diferenciar los valores de las tres entidades en una tabla.

```
@Entity
@Table
@Inheritance( strategy = InheritanceType.SINGLE_TABLE )
@DiscriminatorColumn( name="type" )
public class Staff implements Serializable
{
   @Id
   @GeneratedValue( strategy = GenerationType.AUTO )
   private int sid;
   private String sname;
}
```

Subclases

```java
@Entity
@DiscriminatorValue( value="TS" )
public class TeachingStaff extends Staff
{
   private String qualification;
   private String subjectexpertise;
}
```

```java
@Entity
@DiscriminatorValue( value = "NS" )
public class NonTeachingStaff extends Staff
{
   private String areaexpertise;
}
```

Probando el modelo

```
//Teaching staff entity
 TeachingStaff ts1=new TeachingStaff(1,"Gopal","MSc MEd","Maths");
 TeachingStaff ts2=new TeachingStaff(2, "Manisha", "BSc BEd", "English");

 //Non-Teaching Staff entity
 NonTeachingStaff nts1=new NonTeachingStaff(3, "Satish", "Accounts");
 NonTeachingStaff nts2=new NonTeachingStaff(4, "Krishna", "Office Admin");

 //storing all entities
 techingStaffRepository.save(ts1);
 techingStaffRepository.save(ts2);
 nonTeachingStaffRepository.save(nts1);
 nonTeachingStaffRepository.save(nts2);
```

Resultado en la base de datos

- Resultado luego de realizar la persistencia

Sid	Type	Sname	Areaexpertise	Qualification	Subjectexpertise
1	TS	Gopal		MSC MED	Maths
2	TS	Manisha		BSC BED	English
3	NS	Satish	Accounts		
4	NS	Krishna	Office Admin		

Estrategia Joined

```java
@Entity
@Table
@Inheritance( strategy = InheritanceType.JOINED )

public class Staff implements Serializable
{
   @Id
   @GeneratedValue( strategy = GenerationType.AUTO )

   private int sid;
   private String sname;
}
```

```java
@Entity
@PrimaryKeyJoinColumn(referencedColumnName="sid")

public class TeachingStaff extends Staff
{
   private String qualification;
   private String subjectexpertise;
}
```

```java
@Entity
@PrimaryKeyJoinColumn(referencedColumnName="sid")

public class NonTeachingStaff extends Staff
{
   private String areaexpertise;
}
```

Tablas generadas

Personal

Sid	Dtype	Sname
1	TeachingStaff	Gopal
2	TeachingStaff	Manisha
3	NonTeachingStaff	Satish
4	NonTeachingStaff	Krishna

TeachingStaff

Sid	Qualification	Subjectexpertise
1	MSC MED	Maths
2	BSC BED	English

NonTeachingStaff

Sid	Areaexpertise
3	Accounts
4	Office Admin

Estrategia Table per Class

```
@Entity
@Table
@Inheritance( strategy = InheritanceType.TABLE_PER_CLASS )
public class Staff implements Serializable
{
   @Id
   @GeneratedValue( strategy = GenerationType.AUTO )
   private int sid;
   private String sname;
}
```

```
@Entity
public class TeachingStaff extends Staff
{
   private String qualification;
   private String subjectexpertise;
}
```

```
@Entity
public class NonTeachingStaff extends Staff
{
   private String areaexpertise;
}
```

Resultado en la base de datos

- Resultado luego de realizar la persistencia

TeachingStaff

Sid	Qualification	Sname	Subjectexpertise
1	MSC MED	Gopal	Maths
2	BSC BED	Manisha	English

NonTeachingStaff

Sid	Areaexpertise	Sname
3	Accounts	Satish
4	Office Admin	Krishna

Relación de composición - Factura - Items

- Para manejar la relación entre una Factura y sus Ítems en JPA de manera óptima, lo más común es utilizar una relación uno a muchos (@OneToMany) entre las entidades.
- Una factura puede tener múltiples ítems, y cada ítem pertenece a una única factura. Esta es una estrategia común que se puede optimizar tanto en términos de rendimiento como de mantenibilidad.

Entidad Factura

```java
@Entity
@Table(name = "facturas")
public class Factura {

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

    private String cliente;

    @OneToMany(mappedBy = "factura", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Item> items = new ArrayList<>();

    public void addItem(Item item) {
        items.add(item);
        item.setFactura(this);
    }

    public void removeItem(Item item) {
        items.remove(item);
        item.setFactura(null);
    }
}
```

Entidad Item

```java
@Entity
@Table(name = "items")
public class Item {

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

    private String producto;
    private int cantidad;
    private double precioUnitario;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "factura_id")
    private Factura factura;
}
```

Relación y Manejo

- Factura contiene una lista de Items, representada por una relación uno a muchos (@OneToMany).
  - **cascade = CascadeType.ALL:** asegura que cuando se guarde, actualice o elimine una Factura, las operaciones relacionadas se propaguen a los Items.
  - **orphanRemoval = true:** Si un ítem se elimina de la lista de ítems de la factura, también será eliminado de la base de datos.
  - **El método addItem(Item item)** asegura que la relación bidireccional se mantenga consistente.
- Ítem tiene una relación muchos a uno (@ManyToOne) con Factura. Aquí se especifica la clave foránea con la anotación @JoinColumn.

Clases de Servicio

- Se utilizan para encapsular la lógica de negocio de una aplicación. Estas clases actúan como intermediarias entre la capa de controladores (que maneja las solicitudes del usuario) y la capa de persistencia (que interactúa con la base de datos).
- El propósito principal de una clase de servicio es aislar y organizar la lógica de negocio, manteniendo el código más limpio, modular y reutilizable.

Características principales de una clase de servicio

- **Encapsulan la lógica de negocio:** Contienen el código que aplica las reglas y procesos centrales de la aplicación. Esto permite que los controladores se enfoquen en manejar solicitudes y respuestas, mientras que la clase de servicio gestiona el "qué" y "cómo" hacer las operaciones.
- **Facilitan el mantenimiento:** Al separar la lógica de negocio de otras responsabilidades, es más fácil actualizar, modificar o depurar el código sin afectar otras partes del sistema.
- **Interfazan con la capa de persistencia:** A menudo, las clases de servicio utilizan repositorios o DAO (Data Access Objects) para interactuar con la base de datos a través de JPA, Hibernate, o cualquier otro ORM.
- **Reutilizables:** Al contener funciones que pueden ser utilizadas en varios lugares de la aplicación, permiten evitar la duplicación de código.

Ejemplo

```java
@Service
public class FacturaService {

    @Autowired
    private FacturaRepository facturaRepository;

    @Transactional
    public Factura crearFactura(Factura factura) {
        return facturaRepository.save(factura);
    }

    public Factura obtenerFactura(Long id) {
        return facturaRepository.findById(id).orElseThrow(() ->
                new RuntimeException("Factura no encontrada"));
    }

    public List<Factura> obtenerTodasLasFacturas() {
        return facturaRepository.findAll();
    }
}
```