Universidad de El Salvador.

Facultad Multidisciplinaria de Occidente.

Departamento de Ingeniería y Arquitectura.

Cátedra: *Programación III.*

Semana 3

GUIA #3 – Plain Old Java Object, Entities & Java Persistence Query Languaje.

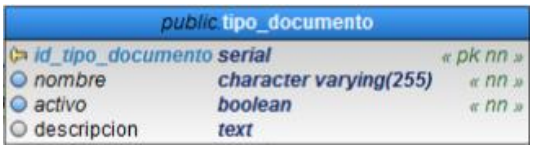
Objetivo: Que el estudiante aprenda de forma general, como se realiza el ORM (*Old Relational Mapping*) de una tabla, en una base de datos, junto a sus relaciones, para así obtener las Entities clases de una base de datos, mediante JPA, e introducir al estudiante, al uso de JPQL *(Java Persistence Query Languaje)* para realizar consultas en una base de datos.

Desarrollo.

Un **POJO** (acrónimo de Plain Old Java Object) es utilizado por programadores Java para enfatizar el uso de clases simples y que no dependen de un framework en especial, un POJO, es una clase que no implementa o extiende de otra.

Una **Entity o Entidad** es un objeto que por lo general, es la representación de una tabla de la base de datos y esta representación contiene todas las propiedades o campos que la tabla posea, junto a las relaciones existentes con otras entidades en la base de datos, en JAVA, cada tabla se representa por medio de una **Entity Class**, la cual se crea a partir de una POJO, agregando anotaciones especiales.

En este caso, seguiremos trabajando con la tabla de documento, y tipo documento, y se hará el Entity Class para ambas tablas.



Para poder hacer el Entity Class debemos primero crear la clase de Java que represente a la tabla, como podemos ver, tipo_documento y documento poseen campos de tipo “serial”, por lo cual al hacer la clase de java, podemos declarar la propiedad como tipo Integer, en el caso de los campo de tipo Character variying podemos usar String, con los campos de tipo text podemos usar como propiedades el tipo String, quedando nuestra clase de tipo_documento de la siguiente manera.

```java
package sv.edu.uesocc.ingenieria.prn335_2016.inventario.datos.definiciones;

/**
 *
 */
public class TipoDocumento{

    private Integer idTipoDocumento;
    private String nombre;
    private boolean activo;
    private String descripcion;


    // Getters & Setters

}
```

Ahora tenemos la clase de Java para tipo_documento, ahora debemos convertir la clase, en una Entity Class, y lo haremos agregando las siguientes anotaciones, las cuales se encuentran dentro de **javax.persistence.*,** y será necesario importarlas en nuestras clases, donde las anotaciones sean usadas.

- Agregar antes de la declaración de la clase la anotación ***@Entity***, con lo nuestro proveedor de persistencia sabrá que la clase es un Entity.
- Agregar después de ***@Entity***, la anotación ***@Table***, para definir las siguientes propiedades
    - *Name*, el nombre de la tabla en la base de datos.
    - *Catalog*, catalogo o base de datos donde está la tabla.
    - *Schema*, el esquema de donde está la tabla.
- Identificar la llave primaria, en este caso es idTipoDocumento y decorarlo con la anotación ***@Id***, ya que no es una llave compuesta.
- Ahora debemos decorar todos los campos con la anotación ***@Column***, para denotar que todas las propiedades con columnas de la tabla junto a las siguientes propiedades.
    - *Name*, el nombre del campo
    - *Nullable*, si el campo puede ser nulo o no.
    - *Lenght*, longitud máxima de la columna o propiedad
    - *Unique* si el campo es único.

- Agregar un constructor sin argumentos a la clase.
- Implementar *Serializable* en la clase, esto es para habilitar a nuestro Entity que pueda ser manejado de forma serializada, y pueda enviarse a través de la red por ejemplo.

Quedando nuestro Entity Class para tipo_documento de la siguiente manera.

```
12    // Package & Imports
13
14    @Entity
15    @Table(name = "tipo_documento",catalog = "inventario",schema = "public")
16    public class TipoDocumento{
17
18        public TipoDocumento(){    } // Constructor vacio
19
20        @Id
21        @Column(name = "id_tipo_documento",nullable = false)
22        private Integer idTipoDocumento;
23
24        @Column(name = "nombre", nullable = false, length = 255)
25        private String nombre;
26
27        @Column(name = "activo", nullable = false)
28        private boolean activo;
29
30        @Column(name = "descripcion")
31        private String descripcion;
32
33        // Getters & Setters
34    }
```

En el caso de Documento tenemos un campo con tipo *Date*, por lo cual debemos agregar a la propiedad en la clase, la anotación ***@Temporal***, y en este caso que es una fecha, quedaría de la siguiente manera la decoración.

@Temporal(javax.persistence.TemporalType.DATE).

De tal manera, que nuestro Entity para Documento estaría de la siguiente manera.

```java
// Package & Imports
@Entity
@Table(name="documento",schema="public")
public class Documento implements Serializable {

    public Documento(){    } // Constructor vacio

    @Id
    @Column(name="id_documento", nullable= false)
    private Integer idDocumento;

    @Column(name="codigo", nullable = false, length = 225)
    private String codigo;

    @Column(name="descripcion", nullable = true, length = 225)
    private String descripcion;

    @Column(name = "fecha_realizacion", nullable = false)
    @Temporal(javax.persistence.TemporalType.DATE)
    private Date fechaRealizacion;

    @Column(name = "ubicacion_fisica", nullable=true)
    private String ubicacionFisica;

    @Column(name = "url", nullable=true)
    private String url;

    @Column(name = "oficializado", nullable=false)
    private boolean oficializado;

    // Getters & Setters
}
```

En este punto no está aún terminado nuestro Entity Class pues hacen falta las relaciones, *Documento* tiene una relación hacia tipo_documento de ***muchos a uno*** entonces lo especificaremos agregando una propiedad del tipo *TipoDocumento*, con la anotación ***@ManyToOne.***

```java
// package & Imports
    // Declaracion de la clase
        // Constructor Vacio
        // Declaracion de propiedades

        @ManyToOne
        private TipoDocumento idTipoDocumento;

        // Getters & Setters

}
```

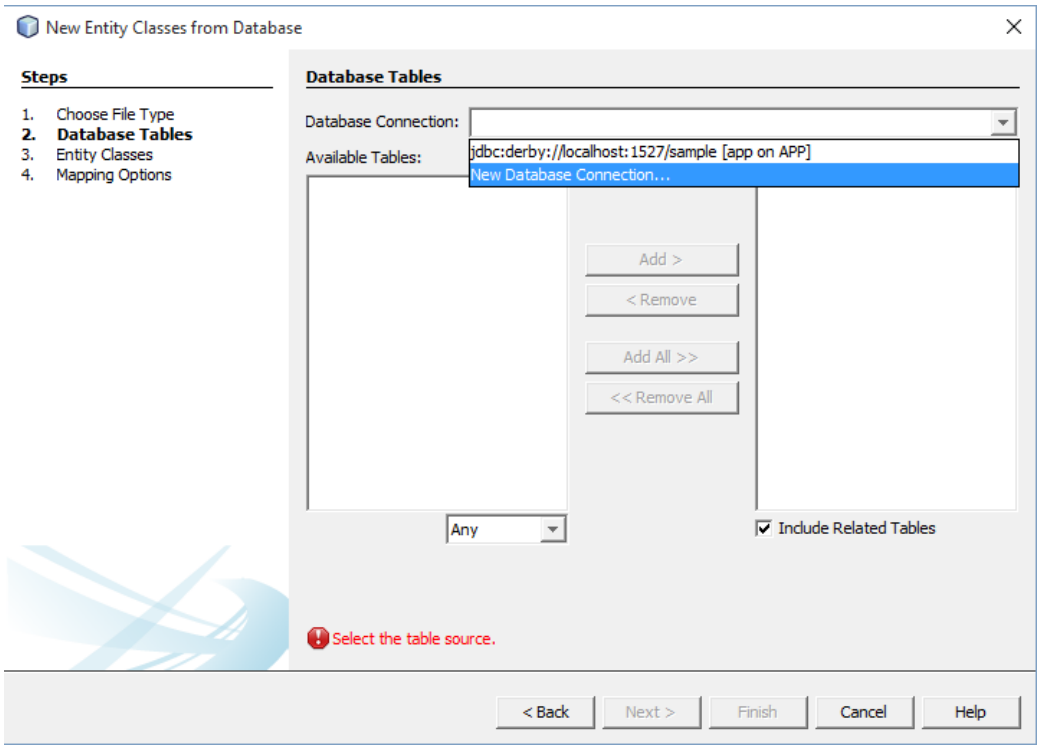
Y en la Entity, *TipoDocumento* agregaremos también una nueva propiedad que especifique la relación de ***uno a muchos*** hacia *Documento*, esta será una lista de *Documento* con la anotación ***@OneToMany***, con la propiedad ***mappedBy***, en la cual especificamos el nombre de la propiedad que es dueña de la relación, en nuestro caso, *idTipoDocumento*, quedando de la siguiente manera nuestros Entity Class.

```
33    // package & Imports
34        // Declaracion de la clase TipoDocumento
35            // Constructor Vacio
36            // Declaracion de propiedades
37
38            @OneToMany(mappedBy = "idTipoDocumento")
39            private List<Documento> documentoList;
40
41            // Getters & Setters
42    }
```

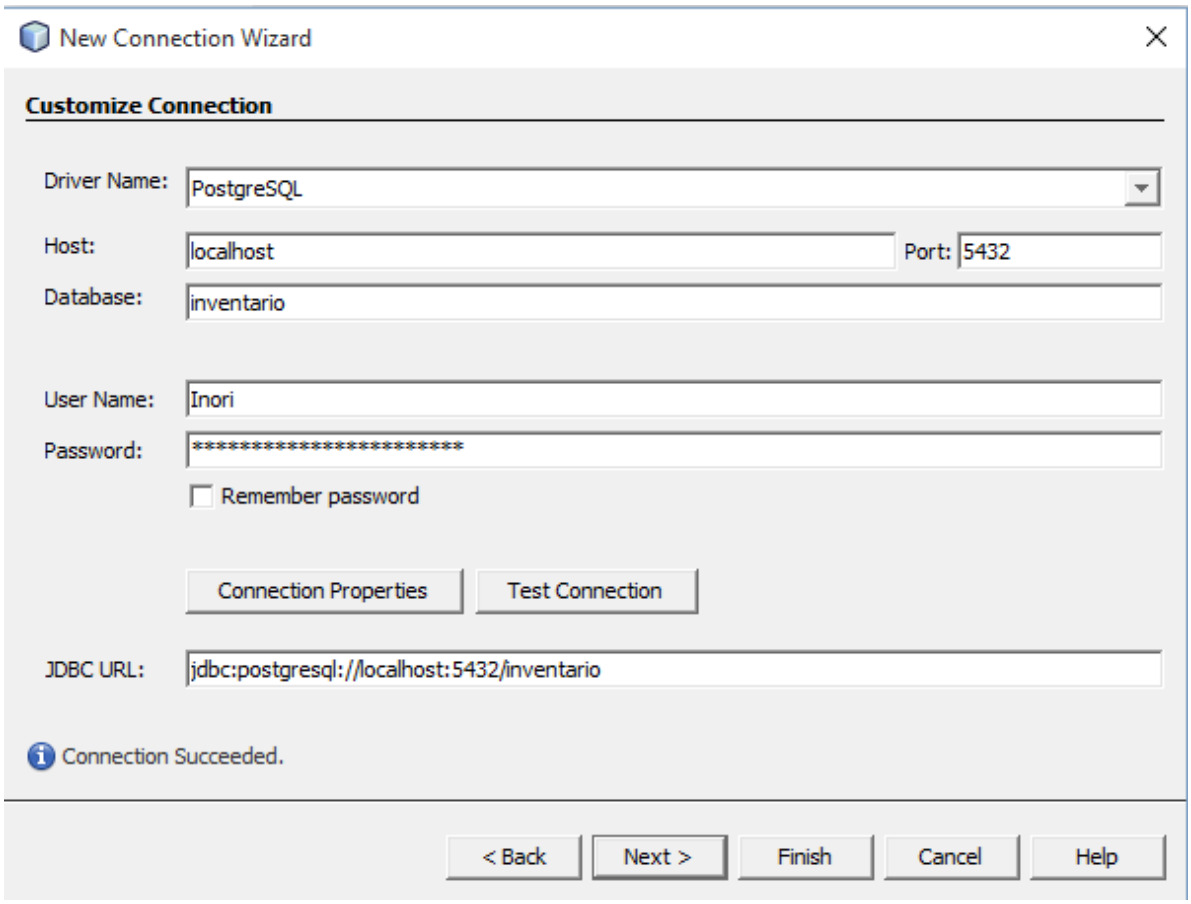
Mapeo de las Entities usando el asistente de Netbeans.

Mediante el asistente de Netbeans es posible realizar el mapeo de las tablas de la base de datos, siguiendo los siguientes pasos.

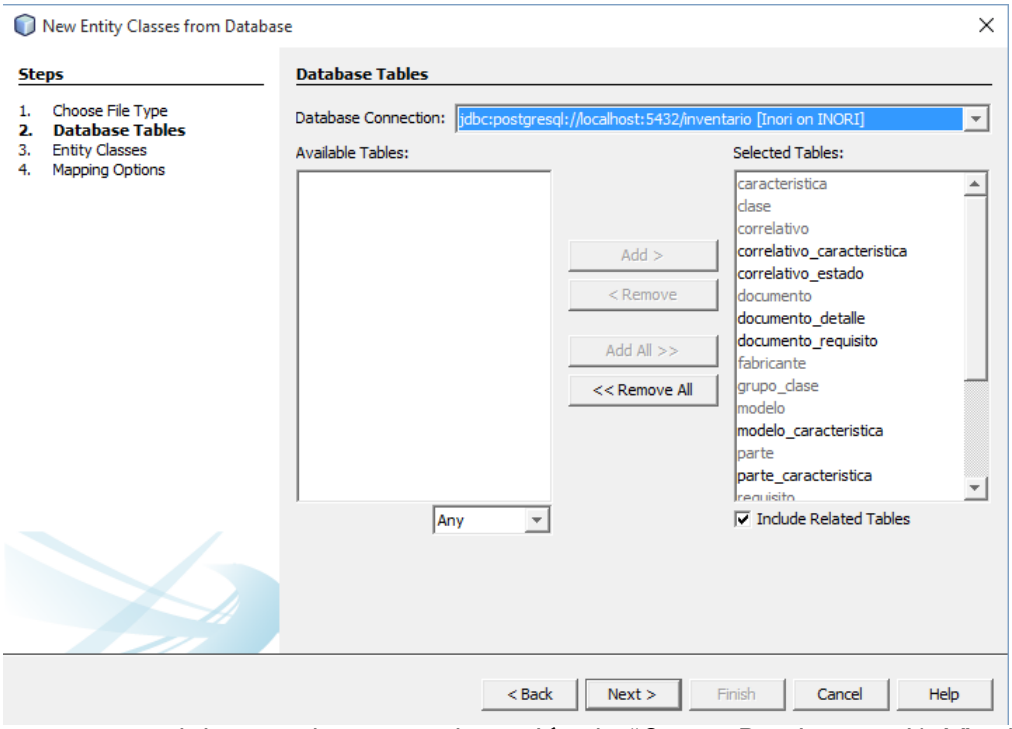
1. Creamos un nuevo proyecto en Netbeans, seleccionamos la categoría ***Java*** y seleccionamos ***Java Class Library*** y le damos a **siguiente**, le asignamos InventarioLib de nombre, y le damos a finalizar.
2. Dentro del proyecto *InventarioLib*, creamos un nuevo ***Java Package***, con el nombre ***sv.edu.uesocc.ingenieria.prn335_2016.inventario.datos.definiciones.***
3. Hacemos click *derecho* sobre el paquete, y luego click en *New* y seleccionamos ***Entities Classes From Database.*** Ahora en esta pantalla estaría de la siguiente forma.



4. Seleccionamos ***New Database Connection***.
    - Seleccionamos el Driver de ***PostgreSQL*** y le damos a siguiente.
    - Una vez hecho esto, aparecerá una nueva ventana, con los campos Host, Port, Database, User Name, Password y JDBC URL.
    - En **Host**, pondremos el host de nuestro PC, **localhost**, el puerto por defecto de **PostgreSQL** es **5432**, **Database** pondremos *inventario*, ya que es la base de datos que estamos obteniendo los entities, y agregamos el nombre de usuario que creamos en la guía anterior junto a su contraseña.
    - Una vez hecho esto, damos al botón de **Test Connection**, y si todo va bien aparecerá en letras azul un mensaje de "***Connection Succeeded***".
    - Quedando de la siguiente manera, habiendo comprobado que la conexión tuvo éxito solo resta, darle a finalizar.



5. Una vez creada la conexión JDBC si no estuviese, en la ventana de Entity Clases From Database aparecerán del lado izquierdo todas las tablas en la base de datos, y daremos click a Add All, para seleccionar todas, para decirle al asistente que deseamos todos los entities, y le daremos a Siguiente.



6. En este paso, debemos desmarcar la opción de "*Create* Persistence *Unit*" y hacemos click en siguiente o next.

7. En este momento el asistente nos mostrara varias opciones para la creación de los entities, entre ellas esta:

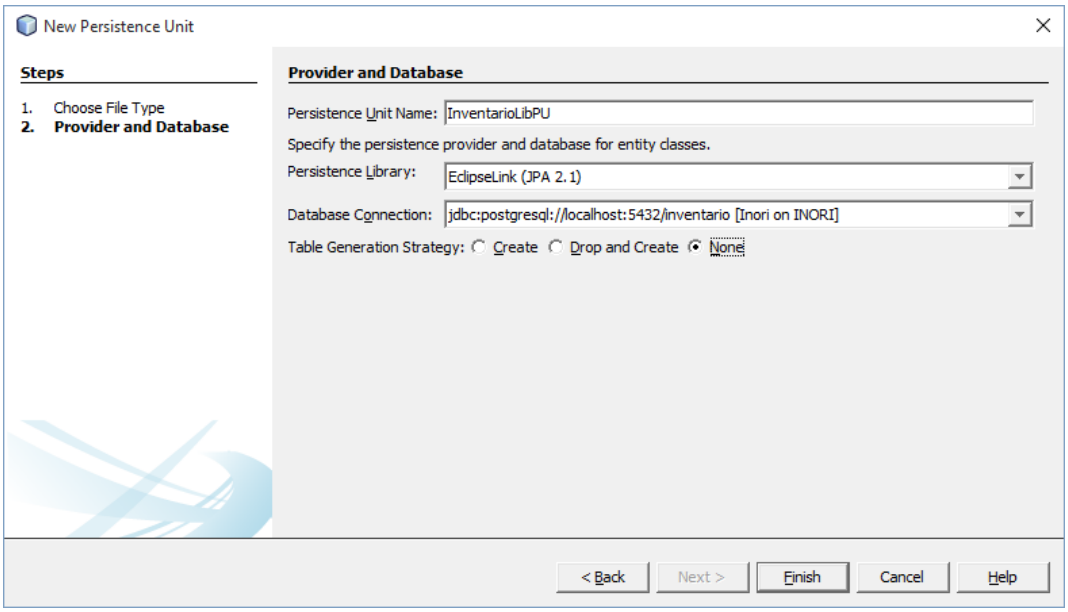
   - ***Association Fetch***: Es el tipo de asociación que se realiza a la hora de hacer una consulta, en la cual *Eager*, trae el valor de una tupla, junto a todas sus relaciones, supongamos que realizamos la consulta para obtener un documento, de modo Eager, traerá además de documento, todos los datos de tipo documento, por el contrario *Lazy*, solamente traerá los datos de Documento y únicamente los de tipo documento cuando realice una nueva consulta en base a la anterior o que se especifique que será por medio de *Eager*, en este caso dejaremos en Default, para usar el que nuestro proveedor de persistencia use por defecto.
   - ***Collection Type***: Seleccionaremos List, para que cuando la consulta devuelva muchas tuplas, estén almacenadas como una lista.
   - Marcaremos la opción de *Fully Qualified Database Tables Names*, para que el asistente ponga en cada entity el nombre completo de cada tabla.

8. Al dar finalizar el asistente creara todas las Entities clases.

Creando el Persistence Unit & introducción a JPQL.

Una unidad de persistencia (Persistence Unit) desde ahora llamaremos PU, representa un conjunto de entidades que pueden ser mapeadas a una base de datos, así como la información necesaria para que la aplicación JPA (Java Persistence API) pueda acceder a dicha base de datos. Se define mediante un archivo llamado *persistence.xml*

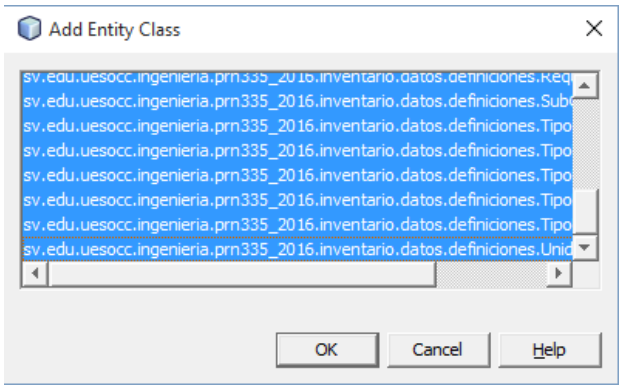
1. En el proyecto donde creamos las entities Classes, haremos click derecho, y buscaremos *Persistence Unit* y aparece un asistente que tendrá:
   - ***Persistence Unit Name***: El nombre que queremos darle a nuestra PU.
   - ***Persistence Library***: El proveedor de persistencia que deseamos usar, en este caso usaremos EclipseLink 2.1 como proveedor.
   - ***Database Connection***: La conecion a la base de datos, en al cual seleccionarmos la conexión que creamos al mapear las entidades.
   - **Importante,** en Table Strategy Generation seleccionar ***None***, pues podría dañar todos los datos en la base de datos.



2. Dar Click a finalizar.

3. Se abrirá el archivo de *persistence.xml,* en el caso de no abrirse, podemos ir hasta el source package META-INF y dentro se encontrara el persistence.xml.

4. Una vez abierto daremos al botón Add Class y aparecerá una ventana para seleccionar las Entity Classes que deseamos agregar al PU, seleccionaremos todas y daremos al botón de OK.



5. Cerramos el archivo persistence.xml, y guardamos los cambios.

Hecho esto compilamos nuestro proyecto. Ahora llego la hora de usar JPQL, y Netbeans nos brinda una herramienta para esto, hacemos click derecho a el *persistence.xml*, y seleccionamos **Run JPQL Query.**

Introducción a JPQL (Java Persistence Query Languaje)

Es importante recalcar que JPQL no es SQL, y tienen muchas diferencias, ya que JPQL está más orientado a objetos, requiere un modo diferente de tratar. Si JPQL no es SQL, entonces que es, JPQL es el lenguaje para hacer Queries sobre Entities, y nos provee una forma de expresar las Queries en forma de entidades y sus relaciones, que operan en el estado de persistencia de las entidades definidas en el modelo de objetos, y no sobre el modelo físico de la base de datos.

Terminología.

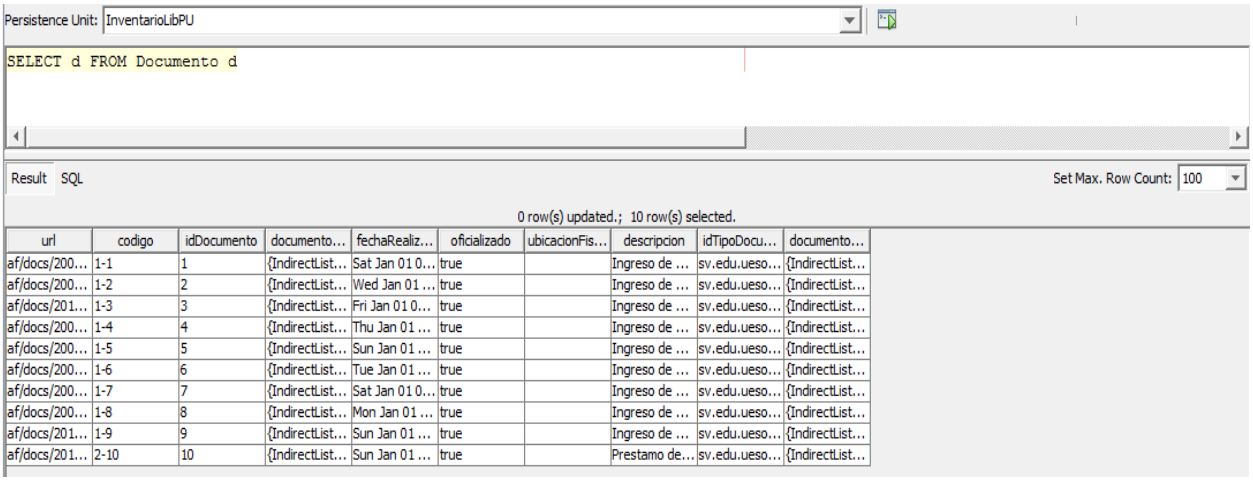
Las Queries en JPQL pueden caer en una de las siguientes categorías: ***Select Aggregate, Update*** y ***Delete.*** *Select* devuelve el estado persistido de una o más entidades, filtrándose según se requiera, Aggregate es una variante de Select que agrupa resultados y muestra un resumen, y Update junto a Delete son Queries usadas para modificar o eliminar registros de uno o más Entities.

Las Select Queries, son las más significantes a la hora de obtener datos desde nuestra base de datos, y la forma general de una Query de este tipo es la siguiente.

SELECT <*Expresión de selección*> FROM <*Clausula From*> [WHERE <*Expresión Condicional*>] [ORDER BY <*Clausula de ordenamiento*>]

La expresión de selección define el formato de los resultados, mientras que la cláusula *From*, define de qué entidades se *tomaran los datos*, ambas son obligatorias en cualquier Query de tipo Select, mientras que *Where*, *Order By* son opcionales.

Ejemplo de Query Select.



url	codigo	idDocumento	documento...	fechaRealiz...	oficializado	ubicacionFis...	descripcion	idTipoDocu...	documento...
af/docs/200...	1-1	1	{IndirectList...	Sat Jan 01 0...	true		Ingreso de ...	sv.edu.ueso...	{IndirectList...
af/docs/200...	1-2	2	{IndirectList...	Wed Jan 01 ...	true		Ingreso de ...	sv.edu.ueso...	{IndirectList...
af/docs/201...	1-3	3	{IndirectList...	Fri Jan 01 0...	true		Ingreso de ...	sv.edu.ueso...	{IndirectList...
af/docs/200...	1-4	4	{IndirectList...	Thu Jan 01 ...	true		Ingreso de ...	sv.edu.ueso...	{IndirectList...
af/docs/200...	1-5	5	{IndirectList...	Sun Jan 01 ...	true		Ingreso de ...	sv.edu.ueso...	{IndirectList...
af/docs/200...	1-6	6	{IndirectList...	Tue Jan 01 ...	true		Ingreso de ...	sv.edu.ueso...	{IndirectList...
af/docs/200...	1-7	7	{IndirectList...	Sat Jan 01 0...	true		Ingreso de ...	sv.edu.ueso...	{IndirectList...
af/docs/200...	1-8	8	{IndirectList...	Mon Jan 01 ...	true		Ingreso de ...	sv.edu.ueso...	{IndirectList...
af/docs/201...	1-9	9	{IndirectList...	Sun Jan 01 ...	true		Ingreso de ...	sv.edu.ueso...	{IndirectList...
af/docs/201...	2-10	10	{IndirectList...	Sun Jan 01 ...	true		Prestamo de...	sv.edu.ueso...	{IndirectList...

Podemos observar que la estructura es similar a SQL, podemos notar la primera diferencia, y es que en la cláusula del FROM, *no hay una tabla, sino la Entity que representa a la tabla*, y además de esto seguida de ella, aparece “d” que es el alias que *asignamos* a la Entity de Documento como “d”, este alias es conocido como la *variable de identificación* de la Entity en toda la Query, por lo cual es uso de este es obligatorio.

Además notamos que no definimos que campos queremos obtener en nuestros resultado, o un wildcard para seleccionar todos los campos, en consecuencia hacemos uso del alias de Documento, denotando que el resultado de la Query será de Documento, y no un resultado tabulado de filas.

El resultado de la Query será una *Lista, o colección de cero o más objetos de tipo Documento.* Es posible navegar entre la Entity usando el operador (.), con el cual si por ejemplo hacemos d.codigo, obtendríamos el código de un documento, pero además de esto haciendo d.idTipoDocumento.nombre, obtendríamos el nombre del Tipo de documento al cual el documento pertenece.

Ejemplo.

Persistence Unit: InventarioLibPU

```
SELECT d.codigo , d.descripcion, d.idTipoDocumento.nombre FROM Documento d
```

Result SQL Set Max. Row Count: 100

0 row(s) updated.; 10 row(s) selected.

Column 1	Column 2	Column 3
1-9	Ingreso de 2 proyectores al Centro de Cómputo de Ingeniería FMOcc	M1
1-8	Ingreso de 30 computadoras Dell al Centro de Cómputo de Ingeniería FMOcc	M1
1-7	Ingreso de 3 vehiculos tipo coaster a la FMOcc	M1
1-6	Ingreso de 10 computadoras Dell al departamento de Ingeniería FMOcc	M1
1-5	Ingreso de 2 impresoras y 2 fotocopiadoras al departamento de Ingeniería FMOcc	M1
1-4	Ingreso de 3 archiveros al departamento de Ingeniería FMOcc	M1
1-3	Ingreso de 5 escritorios y 5 sillas al departamento de Física FMOcc	M1
1-2	Ingreso de 5 escritorios y 5 sillas al departamento de Ingeniería FMOcc	M1
1-1	Ingreso de 10 escritorios y 10 sillas al departamento de Ingeniería FMOcc	M1
2-10	Prestamo de 2 escritorios de Ingeniería a Física FMOcc	M2

La Clausula ***FROM***, es usada para declarar una, o más *alias*, los cuales se pueden hacer explícitamente anteponiendo al alias la palabra AS después de la *Entity*, el uso de *JOIN* es el equivalente al usado en *SQL*. En el Ejemplo podemos ver el uso de *AS*, y además de esto, únicamente estamos mostrando 2 campos en nuestra respuesta, el código de un *Documento*, junto al nombre del tipo al que el documento pertenece.

Persistence Unit: InventarioLibPU

```
SELECT d.codigo, td.nombre FROM Documento AS d JOIN d.idTipoDocumento AS td
```

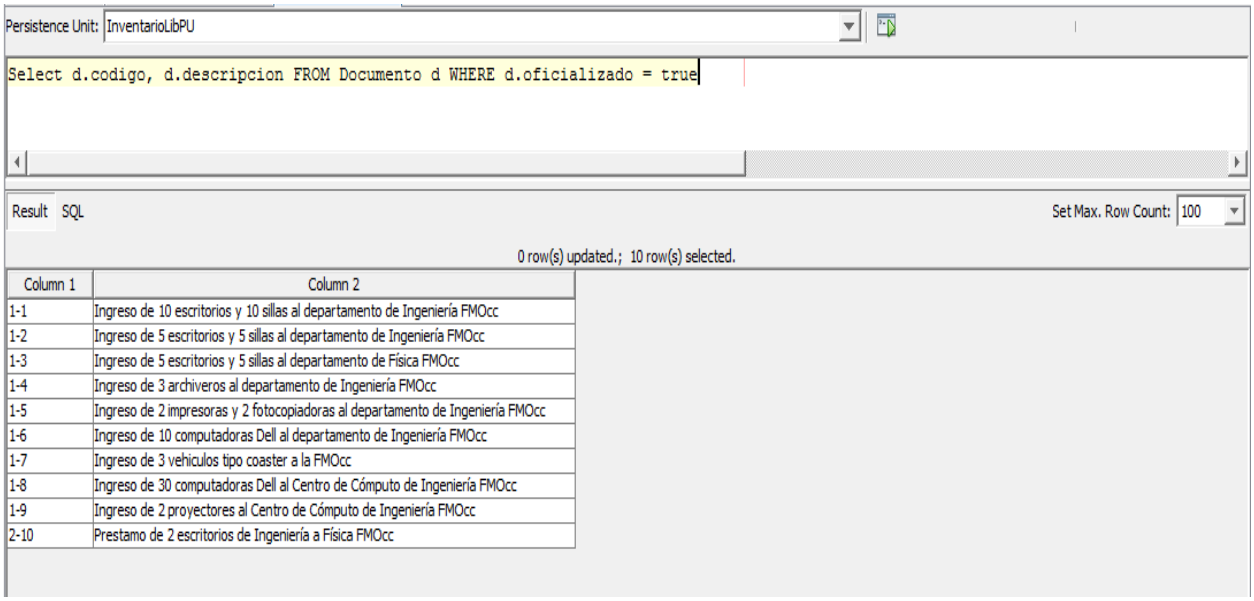
Result SQL Set Max. Row Count: 100

0 row(s) updated.; 10 row(s) selected.

Column 1	Column 2
1-9	M1
1-8	M1
1-7	M1
1-6	M1
1-5	M1
1-4	M1
1-3	M1
1-2	M1
1-1	M1
2-10	M2

Clausula ***WHERE***, es usada para establecer filtros específicos y reducir el resultado, en los ejemplos anteriores, nos devolvía todos los registros de la tabla Documento, pero usando una clausula *WHERE* podemos filtrar únicamente ciertos registros.

Ejemplo.



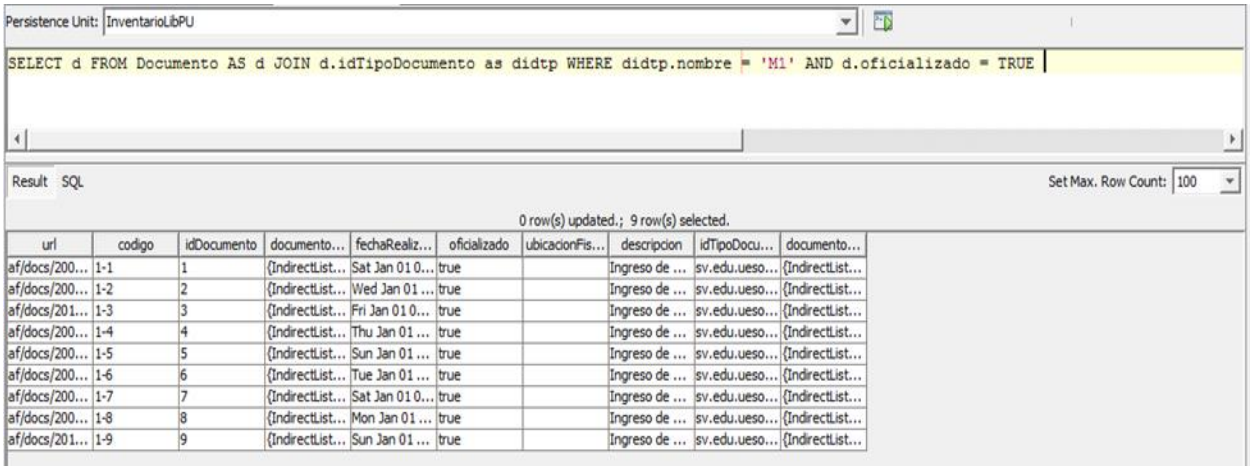
Column 1	Column 2
1-1	Ingreso de 10 escritorios y 10 sillas al departamento de Ingeniería FMOcc
1-2	Ingreso de 5 escritorios y 5 sillas al departamento de Ingeniería FMOcc
1-3	Ingreso de 5 escritorios y 5 sillas al departamento de Física FMOcc
1-4	Ingreso de 3 archiveros al departamento de Ingeniería FMOcc
1-5	Ingreso de 2 impresoras y 2 fotocopiadoras al departamento de Ingeniería FMOcc
1-6	Ingreso de 10 computadoras Dell al departamento de Ingeniería FMOcc
1-7	Ingreso de 3 vehiculos tipo coaster a la FMOcc
1-8	Ingreso de 30 computadoras Dell al Centro de Cómputo de Ingeniería FMOcc
1-9	Ingreso de 2 proyectores al Centro de Cómputo de Ingeniería FMOcc
2-10	Prestamo de 2 escritorios de Ingeniería a Física FMOcc

Esta consulta filtraría todos los documentos que estén oficializados y nos mostraría el código de estos, junto a su descripción.

Expresiones básicas.

1. Operador de navegación **(.)**
2. Multiplicacion (*) & Division (/)
3. Suma (+) & Resta (–)
4. Operadores de comparacion =, >, >=, <, <=, <>, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF]
5. Operadores Logicos (AND, OR, NOT)

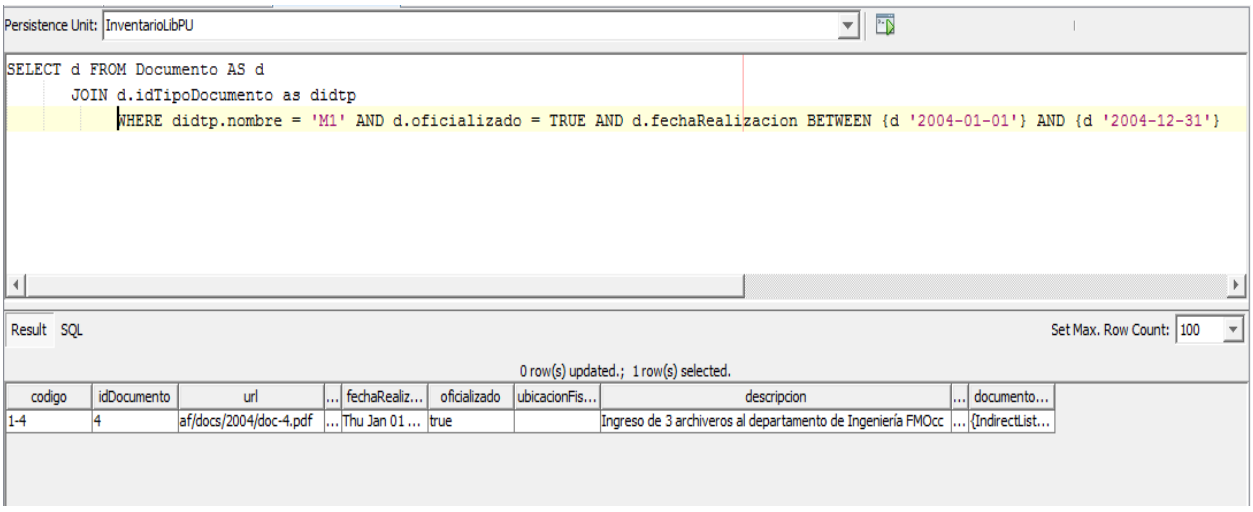
Ahora, conociendo los operadores básicos, realizaremos una consulta que muestre todos los Documentos, que fueron ingresados con M1 y que el documento este oficializado, nótese que no hay una restricción a la hora de mostrar los resultados, por lo cual mostraremos todo el Documento, quedando nuestra Query de la siguiente manera, junto a el resultado obtenido.



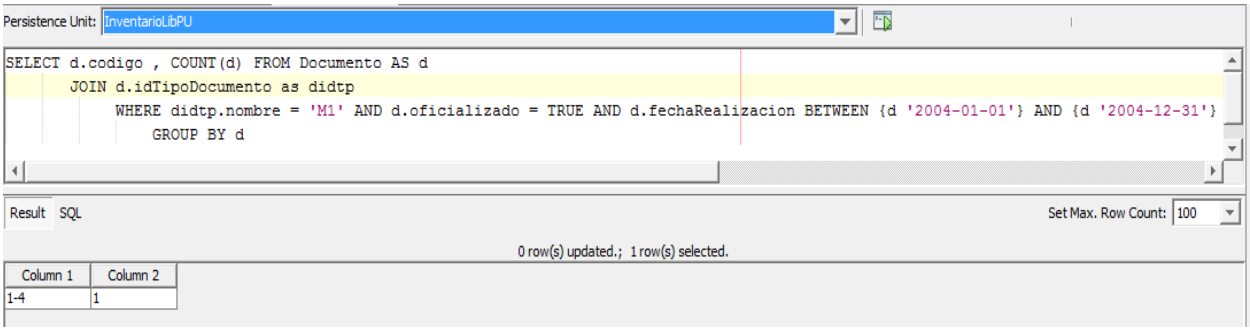
Como podemos ver, aunque son únicamente 9 datos, en una base de datos con miles de registros, o millones de registros, si quisiésemos buscar los que se realizaron en 2004, tardaríamos años tratando de encontrarlos, por lo cual podemos añadir una nueva condición a nuestra consulta anterior, la cual iría en la cláusula del WHERE. Para las condiciones de filtrado cuyo tipos de dato sea temporal (Fecha y Hora o TimeStamp), se usa la sintaxis de escape de JDBC, de la siguiente manera.

{d 'yyyy-mm-dd'}
{t 'hh-mm-ss'}
{ts 'yyyy-mm-dd hh-mm-ss.f'}

De forma que nuestra consulta quedaría esta vez de la siguiente manera, y en el resultado estarían únicamente los documentos entre 1/1/2004 y el 31/12/2004.



Ahora veremos algunas funciones de Totalización, entre las cuales está, **COUNT**. **COUNT**, nos sirve para realizar un conteo del número de ocurrencias de un campo en un grupo, por ejemplo si quisiésemos ver cuántas veces aparece el documento M1 en el año 2004, junto al código del documento, seria de la siguiente manera.

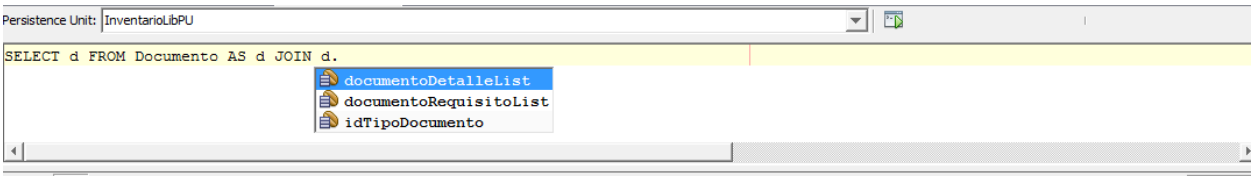


Como podemos observar, además hemos añadido al final de la cláusula de WHERE, GROUP BY, para indicar que los resultados serán agrupados por Documento. Junto a COUNT se puede usar DISTINC con el cual si hubiese registros repetidos, estos serían solamente tomados una vez cuando se haga el conteo. DISTINCT también puede ser usado sin COUNT para no mostrar registros duplicados.

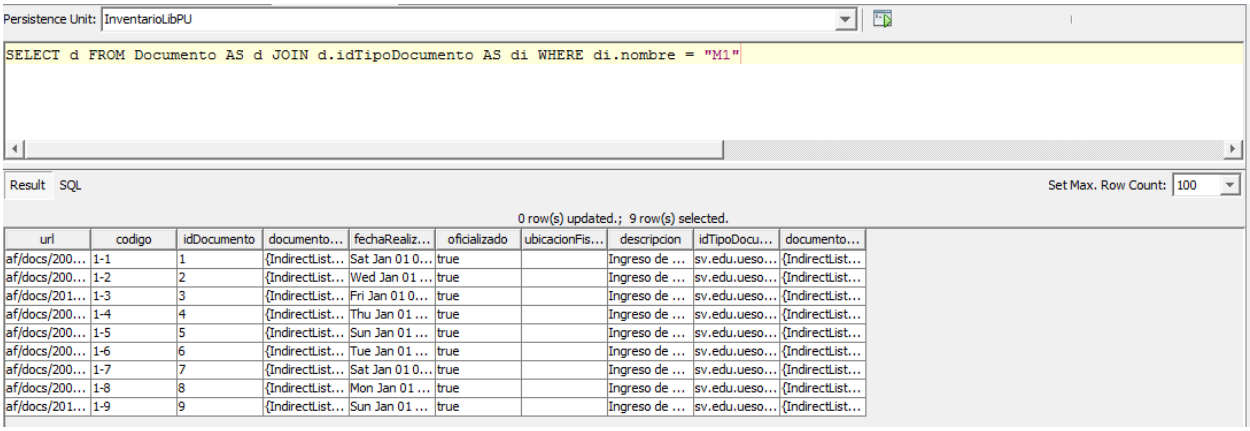
Supongamos que tenemos la siguiente consulta.

SELECT d FROM Documento AS d

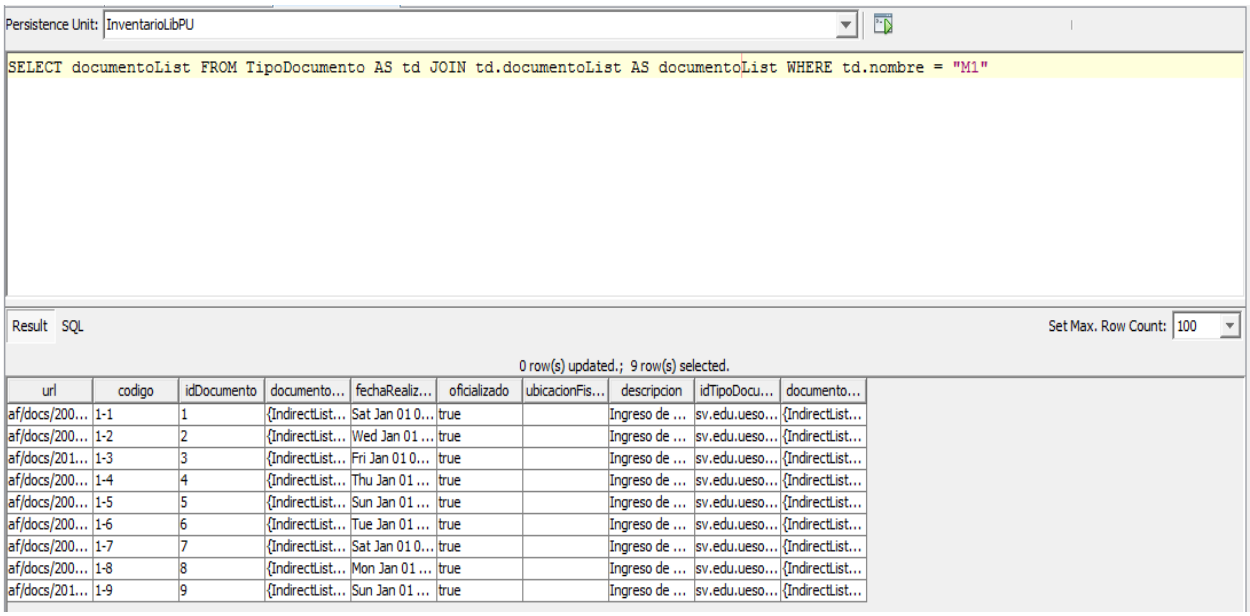
Y seguido hacemos un JOIN, notaremos que aparecen las siguientes opciones, las cuales son las relaciones que tiene Documento hacia las demás tablas, y podemos notar que las tablas que dependen de documento tienen al final agregado "***List***" y las tablas de las cual depende Documento simplemente el nombre de campo al cual están relacionadas.



Si quisiéremos hacer una consulta, que nos muestre todos los documentos con tipo de documento "M1" podríamos hacerlo de una de las siguientes formas, en las cuales se inicia por diferente lugar, el uso de JOIN. Si vamos de Documento hacia *TipoDocumento*, donde Documento tiene el campo que relaciona a *TipoDocumento* será.



Mientras que al hacerlo por medio de TipoDocumento, la cual no tiene en ella relacionado los Documentos a los cuales ella hace referencia, tendremos que usar el ***List***, y el origen de este, está en el momento en que hicimos el mapeo de las entidades, en la que TipoDocumento tenía una propiedad ***@OneToMany*** de tipo ***List<Documento> documentoList,*** la cual nos servirá en este momento para realizar la consulta, quedando de la siguiente manera.



Ejercicios Propuestos.

1- Elabore usando JPQL (Java Persistence Query Languaje) una consulta que sea capaz de mostrar los documentos, que tengan como tipo de requisito “Participacion”.

2- Elabore usando JPQL, una consulta que muestre los correlativos que han ingresado (Cuenten con un documento M1 oficializado) en los últimos 3 años, y se encuentre en Buen estado.