

JPA – Relaciones entre Entidades

Universidad Centroamericana “José Simeón Cañas”
Ciclo 01-2020

Así como las tablas en una base de datos están relacionadas, también las entidades deben estarlo

- Estas relaciones son soportadas por JPA para representar a las entidades a como se encuentran a nivel de base de datos
- Dichas relaciones se hacen en las clases dominios de las entidades correspondientes
- La configuración se hace a nivel de anotaciones para cada una de las posibles relaciones

Por ejemplo, para la relación a nivel de tablas



Asumimos que hay dos clases dominio, Cliente y Vehículo, luego, para configurar esta relación lo hacemos de la siguiente manera:

1. Como la relación es de 1 cliente a N vehículos, eso significa que un cliente puede tener muchos vehículos. Debido a esto crearemos una propiedad en la entidad Cliente (Cliente.java) que contenga a N cantidad de vehículos. Una Lista tipeada (List<T>) es una colección que nos ayuda a este fin.

```
@OneToMany(mappedBy = "cliente", fetch = FetchType.EAGER)
private List<Vehiculo> vehiculos;
```

Utilizamos la anotación @OneToMany para definir la cardinalidad (Cliente.java)

- La anotación @OneToMany define que para esta entidad (Cliente) hay una relación de Uno a Muchos para la propiedad siendo anotada (En este caso la lista de vehículos).

```
@OneToMany(mappedBy = "cliente", fetch = FetchType.EAGER)  
private List<Vehiculo> vehiculos;
```

- Esta anotación recibe dos propiedades
 - **mappedBy**: Que indica la propiedad en la entidad de la lista (Vehículo) que es dueña (o papá) de la relación (cliente)
 - **Fetch**: La forma en que se obtendrán los registros hijos de la relación.
 - **FetchType.EAGER**: Al obtener un determinado cliente, se obtendrán también TODOS los vehículos de ese cliente automáticamente. Esto quiere decir, que al obtener el objeto Cliente, si examinamos la colección de Vehículos, vendrán con todos los objetos vehículos de ese cliente. Hay que tener cuidado con esto, ya que si la cantidad de hijos es demasiado grande (millones), tendremos problemas de rendimiento.
 - **FetchType.LAZY**: Se obtendrán los vehículos de ese cliente únicamente al ejecutar el método getVehiculos(). Contrario al anterior, al obtener el objeto Cliente del DAO, la colección de vehículos vendrá vacía, y será obtenida hasta que ejecutemos el método **get** de dicha colección.

Ahora debemos definir la relación Muchos a Uno en la entidad Vehículos (Vehículo.java)

```
@ManyToOne(fetch = FetchType.EAGER)
@JoinColumn(name = "c_cliente")
private Cliente cliente;
```

- Como un vehículo solamente puede hacer referencia a **un cliente** en específico, no la debemos de representar mediante una colección de Clientes, sino solamente como un único objeto de tipo Cliente.
- Para eso utilizamos la anotación @ManyToOne, sin embargo, no definimos la propiedad **mappedBy**, ya que la entidad Vehículo no es el papá de dicha relación.
- Le decimos que la estrategia de obtención de los registros relacionados es **FetchType.EAGER**. Esto significa que cuando obtengamos un registro Vehículos, automáticamente Hibernate cargará la entidad Cliente correspondiente.
- Utilizamos la anotación @JoinColumn para especificarle **la columna** (no la propiedad) por la cual estas dos entidades están relacionadas, esto servirá para que hibernate pueda realizar la consulta correspondiente a la tabla vehículo por medio de dicha columna para obtener los vehículos de un cliente

Propiedad en Hibernate

- Para utilizar el tipo LAZY debemos de agregar la siguiente propiedad a nuestro método hibernateProperties en la clase JpaConfiguration.java


```
properties.setProperty("hibernate.enable_lazy_load_no_trans","true");
```

Esto habilita el uso de la sesión para obtener la colección al momento de llamar al método get correspondiente.

Demostración

Ingresaremos al aplicativo

- URL <http://localhost:8080/index16> o desde el menú lateral en Clase 16:



Spring MVC / Thymeleaf

Dashboard

Código Clases

Relaciones entre Entidades

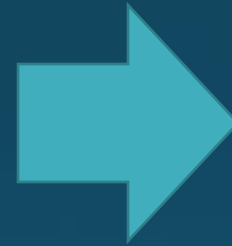
Ingrese el código del Cliente:

Buscar Cliente

Ingresamos un código de cliente a buscar

Relaciones entre Entidades

Ingrese el código del Cliente:



Información del Cliente

Nombres:

Apellidos:

Fecha de Nacimiento:

Estado: ☐ Activo

Vehículos propiedad del cliente

Marca	Modelo	No. Chassis	Fecha de Compra	Estado
Bentley	Brooklands	SAJWA6BC7F8553385	20/08/2014	MATRICULADO
Hyundai	Accent	WAUDFAFL6DN252753	20/08/2014	MATRICULADO
Pontiac	Sunfire	2G4G05GV0B9623670	20/08/2014	NO MATRICULADO

Ingresamos el código del cliente, y al dar clic al botón Buscar, se nos muestra ahora, tanto la información del cliente, como una tabla donde se muestran los vehículos asociados al cliente.

Veamos como se ha logrado esto.

Revisemos el HTML para mostrar la tabla

```
<div class="card mb-3">
  <div class="card-header">
    Vehículos propiedad del cliente
  </div>
  <div class="card-body">
    <div class="table-responsive">
      <table class="table table-bordered" id="tablaVehiculos" width="100%" cellpadding="0">
        <thead>
          <tr>
            <th>Marca</th>
            <th>Modelo</th>
            <th>No. Chassis</th>
            <th>Fecha de Compra</th>
            <th>Estado</th>
          </tr>
        </thead>
        <tbody>
          <tr th:each="v : ${cliente.vehiculos}">
            <td th:text="${v.smarca}"></td>
            <td th:text="${v.smodelo}"></td>
            <td th:text="${v.schassis}"></td>
            <td th:text="${v.fechaDelegate}"></td>
            <td th:text="${v.estadoDelegate}"></td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
```

Lo que hemos hecho para pintar los vehículos del cliente es utilizar el tag de thymeleaf **th:each**, el cual, como recordaremos, itera sobre una colección que es enviada desde el controlador, en este caso la colección de vehículos. Recordemos que desde el controlador hemos enviado el objeto cliente en el ModelAndView llamado como "cliente", por lo que le especificamos la colección con el método **getVehiculos()** de dicho objeto, el cual se define quitándole el prefijo **get** y la primera letra en minúscula (la V) por lo que quedaría **cliente.vehículos**.

Luego, a cada objeto Vehículo de esta colección lo referenciaremos con la letra **v**, y luego procedemos a pintar en cada **<td>** cada propiedad de dicho objeto, utilizando el tag de Thymeleaf **th:text**.

Como el **FetchType** de la propiedad `List<Vehiculo>` la hemos definido como **LAZY**, significa que hasta que ejecutamos el método **get** de dicha propiedad se realiza la consulta para obtener la colección de vehículos. En este caso, la ejecución del método **get** se hace en **\${cliente.vehículos}** del **th:each**.

Si depuramos el aplicativo y ponemos un breakpoint al momento de ejecutar el método DAO que obtiene al cliente veremos en la consola:

```
@RequestMapping("/buscarcliente16")
public ModelAndView buscar16(@RequestParam Integer codigo) {
    ModelAndView mav = new ModelAndView();
    Cliente c = clienteDao.findOne(codigo);
    mav.addObject("cliente", c);
    mav.setViewName("clases/clase16/cliente");
    return mav;
}
```



```
2020-05-13 17:47:23.090 INFO 9540 --- [nio-8080-ex
2020-05-13 17:47:23.090 INFO 9540 --- [nio-8080-ex
2020-05-13 17:47:23.097 INFO 9540 --- [nio-8080-ex
Hibernate: select cliente0_.c_cliente as c_client1_
```

Solo se realiza la consulta a la tabla Cliente para obtener el cliente correspondiente (no los vehículos asociados a el)

Si continuamos la ejecución de la aplicación, al mandarlo a la pagina **cliente.html**, se ejecutará el método **getVehiculos()** que hemos definido en el **th:each**, por lo que veremos la segunda consulta, la cual es un select a la tabla Vehículo que traerá los vehículos para ese cliente:

```
2020-05-13 17:47:23.090 INFO 9540 --- [nio-8080-exec-1] o.s.web
2020-05-13 17:47:23.097 INFO 9540 --- [nio-8080-exec-1] o.s.web
Hibernate: select cliente0_.c_cliente as c_client1_0_0_, cliente
Hibernate: select vehiculos0_.c_cliente as c_client7_1_0_, vehic
```

Efectivamente, ya que hemos definido dicha propiedad como LAZY

Definámosla ahora como EAGER

Si definimos la propiedad como EAGER, al momento de que se ejecuta el método `findOne` del DAO, al revisar la consola, vemos que solo se ejecuta una consulta, pero si la examinamos, en realidad es un `select` con un **join** entre las tablas Cliente y Vehículo, trayendo efectivamente todos los vehículos de dicho cliente (a parte también de los datos del Cliente). Abajo la consulta, en amarillo el **join** que se realiza (por cierto, las consultas que imprime hibernate en la consola se pueden copiar, pegar y luego ejecutar en el PgAdmin4, solo hay que llenar el parámetro en (?)):

```
Hibernate: select cliente0_.c_cliente as c_client1_0_0_, cliente0_.b_activo as b_activo2_0_0_, cliente0_.f_nacimiento as f_nacimi3_0_0_, cliente0_.s_apellidos as s_apelli4_0_0_, cliente0_.s_nombres as s_nombre5_0_0_, vehiculos1_.c_cliente as c_client7_1_1_, vehiculos1_.c_vehiculo as c_vehicu1_1_1_, vehiculos1_.c_vehiculo as c_vehicu1_1_2_, vehiculos1_.b_estado as b_estado2_1_2_, vehiculos1_.c_cliente as c_client7_1_2_, vehiculos1_.f_compra as f_compra3_1_2_, vehiculos1_.s_chassis as s_chassi4_1_2_, vehiculos1_.s_marca as s_marca5_1_2_, vehiculos1_.s_modelo as s_modelo6_1_2_ from store.cliente cliente0_ left outer join store.vehiculo vehiculos1_ on cliente0_.c_cliente=vehiculos1_.c_cliente where cliente0_.c_cliente=?
```

Hibernate realiza un **left outer join** ya que, si el cliente no tiene vehículos asociados, no devolvería ningún registro, por ende, el objeto Cliente vendría **null**. Realizando un **left join** entonces, al no haber registros de vehículos siempre retornaría los datos del cliente, por lo que el objeto Cliente vendría completo, pero su colección de vehículos estaría vacía.