

Relaciones

Relaciones	1
¿Qué aprenderás?	2
Introducción	2
Un pequeño contexto de las relaciones	3
Relación = Cardinalidad	4
Relación 1:1	4
Relación 1:N	5
Relación N:N	7
Trabajar con una relación 1-N en una aplicación JEE	8
Paso 1	8
Paso 2	9
Paso 3	9
Paso 4	10
Relación 1:1	21
Relación mucho a muchos	22
Ejemplo en código de una relación N:N (Muchos a Muchos)	22
Relación Recursiva	24
Ejemplo en código de una relación Recursiva	24



¡Comencemos!

¿Qué aprenderás?

- Entender el contexto de las relaciones.
- Conocer la relación 1:1.
- Conocer la relación 1:N con un ejemplo.
- Conocer la relación N:N.

Introducción

En una base de datos relacional, la interacción y relación entre datos es el punto más importante y de mayor complejidad en un sistema. Todo parte con un buen análisis del negocio en el entorno real, para llevar los datos al ámbito digital mediante repositorios de valores relacionados.

Como ya debes conocer, las bases de datos son conjuntos de tablas que contienen datos **relacionados** entre sí que son fiel reflejo de una realidad de negocio. Se marca en negrita relacionados, ya que como en todo orden de cosas una tabla que no entabla relaciones no es más que un cúmulo de datos sin mayor aporte que guardar datos. Las relaciones permiten que el sistema pueda manejar información y gracias a ellas apoyar la toma de decisiones en una empresa.

Cabe destacar que las relaciones son exclusivas de este tipo de base de datos, pero existen otros tipos de bases de datos que están enfocados a los documentos y no utilizan el concepto de relación. Estamos hablando de las bases de datos no relacionales o NO-SQL.

Un pequeño contexto de las relaciones

El sistema desarrollado en capítulos anteriores tiene dos tablas:

- DEPARTAMENTO.
- EMPLEADO.

Estas tablas modelan la relación entre los trabajadores de la empresa y los distintos departamentos que existen en ella. Si bien es un contexto ideado para representar una relación simple, tiene el potencial de ser mejorado para aplicar problemáticas más complejas, para luego aplicarlas a nuestro sistema. En las siguientes líneas se implementarán más tablas con distintas relaciones, para conocerlas y aplicarlas bien.

Primero recordemos las tablas involucradas:



Imagen 1. Relación empleado - departamento.
Fuente: Desafío Latam.

Existe una variada cantidad de departamentos en la empresa, cada uno en distintos lugares geográficos (distintos países) y por otro lado, tenemos una tabla de empleados, los cuales están asociados a alguno de los departamentos.

Relación = Cardinalidad

Relación 1:1

Una relación uno a uno entre dos tablas es un tipo de relación simple, en donde un registro de la *tabla A* está asociado solo con un registro de la *tabla B*, para crear este tipo de relación se hace mediante las claves primarias (PK) y claves foráneas (FK), donde una clave primaria de la *tabla A* pasa a ser una clave foránea en la *tabla B* generando así la relación entre ambas tablas.

A continuación, podemos ver un ejemplo de este tipo de relación en la siguiente imagen, donde tenemos una tabla "Estudiantes" y una tabla "Información de Contacto", siendo la tabla "Estudiantes" nuestra tabla A y la tabla "Información de Contacto" nuestra tabla B, donde podemos ver que un estudiante puede tener solo un registro de en la tabla de información de contacto.

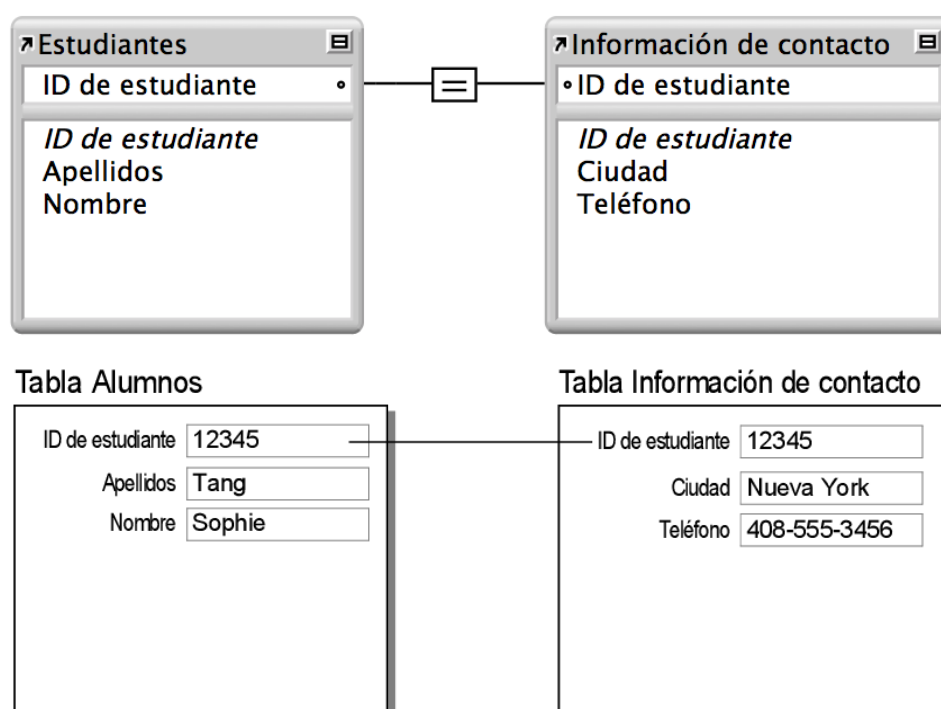


Imagen 2. Relaciones 1 a 1.
Fuente: fmhelp.filemaker.com

Relación 1:N

En una relación uno a muchos (1:N) entre dos tablas, existe una tabla padre o cabecera y una tabla hija o detalle, en donde un registro de la tabla padre puede estar asociado a 1 o muchos registros de la tabla hija, para crear esta relación entre dos tablas se hace mediante las claves primarias (PK) y claves foráneas (FK), donde una clave primaria de la tabla cabecera pasa a ser una clave foránea en la tabla hija generando así la relación entre ambas tablas, y así sucesivamente la misma clave primaria podría estar asociada a muchos registros de la tabla hija creando así el tipo de asociación 1 a muchos (1:N).

A continuación, podemos ver un ejemplo de este tipo de relación en la siguiente imagen, donde tenemos una tabla "Clientes" y una tabla "Pedidos", siendo la tabla "Clientes" nuestra tabla padre y la tabla "Pedidos" nuestra tabla hija, donde se puede interpretar que un cliente puede realizar uno o muchos pedidos de productos.

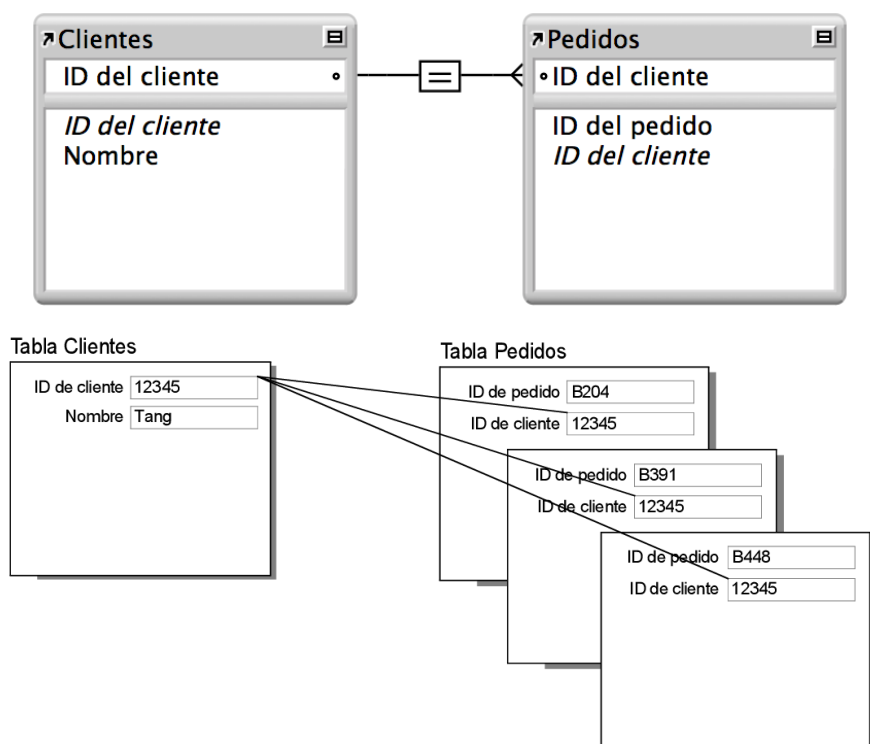


Imagen 3. Relación 1 a N.

Fuente: fmhelp.filemaker.com.

Además, podemos visualizar este tipo de relación en nuestro ejemplo anterior, donde un empleado solo puede pertenecer a un departamento en concreto, por ejemplo, en el departamento de informática. Claro que podría pertenecer al departamento comercial también, pero es raro ver a un informático que también sea ingeniero comercial, y además no podría trabajar en dos partes al mismo tiempo, pero bueno esa es otra historia, el caso es que un empleado puede pertenecer a un departamento de la empresa. Por otro lado, una empresa puede tener asociados uno o muchos empleados en sus dependencias.

Este contexto está catalogado bajo la clásica relación de 1 a N la cual es la más común de encontrar en los modelos de datos.

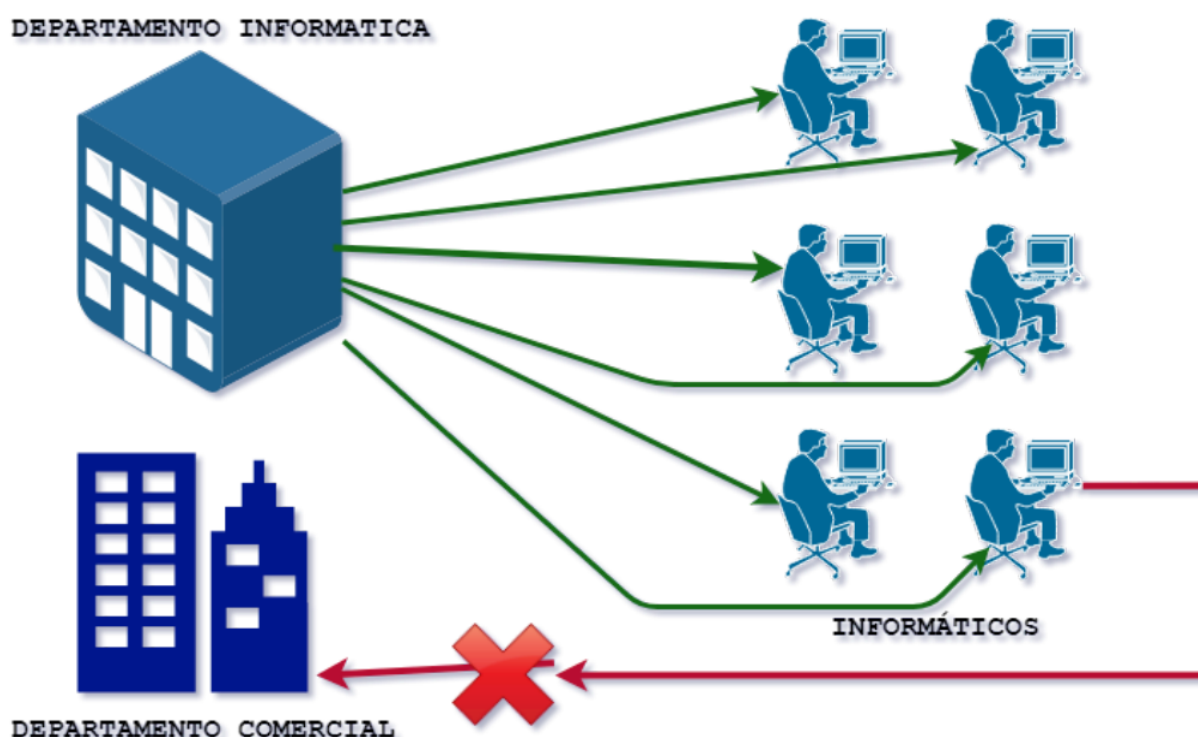


Imagen 4. Ejemplo relación 1 a N.
Fuente: Desafío Latam.

Según este ejemplo, el informático que tiene por ejemplo el id=001 (marcado en rojo) no puede existir en la tabla empleados más de una vez, ya que tiene una clave única (no puede haber más de un registro con el mismo ID) y por ende, no puede pertenecer a más de un departamento.

Relación N:N

Para representar una relación muchos a muchos (N:N) entre dos tablas, se debe crear una tabla intermedia que almacena la clave primaria de la tabla A y la clave primaria de la tabla B, esto porque un registro de la tabla A puede estar asociado a 1 o muchos registros de la tabla B y viceversa, por ende, es esta tercera tabla (C) es la que crea el tipo de asociación muchos a muchos (N:N) entre dos tablas.

A continuación, podemos ver un ejemplo de este tipo de relación en la siguiente imagen, donde tenemos una tabla "Estudiantes", una tabla "Matrículas" y una tabla "Clases", siendo la tabla "Matrículas" nuestra tabla intermedia (C), donde se puede interpretar que un estudiante puede tener una o muchas clases y una clase puede tener uno o muchos estudiantes generando esta relación en nuestra tabla intermedia "Matrículas".

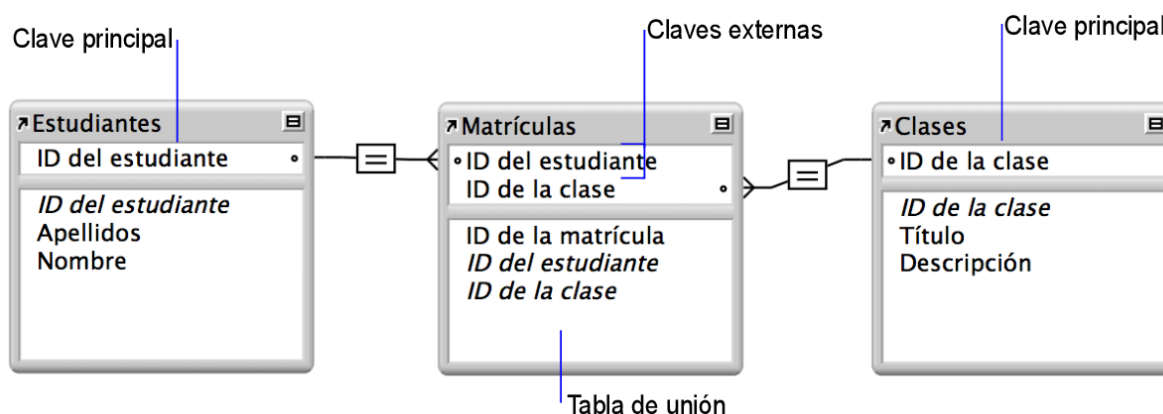


Imagen 5. Relación N a N.
Fuente: fmhelp.filemaker.com.

Trabajar con una relación 1-N en una aplicación JEE

A continuación se implementará un pequeño proyecto con *servlet*, *jsp* y patrón dao para trabajar con una relación 1 a muchos.

Se omitieron los pasos para crear un proyecto web dinámico. Solo iremos a las clases necesarias.

Paso 1

- Generar un proyecto dynamic web project de nombre *EjemploRelaciones*.

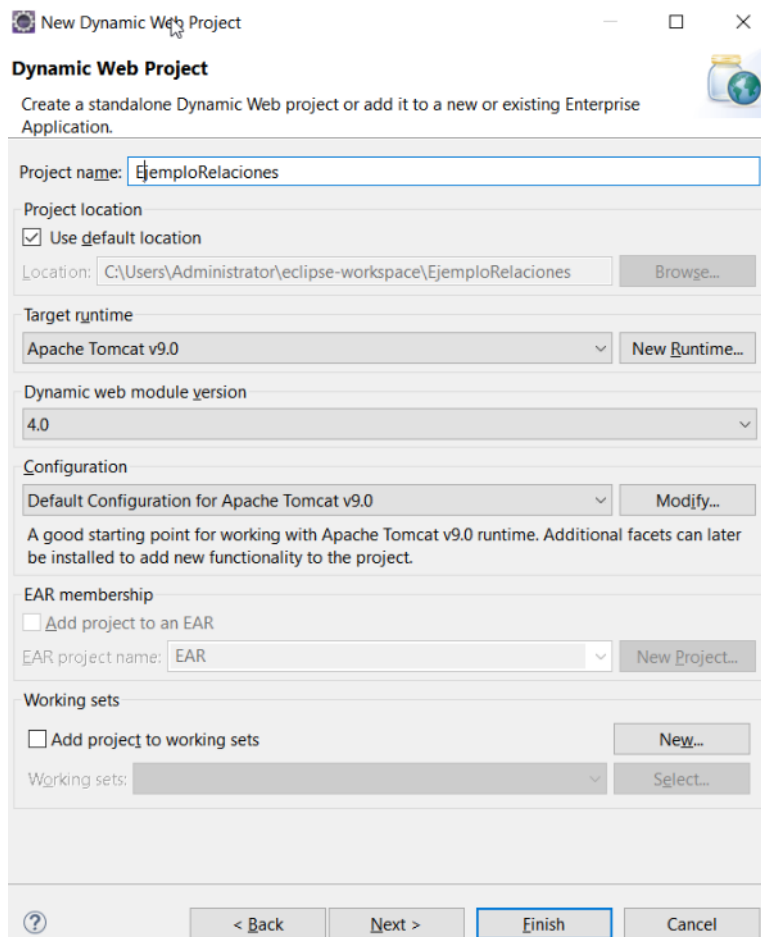


Imagen 6. Creando el proyecto.

Fuente: Desafío Latam.

Paso 2

- Generar un jsp de nombre *ListaUnoMuchos.jsp*

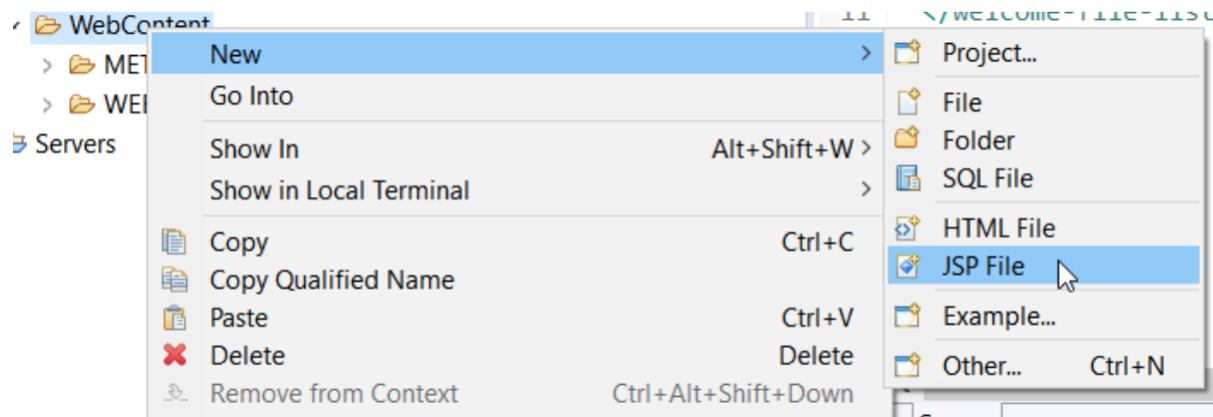


Imagen 7. Crear un JSP.
Fuente: Desafío Latam.

Paso 3

- Importar Bootstrap CDN (librerías alojadas en internet) utilizando el script mostrado a continuación:

```
ListaUnoMuchos.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2   pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4 <html>
5
6 <head>
7   <meta charset="ISO-8859-1">
8   <title>Insert title here</title>
9
10  <!-- importamos bootstrap -->
11  <!-- Latest compiled and minified CSS -->
12  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css" integrity="sha384-BVYiiSIFeK1cGfL7Xr98k7s2/R9WwGDlLpvB9pAnBlEU+dJi0F/r8P99TU0T6o" crossorigin="anonymous">
13
14  <!-- Optional theme -->
15  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap-theme.min.css" integrity="sha384-rHyoN1seRs4N4W5UR6IE90qS/S7WpXHVuwgN6W3OA28A37qNXXPkeMHbUWfxmh" crossorigin="anonymous">
16
17  <!-- Latest compiled and minified JavaScript -->
18  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js" integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWxRxp4v4Lf9CZv3b60273I7y30Fg" crossorigin="anonymous">
19
20
21 </head>
22 <body>
23
24 </body>
25 </html>
```

Imagen 8. Importando Bootstrap.
Fuente: Desafío Latam.

Paso 4

- Añadir dentro del body un jumbotron, un navbar y un div container propios de bootstrap, para ver la plantilla ir al siguiente [link](#).

```
23 <body>
24
25 <div class="jumbotron text-center" style="margin-bottom: 0">
26   <h1>Ejemplo relaciones con JSP</h1>
27   <p>Para el curso de JEE</p>
28 </div>
29
30 <nav class="navbar navbar-expand-sm bg-dark navbar-dark">
31   <a class="navbar-brand" href="#">Navegacion</a>
32   <button class="navbar-toggler" type="button" data-toggle="collapse"
33     data-target="#collapsibleNavbar">
34     <span class="navbar-toggler-icon"></span>
35   </button>
36   <div class="collapse navbar-collapse" id="collapsibleNavbar">
37     <ul class="navbar-nav">
38       <li class="nav-item"><a class="nav-link" href="#">Uno a
39         muchos</a></li>
40       <li class="nav-item"><a class="nav-link" href="#">En
41         construcción</a></li>
42       <li class="nav-item"><a class="nav-link" href="#">En
43         construcción</a></li>
44     </ul>
45   </div>
46 </nav>
47
48 <div class="container" style="margin-top: 30px">
49
50 </div>
51
52 </body>
```

Imagen 9. Creando la vista.
Fuente: Desafío Latam.

Con estos pasos, solamente tenemos una interfaz más amigable.



Imagen 10. Vista implementada.

Fuente: Desafío Latam.

El verdadero trabajo comienza ahora.

Si volvemos a pensar en las tablas vemos que tenemos una tabla de departamentos que puede tener muchos o un empleado, y a su vez una tabla empleados en donde un mismo empleado no puede pertenecer a más de un departamento.

El usuario obviamente no conoce sql y quiere saber, mediante una interface web, cuántos empleados están asociados al departamento, por ejemplo, el de informática. Para ello, primero debemos pensar en la consulta que nos traerá los datos, ya que el *select * from* nos quedará corto porque debemos relacionar 2 tablas n_1. Nuestra solución está en una sentencia con inner join para traer a todos los empleados de un departamento en particular.

```
SELECT EMP.NUMEMPLEADO, EMP.NOMBRE, DEP.NOMDEPTO  
FROM EMPLEADO EMP INNER JOIN DEPARTAMENTO DEP ON DEP.NUMDEPTO = EMP.NUMDEPTO  
WHERE DEP.NOMDEPTO = 'INFORMATICA';
```

Imagen 11. Consulta INNER JOIN entre empleado y departamento.

Fuente: Desafío Latam.

Esta query nos rescata lo que el usuario desea (todos los empleados que estén en el departamento de informática) por lo cual debemos guardarla para aplicarla en nuestra clase *daoImplement* que aún no creamos, pero que lo haremos enseguida. Pero antes debemos armar el flujo así que volvemos a código.

Continuamos en el JSP. Ya se tiene una interfaz en donde desplegar los datos, pero el usuario desea el mismo aplicar el criterio de búsqueda, por lo cual necesita de unos filtros. Procedemos a crear un pequeño modelo de filtros con solo un campo de texto, una tabla y un botón de búsqueda.

```

48= <div class="container" style="margin-top: 30px">
49=   <div class="row">
50=     <div class="col-sm-3">
51=       <h2>departamento</h2>
52=     </div>
53=
54=     <div class="col-sm-9">
55=       <h2>resultados</h2>
56=     </div>
57=   </div>
58=   <br>
59=   <div class="row">
60=     <form action="procesaBusquedaEmplDept" method="post">
61=       <div class="col-sm-3">
62=         <label for="NOMBRE DEPARTAMENTO">Nombre Departamento:</label>
63=         <input type="text" class="form-control" id="nomDepto" name="nomDepto">
64=         <br>
65=         <button type="button" class="btn btn-primary">Buscar</button>
66=       </div>
67=       <div class="col-sm-9">
68=         <div id="tabla">
69=           <br>
70=           <table class="table table-sm table-dark">
71=             <thead>
72=               <tr>
73=                 <th scope="col">Numero Empleado</th>
74=                 <th scope="col">Nombre Empleado</th>
75=                 <th scope="col">Nombre Departamento</th>
76=               </tr>
77=             </thead>
78=             <tbody>
79=               <td>Marco Zaror</td>
80=               <td>0458</td>
81=               <td>CHILE</td>
82=             </tbody>
83=           </table>
84=         </div>
85=       </div>
86=     </form>
87=   </div>
88= </div>

```

Imagen 12. Mejorando la vista de búsqueda.

Fuente: Desafío Latam.

Solamente se aplicó un poco de bootstrap, lo importante aquí es la tabla, el botón y sobre todo la etiqueta form, la cual tiene la referencia del servlet que procesa la búsqueda. Esta página se debe ver así:



Imagen 13. Resultado de la implementación de la vista.
Fuente: Desafío Latam.

Teniendo el formulario completo, debemos fijarnos en el botón el cual no es de tipo submit por lo cual al pincharlo no procesa la petición y jamás llegará al *servlet*, siempre tener ojo con este tema. El botón que quiera procesar una petición debe ser de tipo *submit*. Hacemos el cambio de tipo del botón para poder continuar.

Bien, tenemos la interface con un filtro que permitirá enviar un texto al *backend*, y una tabla que desplegará el resultado. Ahora es tiempo de trabajar con el *servlet*. Para crearlo, solamente se debe ir a la carpeta *Java Resources* y pinchar con botón derecho, para crear un nuevo *package* de nombre *com.desafiolatam.servlet*:

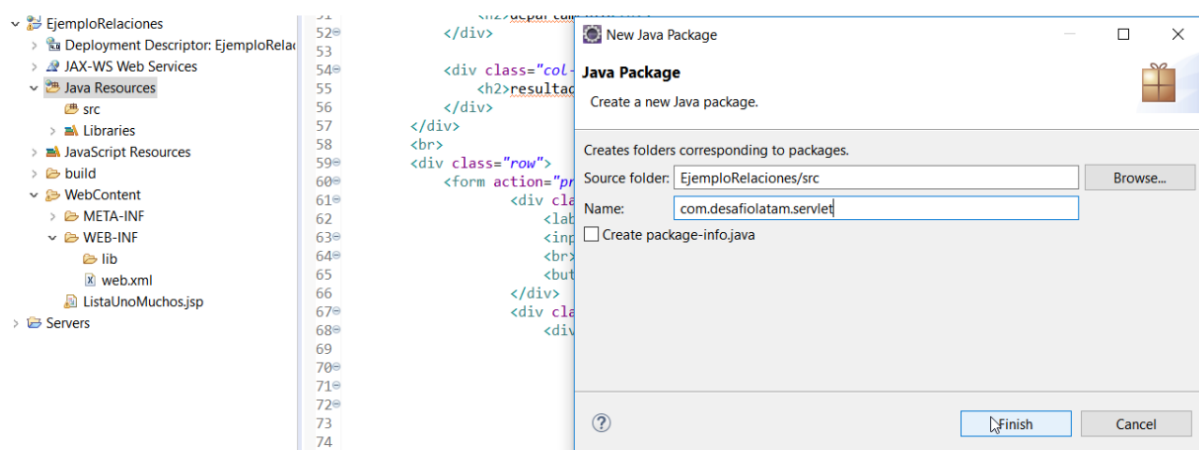


Imagen 14. Creando Servlet.
Fuente: Desafío Latam.

Dentro del package, crearemos un nuevo *servlet* de nombre *ProcesaBusquedaEmpDept*. Al crear el nuevo *servlet*, eclipse genera varios métodos de forma automática así que tenemos que borrarlos y solo quedar con el método *doPost* tal como se muestra en la imagen.

```
1 package com.desafiolatam.servlet;
2
3 import java.io.IOException;
4
5
6
7
8
9
10
11 @WebServlet("/procesaBusquedaEmplDept")
12 public class ProcesaBusquedaEmplDept extends HttpServlet {
13     private static final long serialVersionUID = 1L;
14
15     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
16
17     }
18
19 }
```

Imagen 15. Manteniendo el método doPost.

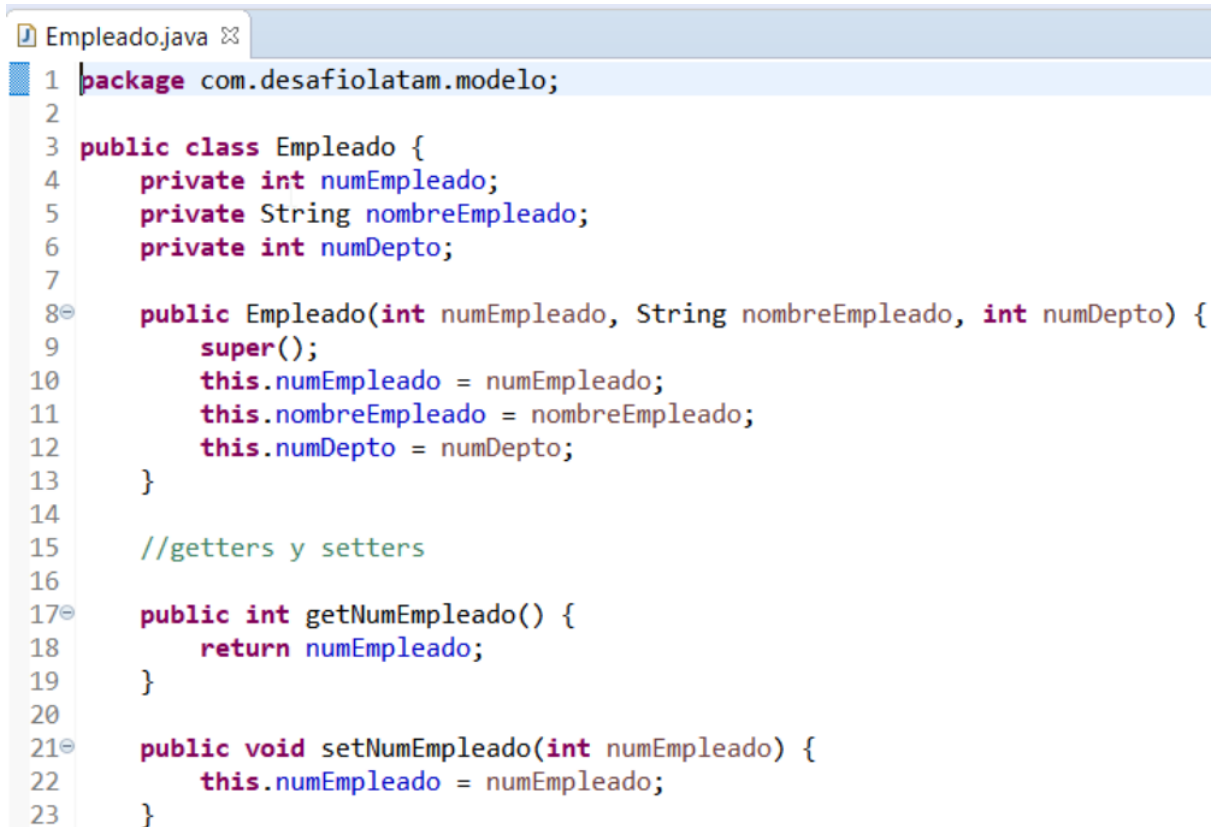
Fuente: Desafío Latam.

Este es el servlet que recibe la petición y debe conocer a qué clases llamar para generar la consulta a la base de datos. Ahora es tiempo de generar los objetos de negocio que son espejo de las entidades reales, en este caso necesitamos una clase Departamento y una clase Empleado. Crear estas clases en un package de nombre *com.desafiolatam.modelo*. (se omiten en la imagen los setter y getter).

```
Departamento.java
1 package com.desafiolatam.modelo;
2
3 public class Departamento {
4     private int numDepto;
5     private String nombreDepto;
6     private String ubicacionDepto;
7
8     public Departamento(int numDepto, String nombreDepto, String ubicacionDepto) {
9         super();
10        this.numDepto = numDepto;
11        this.nombreDepto = nombreDepto;
12        this.ubicacionDepto = ubicacionDepto;
13    }
14
15    //getters y setters
16
17    public int getNumDepto() {
18        return numDepto;
19    }
20
21    public void setNumDepto(int numDepto) {
22        this.numDepto = numDepto;
23    }
```

Imagen 16. Creando el modelo de Departamento.

Fuente: Desafío Latam.



```
Empleado.java
1 package com.desafiolatam.modelo;
2
3 public class Empleado {
4     private int numEmpleado;
5     private String nombreEmpleado;
6     private int numDepto;
7
8     public Empleado(int numEmpleado, String nombreEmpleado, int numDepto) {
9         super();
10        this.numEmpleado = numEmpleado;
11        this.nombreEmpleado = nombreEmpleado;
12        this.numDepto = numDepto;
13    }
14
15    //getters y setters
16
17    public int getNumEmpleado() {
18        return numEmpleado;
19    }
20
21    public void setNumEmpleado(int numEmpleado) {
22        this.numEmpleado = numEmpleado;
23    }
}
```

Imagen 17. Creando modelo de Empleado.
Fuente: Desafío Latam.

Estas dos clases de modelo son quienes transportarán los valores entre la implementación dao que se comunicará con la base de datos hasta el servlet que los devolverá a la vista jsp.

Aún tenemos un tema que resolver. El usuario quiere ver todos los empleados que pertenecen a un departamento así que pensando en objetos, tenemos que recibir en el jsp un departamento que contiene en su estructura una lista de muchos empleados. Para simular este comportamiento podemos crear una nueva clase de nombre *DepartamentoEmpleado.java* la cual tendrá como parámetros una clase *Departamento* y una clase *Empleado*.

Cuando obtengamos los valores de la base de datos tendremos que crear una instancia de esta nueva clase y añadir empleados y departamentos. Primero veamos la estructura de esta nueva clase.

```
1 package com.desafiolatam.modelo;
2
3 public class DepartamentoEmpleado {
4     private Departamento departamento;
5     private Empleado empleado;
6
7     public DepartamentoEmpleado(Departamento departamento, Empleado empleado) {
8         super();
9         this.departamento = departamento;
10        this.empleado = empleado;
11    }
12
13    public Departamento getDepartamento() {
14        return departamento;
15    }
16    public void setDepartamento(Departamento departamento) {
17        this.departamento = departamento;
18    }
19    public Empleado getEmpleado() {
20        return empleado;
21    }
22    public void setEmpleado(Empleado empleado) {
23        this.empleado = empleado;
24    }
25
26 }
27 }
```

Imagen 18. Estructura de la clase DepartamentoEmpleado.

Fuente: Desafío Latam.

Como se puede apreciar, es una nueva clase que contiene dos objetos de tipo departamento y empleado.

Ahora es tiempo de crear las clases que se encargarán de contactar con la base de datos. Aquí replicaremos la estructura del patrón DAO que se vio en capítulos anteriores, ya que utilizaremos la creación de un pool de conexiones y la estructura de clases del patrón. Vamos a crear los dos paquetes de nombre:

- com.desafiolatam.procesaconexion
- com.desafiolatam.dao

Dentro del package *procesaconexion* creamos la clase *AdministradorConexion* y generamos la conexión.


```
1 package com.desafiolatam.procesaconexion;
2
3*import java.sql.Connection;
13
14 public class AdministradorConexion {
15
16     protected Connection conn = null;
17     protected PreparedStatement pstmt = null;
18     protected ResultSet rs = null;
19
20     protected Connection generaConexion() {
21         String usr = "sys as sysdba";
22         String pwd = "admin";
23         String driver = "oracle.jdbc.driver.OracleDriver";
24         String url = "jdbc:oracle:thin:@//localhost:1521/desafio_ejemplo01";
25
26         try {
27             Class.forName(driver);
28             conn = DriverManager.getConnection(url,usr,pwd);
29         } catch (Exception ex) {
30             ex.printStackTrace();
31         }
32         return conn;
33     }
34
35     protected Connection generaPoolConexion() {
36         Context initContext;
37         try {
38             initContext = new InitialContext();
39             DataSource ds = (DataSource) initContext.lookup("java:/comp/env/jdbc/ConexionOracle");
40             try {
41                 conn = ds.getConnection();
42             } catch (SQLException e) {
43                 e.printStackTrace();
44             }
45         } catch (NamingException e) {
46             e.printStackTrace();
47         }
48         return conn;
49     }
50
51 }
```

Imagen 19. Clase AdministradorConexión.

Y nos hacen falta las clases DAO, las cuales crearemos en el paquete del mismo nombre. Recordemos que el patrón dao consta de una interfaz que define los métodos y una clase que implementa dicha interfaz y su código.

```
1 package com.desafiolatam.dao;
2
3*import java.util.List;
7
8 public interface DepartamentoEmpleadoDao {
9     public List<DepartamentoEmpleado> obtieneDepartamento(String nomDepto);
10 }
11
```

Imagen 20. Creando interfaz DepartamentoEmpleadoDAO.

Fuente: Desafío Latam.

```
10 import com.desafiolatam.modelo.Empleado;
11 import com.desafiolatam.procesaconexion.*;
12
13 public class DepartamentoEmpleadoDaoImpl extends AdministradorConexion implements DepartamentoEmpleadoDao {
14
15     public DepartamentoEmpleadoDaoImpl() {
16         Connection conn = generaPoolConexion();
17     }
18
19     @Override
20     public List<DepartamentoEmpleado> obtieneDepartamento(String nomDepto) {
21         List<DepartamentoEmpleado> deptosEmpleados = new ArrayList<DepartamentoEmpleado>();
22         String query = "SELECT * FROM EMPLEADO EMP INNER JOIN DEPARTAMENTO DEP ON DEP.NUMDEPTO = EMP.NUMDEPTO\r\n" +
23             "WHERE ";
24
25         if((nomDepto.isEmpty() && nomDepto.isEmpty()) ) {
26             query = "SELECT * FROM EMPLEADO EMP INNER JOIN DEPARTAMENTO DEP ON DEP.NUMDEPTO = EMP.NUMDEPTO";
27         }else {
28             query += "DEP.NUMDEPTO = '"+nomDepto+"'";
29         }
30
31         try {
32             pstmt = conn.prepareStatement(query);
33             rs = pstmt.executeQuery();
34             while(rs.next()) {
35                 Departamento depto = new Departamento(rs.getInt("NUMDEPTO"), rs.getString("NOMDEPTO"), rs.getString("UBICACIONDEPTO"));
36                 Empleado empleado = new Empleado(rs.getInt("NUMEMPLEADO"), rs.getString("NOMBRE"),rs.getInt("NUMDEPTO"));
37                 DepartamentoEmpleado deptoEmpl = new DepartamentoEmpleado(depto,empleado);
38                 deptosEmpleados.add(deptoEmpl);
39             }
40         } catch (SQLException e) {
41             e.printStackTrace();
42         }
43         return deptosEmpleados;
44     }
45 }
46 }
```

Imagen 21. Implementando el patrón DAO.
Fuente: Desafío Latam.

Atención al código de la implementación.

- En la línea 21 se genera una lista de tipo *DepartamentoEmpleado* la cual usaremos para devolver los valores rescatados de la *query*.
- La línea 22 contiene la *query* con el *inner join*, en donde como resultado obtendremos los datos de las dos tablas. Por esta razón creamos una nueva clase que al igual que la respuesta, soporta los dos tipos de datos que vamos a obtener (departamentos y empleados).
- En la línea 34 en donde recorremos el *resultset* para rescatar los valores, creamos dos objetos, uno de tipo *Empleado* y otro de tipo *Departamento* los cuales son inicializados con los valores retornados desde la base de datos (mediante su constructor). Luego de tener los dos objetos, los añadimos a la lista de tipo *DepartamentoEmpleado*, que como se mostró anteriormente espera dos objetos de tipo empleado y departamento.

Con esto ya tenemos el patrón dao funcionando así que es necesario volver al *serv/let*, el cual debe recibir el resultado para enviarlo a la vista.

```
17 @WebServlet("/procesaBusquedaEmplDept")
18 public class ProcesaBusquedaEmpDept extends HttpServlet {
19     private static final long serialVersionUID = 1L;
20
21     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
22         String nomDepartamento = (String) request.getParameter("nomDepto");
23
24         List<DepartamentoEmpleado> deptosEmpleadosList = new ArrayList<DepartamentoEmpleado>();
25
26         DepartamentoEmpleadoDaoImpl obtieneDeptoEmpleado = new DepartamentoEmpleadoDaoImpl();
27
28         deptosEmpleadosList = obtieneDeptoEmpleado.obtieneDepartamento(nomDepartamento);
29
30         request.setAttribute("departamentoEmpleado", deptosEmpleadosList);
31
32         request.getRequestDispatcher("ListaUnoMuchos.jsp").forward(request, response);
33     }
34 }
35 }
```

Imagen 22. Procesando el resultado para mostrarlo a la vista.

Fuente: Desafío Latam.

El *servlet* como buen controlador no maneja lógica de negocio, solamente hace las llamadas correspondientes al *dao* y devuelve los valores a la vista.

- En la línea 24, se crea una lista de tipo *DepartamentoEmpleado* y a continuación se crea una instancia del *dao* *DepartamentoEmpleadoDaoImpl* para utilizar el método *obtieneDepartamento*, que como se vio anteriormente recibe como parámetro el nombre del elemento a buscar.
- Luego de obtener los valores y guardarlos en la lista creada utilizando el objeto *request*, añade la lista obtenida y luego redirige a la vista pasando tal *request*.

Ahora el *jsp* está listo para recibir los datos y desplegarlos en pantalla.

Para esto, tenemos que instanciar como si una clase se tratara a la lista de *DepartamentoEmpleado* para luego mediante la variable *request* obtener el atributo *departamentoEmpleado*.

```
23 <%
24     List<DepartamentoEmpleado> deptos = new ArrayList<DepartamentoEmpleado>();
25     deptos = (List)request.getAttribute("departamentoEmpleado");
26 %>
27 </head>
```

Imagen 23. Instanciando la clase *DepartamentoEmpleado*.

Fuente: Desafío Latam.

Ahora se interviene la tabla, generando un ciclo *for* recorriendo la lista y desplegando los valores en cada celda:

```
<table class="table table-sm table-dark">
  <thead>
    <tr>
      <th scope="col">Numero Empleado</th>
      <th scope="col">Nombre Empleado</th>
      <th scope="col">Nombre Departamento</th>
    </tr>
  </thead>
  <tbody>
    <%
      if(deptos != null){
        for(DepartamentoEmpleado depto: deptos){
          %>
          <tr>
            <td><%=depto.getEmpleado().getNumEmpleado()%></td>
            <td><%=depto.getEmpleado().getNombreEmpleado()%></td>
            <td><%=depto.getDepartamento().getNombreDepto()%></td>
          </tr>
          <%=}}%>
        </tbody>
      </table>
```

Imagen 24. Mostrando los elementos de la lista.

Fuente: Desafío Latam.

Se puede apreciar que podemos acceder a la lista de empleados de un departamento por las llamadas que se hacen en cada celda. Fijarse que desde depto obtenemos el nombre y el número de empleado y para obtener el nombre del departamento usamos `depto.getDepartamento().getNombreDepto`. Para esto se utilizó una clase que contiene a las dos entidades, para un uso más natural e intuitivo.

Navegación

Uno a muchos

En construcción

En construcción

departamento

resultados

Nombre Departamento:

CONTABILIDAD

Buscar

Numero Empleado	Nombre Empleado	Nombre Departamento
31	Kilos Demetrio	CONTABILIDAD
32	Astorga Daniel	CONTABILIDAD
33	Yeny Marlen	CONTABILIDAD
34	Julia Zapata	CONTABILIDAD
35	Esteban Cerrado	CONTABILIDAD
41	Ricardo Abrilar	CONTABILIDAD
42	Rocio Vicencio	CONTABILIDAD
43	Bianca Vicencio	CONTABILIDAD

Imagen 25. Resultado de la consulta.

Fuente: Desafío Latam.

Ahora el usuario puede ver cuántos empleados existen en cada departamento.

Relación 1:1

Por cada registro de la tabla principal solo existe un registro en su tabla relacionada. Para entender este tipo de relación, podemos pensar en la tabla de usuarios registrados, la cual tiene un id de usuario. Pensemos que existe una nueva tabla que guarda el registro correspondiente a sus documentos de extranjería, y si analizamos el caso, cada persona (usuario) solo puede tener un único id de pasaporte y a su vez un pasaporte solo puede pertenecer a una sola persona.

Como aprendimos en el ejercicio anterior, tenemos que pensar en la salida que el usuario desea ver y también en que el usuario no sabe hacer queries, por lo tanto, debemos facilitar una consulta mediante un formulario jsp y la arquitectura *servlet-dao* que se implementó anteriormente.

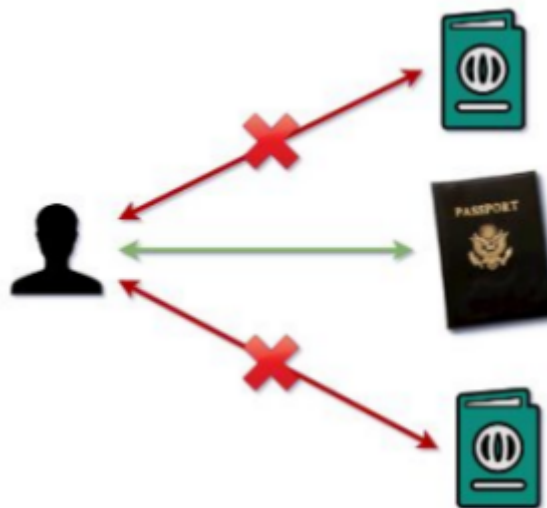


Imagen 26. Relación 1 a 1.
Fuente: Desafío Latam.

Para implementar una llamada a relación 1 a 1 simplemente creamos en la base de datos una nueva tabla de registro de datos de documentos extranjería y la asociaremos a la tabla empleados.

```
CREATE TABLE DOC_EXTRANJERIA (ID_PASAPORTE NUMBER PRIMARY KEY, PAIS_ORIGEN VARCHAR(100), DESCRIPCION VARCHAR(100));  
ALTER TABLE DOC_EXTRANJERIA ADD CONSTRAINT DOC_EXTRANJERIA_EMPL FOREIGN KEY (ID_PASAPORTE) REFERENCES EMPLEADO (NUMEMPLEADO) ON DELETE CASCADE;
```

Imagen 27. Creando tabla DOC_EXTRANJERIA.
Fuente: Desafío Latam.

Relación mucho a muchos

Una relación N:N se produce básicamente cuando muchos elementos de una tabla se relacionan con muchos elementos de otra tabla. Para graficar esta situación pensemos en la relación entre un empleado y productos, en donde un empleado puede tener muchos productos y un producto puede pertenecer a muchos empleados.

En el diseño de un modelo de datos no se recomienda esta relación por el hecho de que no es posible identificar un producto en particular en contra de un empleado, por lo cual para subsanar este problema, es que se debe crear una tabla que rompa esta relación.

Una tabla intermedia entre usuarios y productos que guardará en ella el id del empleado, el id del producto y, por ejemplo, la descripción del producto. Con esta tabla intermedia ya se puede identificar qué producto pertenece a un empleado en particular.

Ejemplo en código de una relación N:N (Muchos a Muchos)

El ejemplo consiste en que un Profesor enseña muchas unidades de un curso y a su vez una Unidad puede tener muchos profesores que la imparten.

Clase Profesor

```
@Entity
@Table(name="Profesor")
public class Profesor implements Serializable {
    @Id
    @Column(name="Id")
    private int id;
    @Column(name="nombre")
    private String nombre;
    @Column(name="email")
    private String email;
    @ManyToMany(cascade = {CascadeType.ALL})
    @JoinTable(name="ProfesorUnidad",
joinColumns={@JoinColumn(name="IdProfesor")},
inverseJoinColumns={@JoinColumn(name="IdUnidad")})
    private Set<Unidad> unidades=new HashSet();
    public Profesor(){
    }
    public Profesor(int id, String nombre, String email) {
```

```
        this.id = id;
        this.nombre = nombre;
        this.email = email;
    }
    //agregar Getter y Setter
}
```

Clase Unidad

```
@Entity
@Table(name="Unidad")
public class Unidad implements Serializable {
    @Id
    @Column(name="IdUnidad")
    private int IdUnidad;
    @Column(name="nombre")
    private String nombre;

    @ManyToMany(cascade = {CascadeType.ALL},mappedBy="unidades")
    private Set<Profesor> profesores=new HashSet();
    public Unidad() {
    }
    public Unidad(int IdUnidad, String nombre) {
        this.IdUnidad = IdUnidad;
        this.nombre = nombre;
    }
    //agregar Getter y Setter
}
```

Relación Recursiva

Existe un tipo de relación bastante particular, el cual indica que si una tabla se relaciona consigo misma estamos en presencia de una relación recursiva o asociada a sí misma. Por ejemplo, un empleado-supervisor en donde desempeña un rol de supervisor en un lado de la relación y por el otro lado desempeña el rol de empleado. Los esquemas de asignación pueden incluir relaciones recursivas donde un elemento y su antecesor son del mismo tipo.

Ejemplo en código de una relación Recursiva

El ejemplo trata de que un directorio puede tener archivos y a su vez otros directorios que contendrán otros archivos y directorios (carpetas). Por lo tanto, se trata de una relación recursiva. El concepto que tenemos que entender es que una carpeta también es un directorio.

Clase Directorio

```
public abstract class Directorio {  
    private String nombre;  
    public String getNombre() {  
        return nombre;  
    }  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
    public Directorio(String nombre) {  
        super();  
        this.nombre = nombre;  
    }  
    public abstract boolean esCarpeta() ;  
    public abstract List<Directorio> getDirectorios();  
}
```


Clase Archivo

```
public class Archivo extends Directorio {
    public Archivo(String nombre) {
        super(nombre);
    }
    @Override
    public boolean esCarpeta() {
        return false;
    }
    @Override
    public List<Directorio> getDirectorios() {
        throw new RuntimeException(" un archivo no contiene
directorios");
    }
}
```

Clase Carpeta

```
public class Carpeta extends Directorio{
    public Carpeta(String nombre) {
        super(nombre);
    }
    private List<Directorio> directorios= new ArrayList<>();
    public List<Directorio> getDirectorios() {
        return directorios;
    }
    public void setDirectorios(List<Directorio> directorios) {
        this.directorios = directorios;
    }
    public void addDirectorio (Directorio d) {
        directorios.add(d);
    }
    @Override
    public boolean esDirectorio() {
        return true;
    }
}
```