Unión Europea
Fondo Social Europeo
*El FSE invierte en tu futuro*

| Fecha | Versión | Descripción |
|---|---|---|
| 05/12/2022 | 1.0.0 | Versión inicial. |
| 28/12/2022 | 1.0.1 | Relación muchos a muchos con entidad en medio. Faltaba equals() y hashset() en Recomendacion. Arreglo constructor Cliente. Modifición de ClienteDTO y Cliente para añadir el HashSet a lista de cuentas. |
| 11/01/2023 | 1.0.2 | Borrado de hijo-padre en relaciones 1 a N. |
| 11/02/2023 | 1.0.3 | Almacenamiento de relación N a N con valor intermedio. Inserción y actualización. |
| 28/11/2024 | 2.0.0 | Revisión y actualización del material. |

# Unidad 5 - Mapeo Objeto-Relacional con JPA en Spring Boot


**Unidad 5 - Mapeo Objeto-Relacional con JPA en Spring Boot**
**El Desfase Objeto-Relacional**
**Comenzando con JPA**
- Configurando la aplicación
- Definiendo la base de datos a trabajar
- Preparando la conexión entre la aplicación Spring Boot y MySQL
- Comenzando a mapear entidades con JPA
- Modificando los repositorios de datos
- Probando la aplicación.

**Mapeando relaciones (Parte 1 de 2)**
- Introducción
- Relaciones 1 a 1
- ¿Cuándo utilizar Lazy Fetch y cuando Eager Fetch?
- Relaciones 1 a N
  - Eliminando elementos hijos de una relación 1 a N
- Relaciones N a N sin atributo en medio de la relación y sin mapear entidad del medio
- Relaciones N a N sin atributo en medio de la relación y mapeando la entidad del medio.

**El lenguaje de consultas JPQL**
**Extendiendo los repositorios**
**Mapeando relaciones (Parte 2 de 2)**
- Relaciones N a N con atributo en medio de la relación y mapeando la entidad del medio. Trabajando con fechas.

# El Desfase Objeto-Relacional

Es el momento de comenzar a trabajar con nuestra aplicación en SpringBoot sobre una base de datos, que en este caso vamos a utilizar MySQL.

El problema que tenemos es que MySQL es una base de datos relacional y nosotros estamos trabajando en SpringBoot con un lenguaje orientado a objetos, que es Java. Con esto nos referimos a que, en MySQL vamos a tener la tabla clientes con sus filas y columnas, mientras que en java tendremos la clase `Cliente.java` con sus propiedades y métodos.

Por ejemplo, en el proyecto de la unidad anterior, la clase `Cliente.java` tenía el siguiente aspecto, con sus atributos y métodos.

```java
package com.example.demo.repository.entity;

import java.util.Objects;

import lombok.Data;

@Data
public class Cliente {

  private Long id;
  private String nif;
  private String nombre;
  private String apellidos;
  private String claveSeguridad;
  private String email;
  private Recomendacion recomendacion;

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Cliente other = (Cliente) obj;
    return Objects.equals(id, other.id);
  }

  @Override
  public int hashCode() {
    return Objects.hash(id);
  }
}
```

Mientras que en la base de datos tendremos una tabla cuyos campos se tendrán que corresponder con los atributos que hayamos definido anteriormente en esa clase. Puesto que son estructuras que no tienen nada que ver entre ellas, tenemos que hacer el mapeo manualmente, haciendo coincidir (a través de los getters o setters) cada uno de los atributos con cada uno de los campos (y vicerversa) cada vez que queramos leer o escribir un objeto desde y hacia la base de datos, respectivamente. Siguiendo con el ejemplo de la entidad `Cliente`, una tabla de clientes para mantener los datos sería la siguiente:

```
1  CREATE TABLE `demospringboot`.`clientes` (
2      `id` INT NOT NULL AUTO_INCREMENT,
3      `nif` VARCHAR(9) NOT NULL,
4      `nombre` VARCHAR(100) NOT NULL,
5      `apellidos` VARCHAR(100) NOT NULL,
6      `claveseguridad` VARCHAR(10) NOT NULL,
7      `email` VARCHAR(100) NOT NULL,
8      PRIMARY KEY (`id`));
```

Eso hace que tengamos que estar continuamente descomponiendo los objetos para escribir la sentencia SQL para insertar, modificar o eliminar, o bien recomponer todos los atributos para formar el objeto cuando leamos algo de la base de datos. Esto nos pasaría si estuviéramos trabajando con JDBC sobre nuestro proyecto, puesto que cuando tuviéramos que realizar una operación con clientes, por ejemplo, insertar uno, tendríamos que construir la sentencia SQL y realizar la descomposición del objeto para poder insertarlo (esto ya se hacia el curso pasado). En el siguiente código vemos a que nos referimos:

```
135     public void insert(Cliente cliente) {
136     try {
137         Class.forName("com.mysql.jdbc.Driver");//Cargar el driver
138         // Establecemos la conexion con la BD
139         Connection conexion = DriverManager.getConnection("jdbc:mysql://localhost/demospringboot",
140             "profesor", "12345");
141
142         //construir orden INSERT
143         String sql= "INSERT INTO cliente(nif, nombre, apellidos, claveseguridad, email) "
144                 + " VALUES (?,?,?,?,?);";
145         PreparedStatement sentencia = conexion.prepareStatement(sql);
146         sentencia.setString(1, cliente.getNif());
147         sentencia.setString(2, cliente.getNombre());
148         sentencia.setString(3, cliente.getApellidos());
149         sentencia.setString(4, cliente.getClaveSeguridad());
150         sentencia.setString(5, cliente.getEmail());
151         sentencia.executeUpdate();
152
153         if (sentencia != null) {
154             sentencia.close();           // Cerrar Statement
155             conexion.close();            //Cerrar conexión
156         }
157     } catch (ClassNotFoundException cn) {
158         cn.printStackTrace();
159     }
160     catch (SQLException e) {
161         e.printStackTrace();
162     }
163
164 }
```

Este ejemplo de cómo insertar un cliente con JDBC en la base de datos demuestra como tenemos que mapear a mano la inserción del cliente. Por todo esto, necesitamos acudir a un mapeo objeto-relacional que no automatice este proceso.

El mapeo objeto-relacional se va a realizar mediante un framework denominado Hibernate, que permite que esta misma operación se traduzca en unas pocas líneas de código en las que podemos trabajar directamente con el objeto Java, puesto que el framework realiza el mapeo en función de las anotaciones que hemos implementado a la hora de definir la clase, que le indican que tabla y campos de la misma se corresponde la clase y sus atributos, respectivamente.

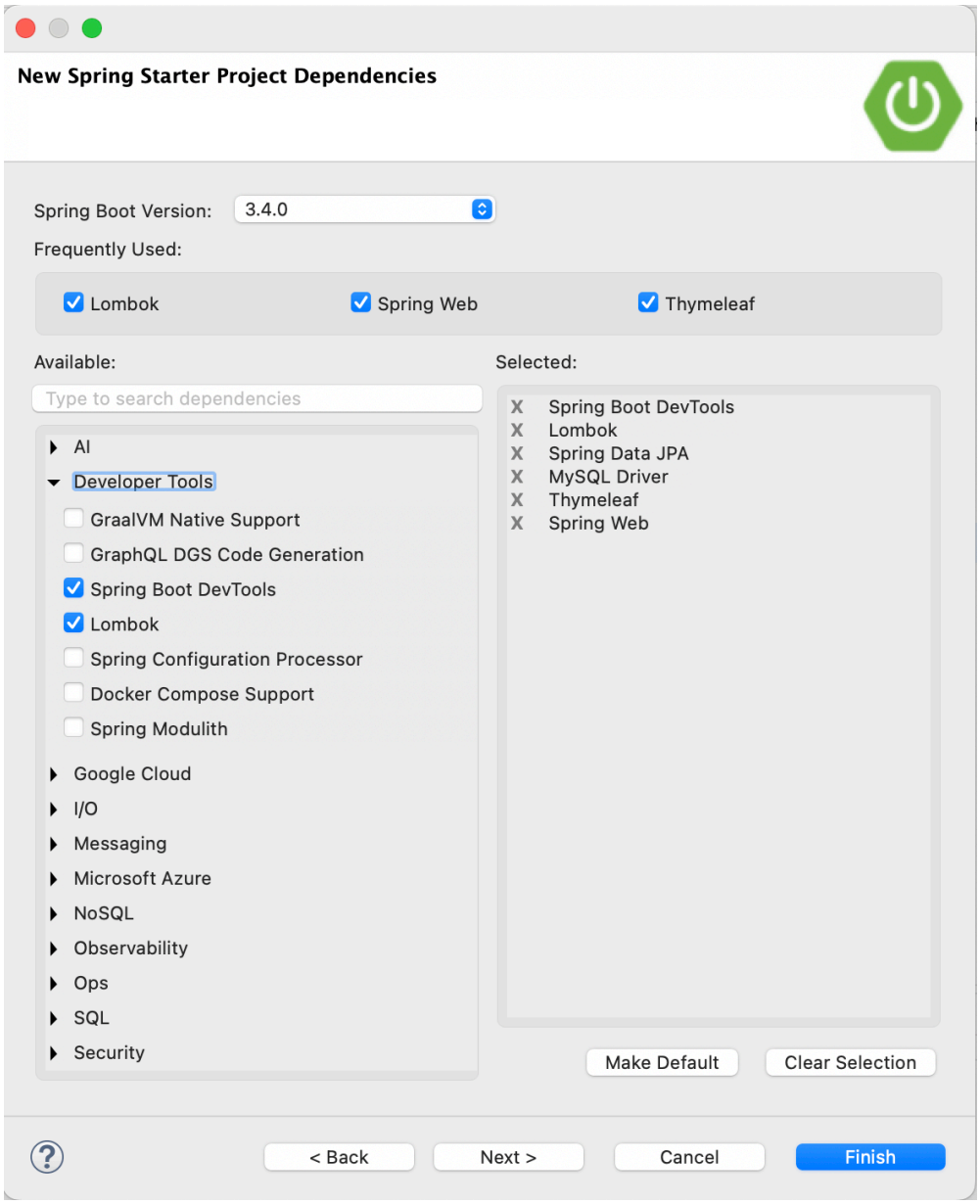
# Comenzando con JPA

## Configurando la aplicación

A nuestra aplicación necesitamos añadirle las dependencias necesarias para trabajar con JPA y con MySQL, que va a ser la base de datos con la que vamos a trabajar.

Aunque podríamos añadirle las dependencias a Maven directamente vamos a crear un nuevo proyecto desde cero al que añadiremos las dependencias y copiaremos las clases creadas al mismo, para poder aprovecharlas. Por ello:

1. Creamos un nuevo proyecto denominado `demo6`.
2. A la hora de añadir las dependencias vamos a añadir todas las que estábamos usando más las opciones `Spring Data JPA` , `MySQL Driver` y `Spring Dev Tools`.Las dos primeras opciones se encuentran en el apartado SQL (no tener en cuenta la versión de Spring Boot ya que puede diferir de los apuntes a la actual), mientras que la última la podemos encontrar en el apartado Developer Tools. La opción `Spring Dev Tools` es la herramienta de Spring Boot donde cada vez que realicemos un cambio en nuestro código el servidor procederá a actualizar la aplicación y reiniciarla sola de forma totalmente transparente y automática, por lo que no necesitamos volver a desplegarla.

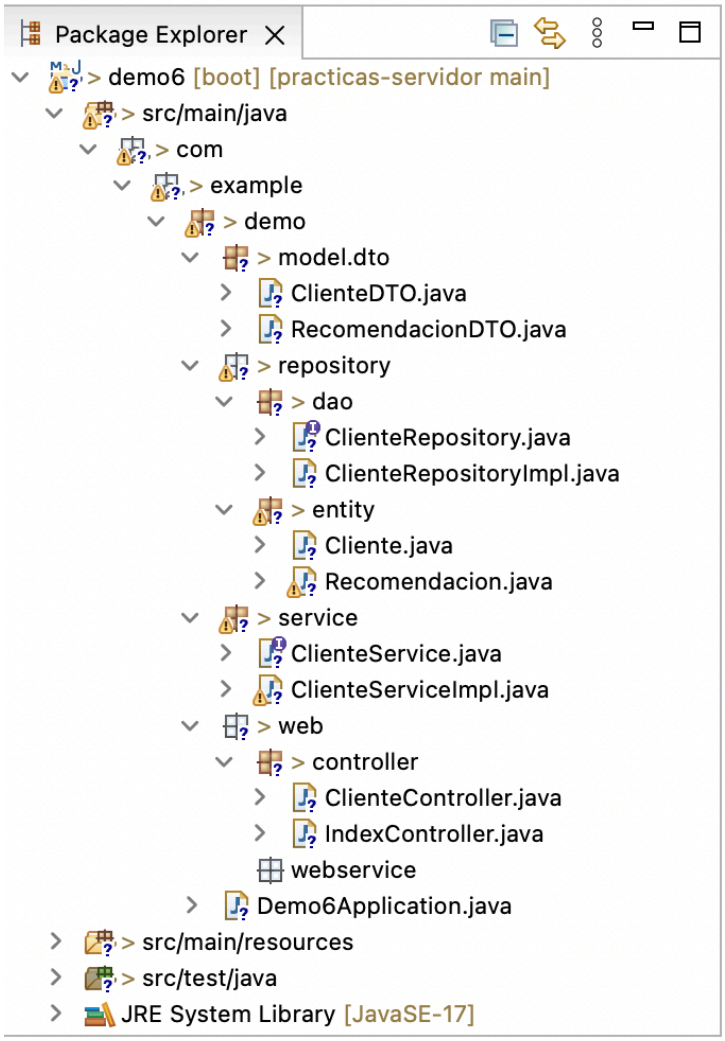



3. `Spring Data JPA` es el nombre de un módulo de Spring que a su vez engloba un gran número de sub-módulos cuyo objetivo es facilitar el acceso y explotación de datos en aplicaciones basadas en Spring siguiendo una práctica común. Los datos que se explotan pueden provenir de orígenes tan dispares como bases de datos relacionales, a través de JPA (que es nuestro caso) o `Spring Data JDBC` (añadir JDBC a Spring), bases de datos NoSQL como MongoDB o Neo4J, o directorios LDAP.
4. La otra dependencia que añadimos es `MySQL Driver`, que es el driver de conexión con la base de datos MySQL.
5. Finalizamos la creación del proyecto y comprobamos el fichero pom.xml, en el que podremos observar que se han añadido las nuevas dependencias.

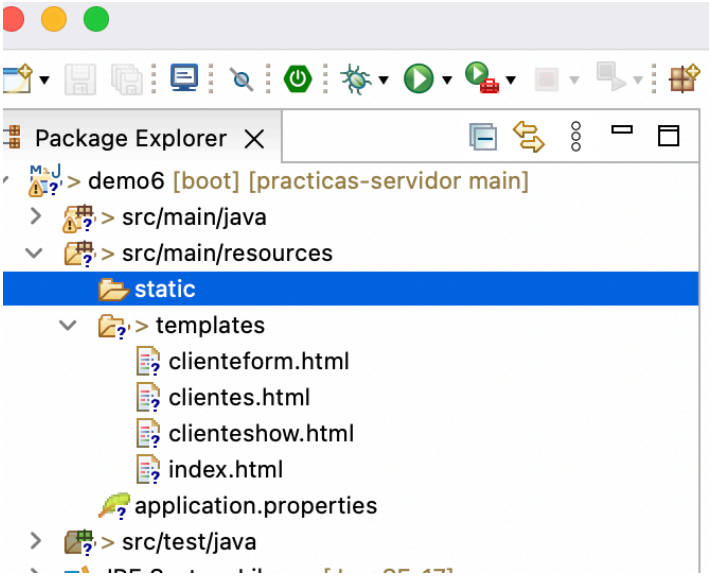
```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
```

```
      <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <scope>runtime</scope>
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>com.mysql</groupId>
      <artifactId>mysql-connector-j</artifactId>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

6. Tras esto, cogemos todos los paquetes del proyecto de la unidad anterior, menos el `com.example.demo` que contiene la clase principal de la aplicación, y los copiamos al nuevo proyecto. Quedará de la siguiente forma:




7. Tras esto copiamos el contenido de la carpeta `templates` al nuevo proyecto, quedando de la siguiente forma:




8. No nos olvidemos de copiar el contenido del fichero `application.properties`, de forma que quedará de la siguiente forma:

```
aplicacion.nombre = Gesti\u00f3n integral
asignatura = Servidor
server.port = 8888
```

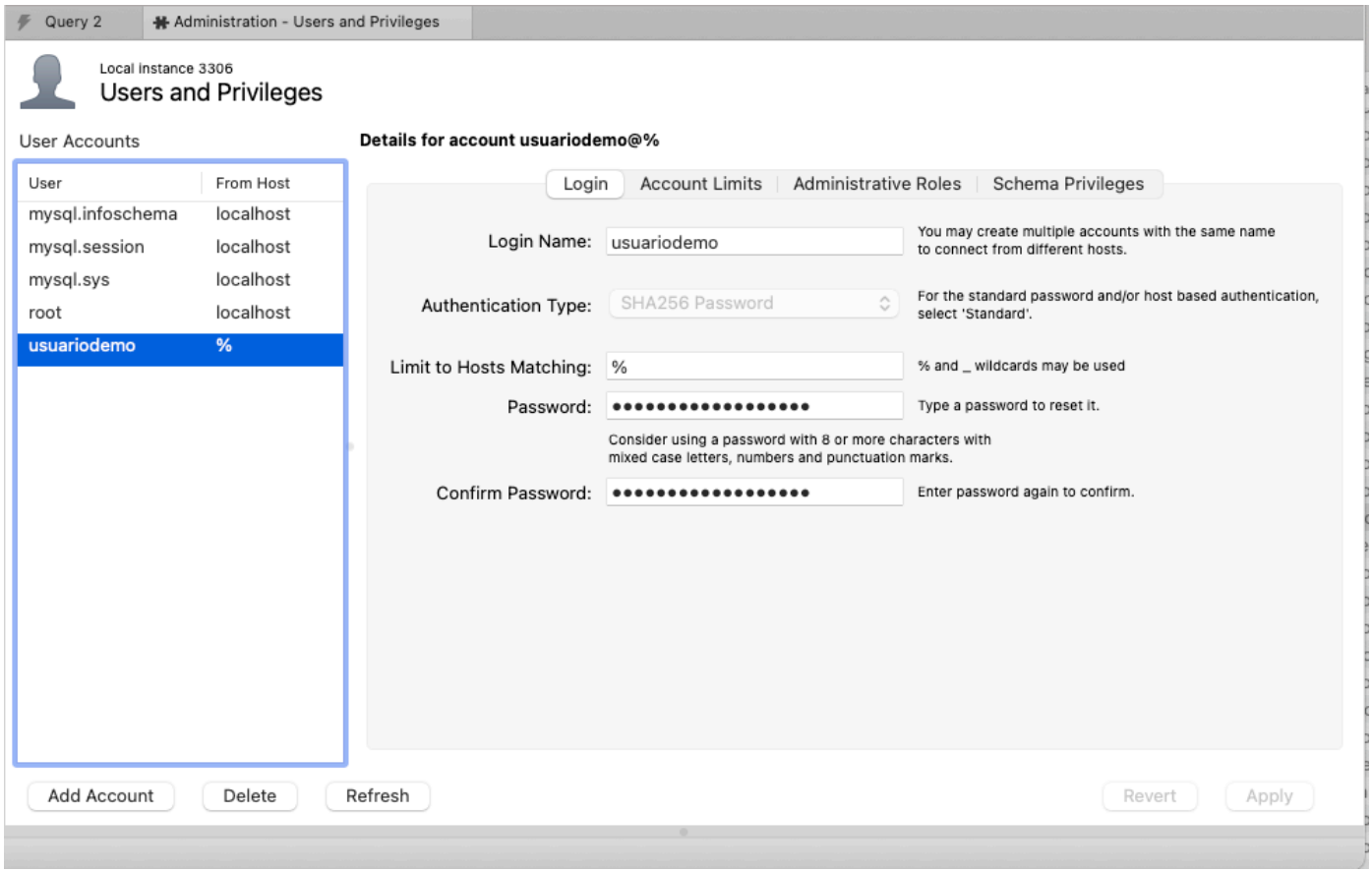
Con esto ya tendremos la aplicación preparada para trabajar con JPA sobre MySQL.

## Definiendo la base de datos a trabajar

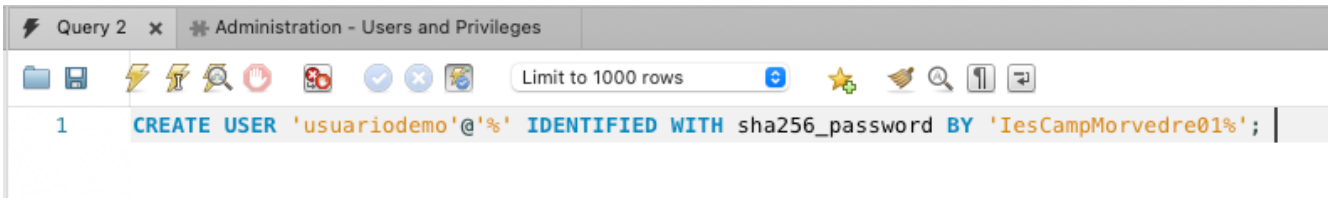
Como vamos a trabajar sobre MySQL lo primero es tener instalada el gestor de bases de datos, así como el MySQL WorkBench para poder trabajar sobre este (estos pasos de instalación lo saltamos).

Ahora sobre la aplicación MySQL WorkBench vamos a realizar los siguientes pasos:

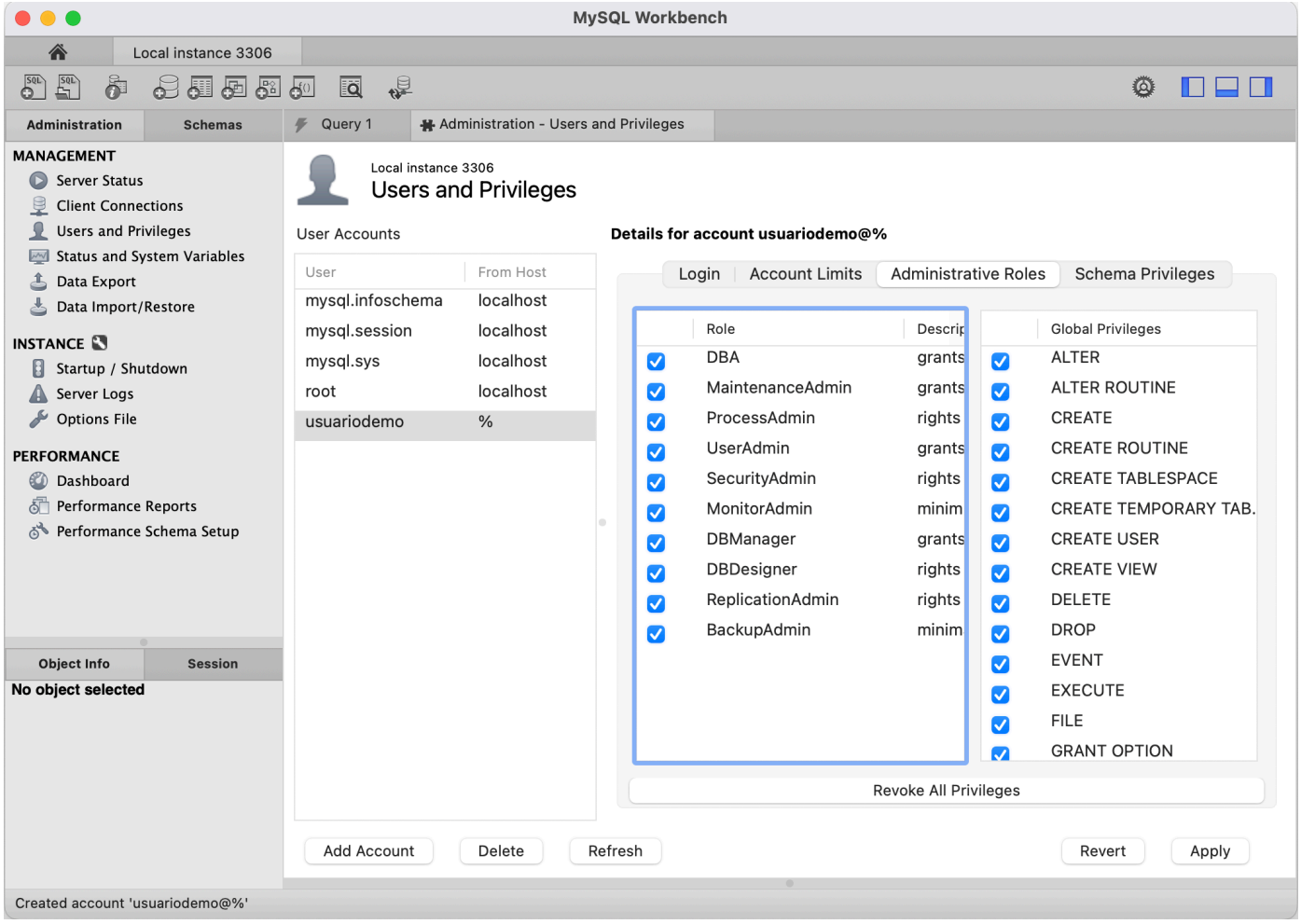
1. Nos conectamos con el usuario `root` al MySQL.
2. Tras ello crearemos un usuario denominado `usuariodemo` al que le asignaremos la contraseña `IesCampMorvedre01%`. Los usuarios se crean seleccionando la pestaña `Administration`, y en ella, dentro de la opción `Management` seleccionamos `Users and Privileges`.



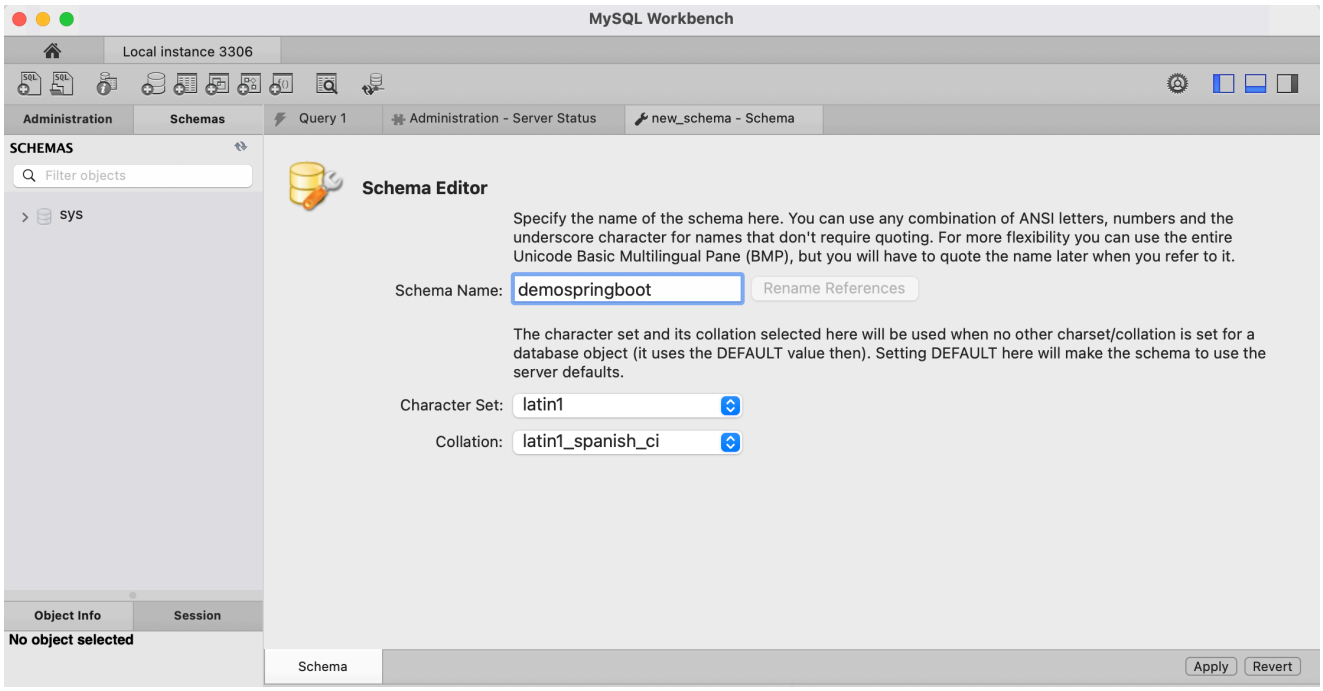

3. Si se ha creado correctamente el usuario pasa al siguiente paso, sino si habéis tenido algún problema a la hora de crearlo, como ha sido mi caso en Mac OS, vamos a lanzar la sentencia de creación del usuario diréctamente sobre una consola de MySQL y luego pasaremos a darles los permisos que necesita en el siguiente paso.  Recordar darle al rayo para ejecutar la sentencia. Tras esto, si volvemos a la pantalla de usuarios ya podemos ver el usuario creado.



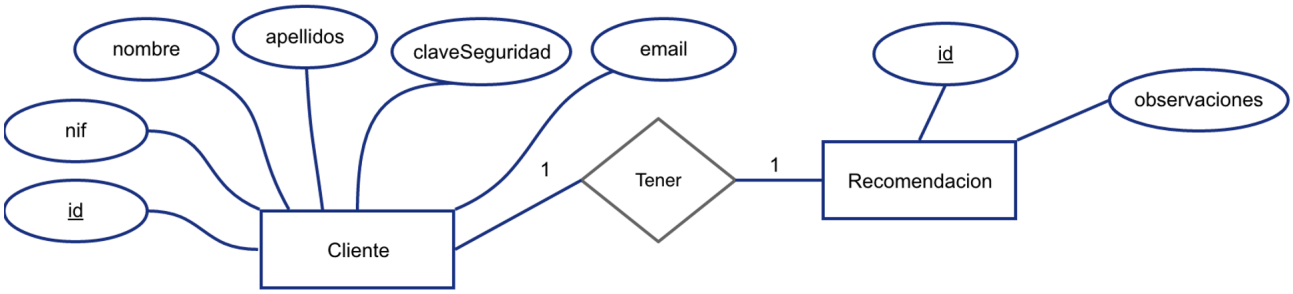

4. Sobre la ventana `Users and Privileges`, teniendo seleccionado el usuario `usuariodemo` nos vamos a la pestaña `Administrative Roles`. Sobre esta pestaña seleccionamos el rol DBA, que automáticamente seleccionará todas las opciones. Pulsamos `Apply` y el usuario ya tendrá permisos para hacer cualquier cosa (este curso no se trata de la gestión o seguridad de los usuarios de la base de datos, simplemente creamos uno y le damos todos los permisos para hacer cualquier cosa).



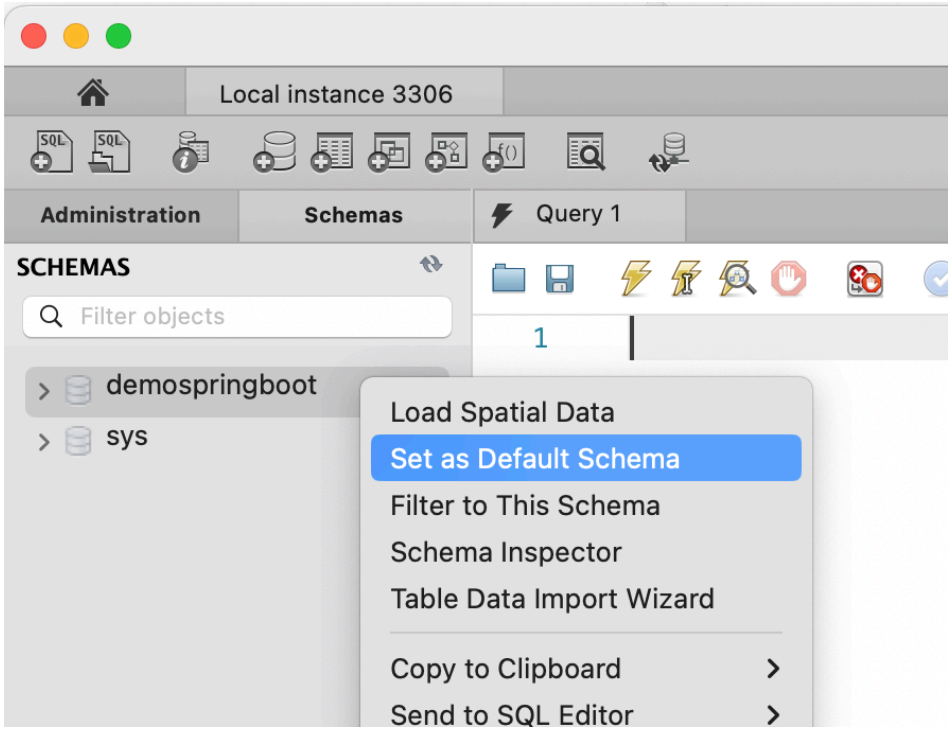

4. Tras ello procedemos a crear un esquema de base de datos denominado `demospringboot` con las siguientes características (esta opción está en la pestaña `Schemas`):



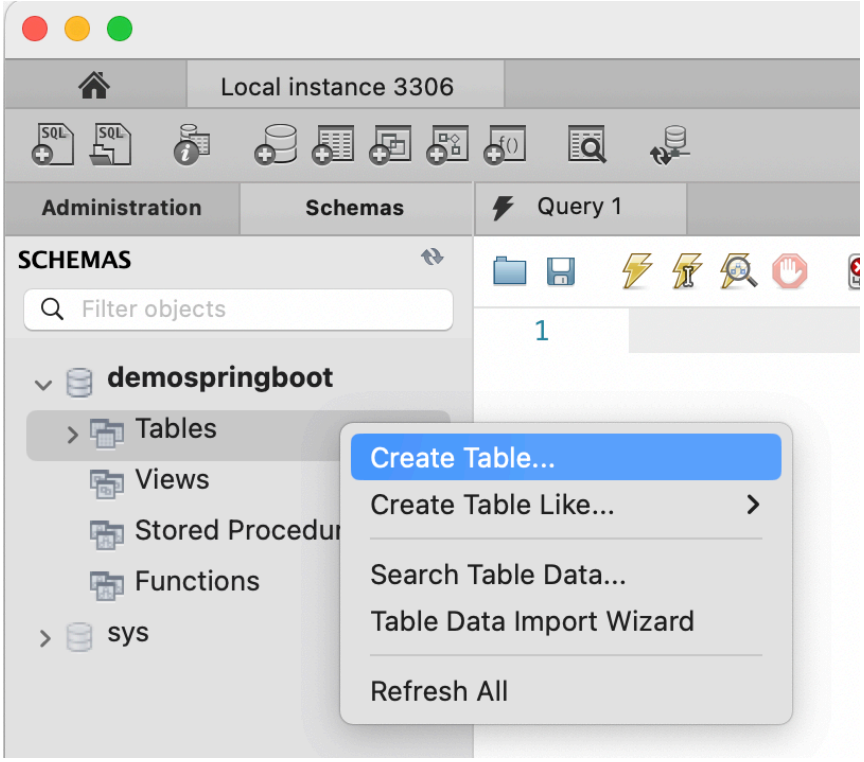

5. Ahora, tener en cuenta que vamos a diseñar la base de datos que propusimos en la unidad anterior, pero para este ejemplo inicial solo vamos a trabajar con la tabla clientes y la tabla recomendaciones, que está representada por la siguiente relación:

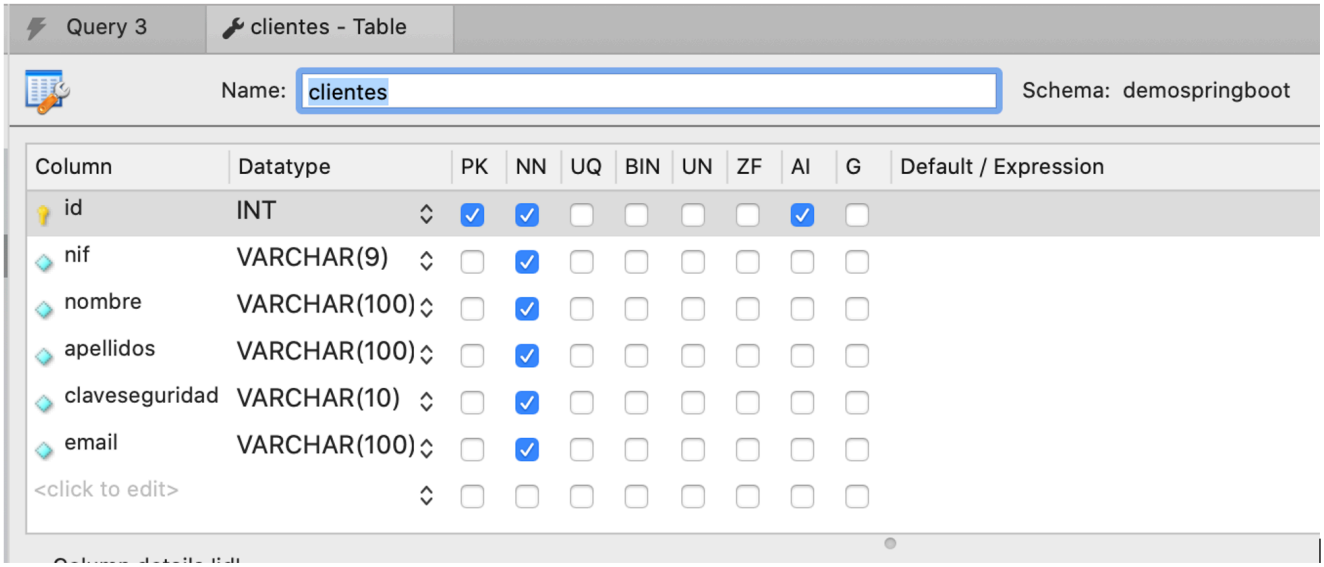



6. Lo primero que hacemos es seleccionar el esquema demospringboot que hemos creado como el esquema por defecto para poder trabajar (seleccionamos la opción `Set as Default Schema`).

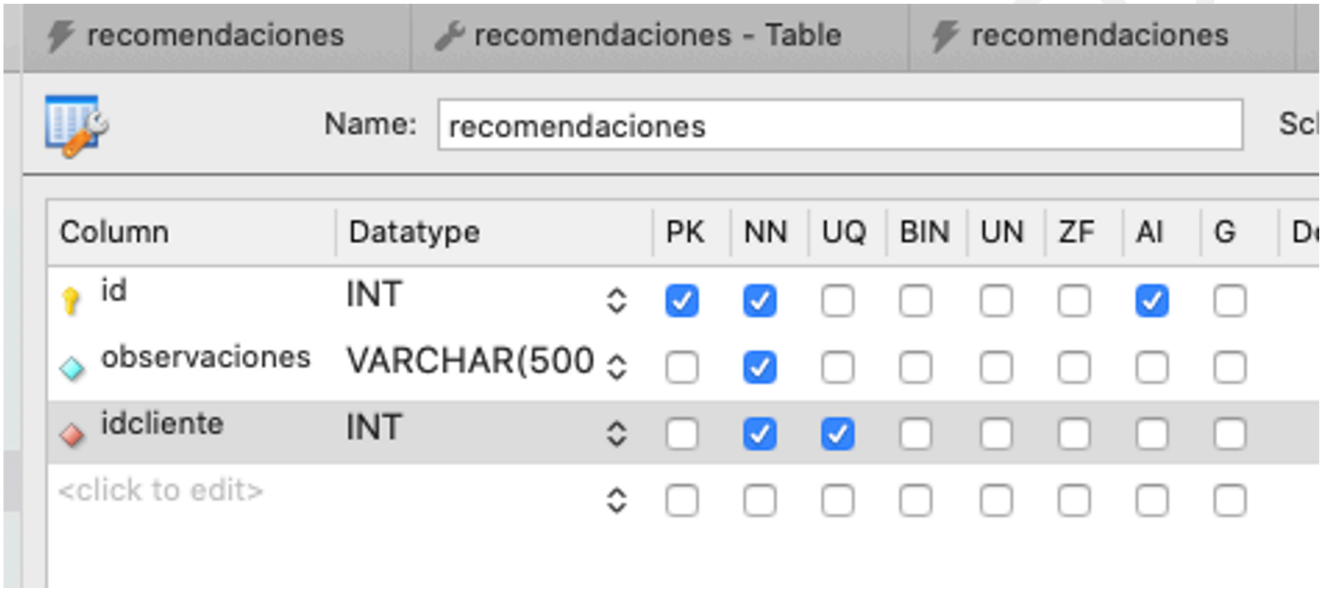



7. A continuación, creamos una tabla denominada `clientes` con los siguientes campos:

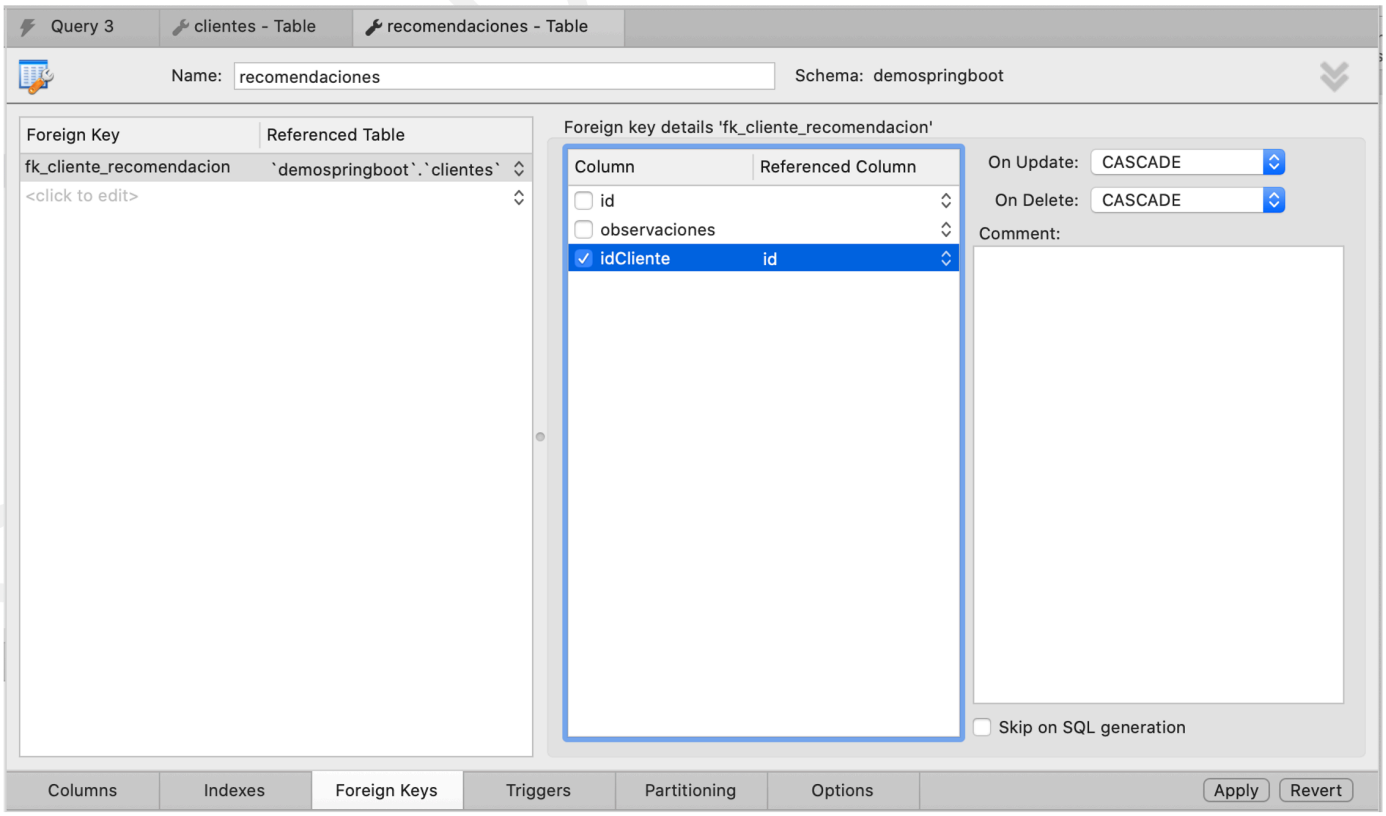




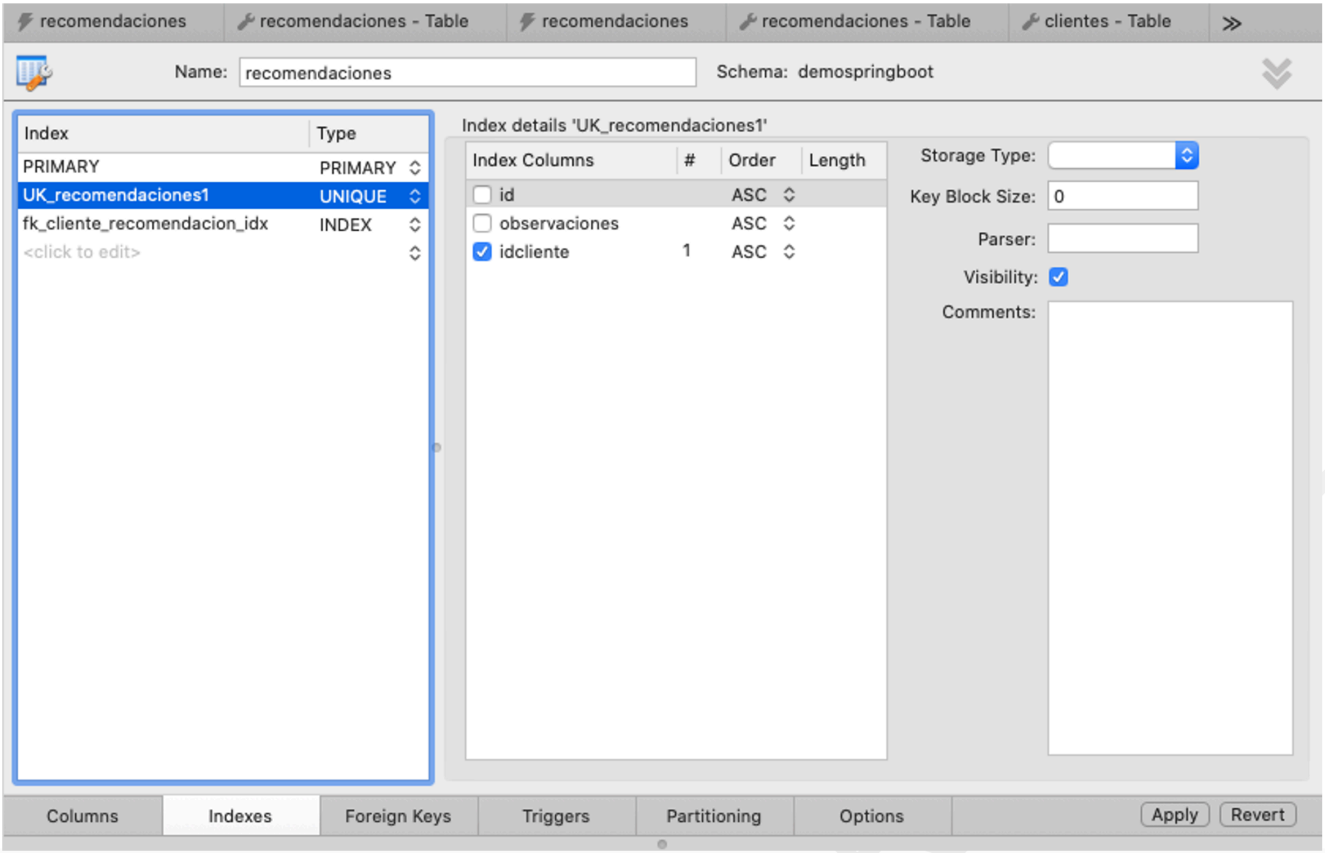

8. Creamos la tabla `recomendaciones`, que tendrà la siguiente estructura:




9. Tras ello vamos a modificar la tabla recomendaciones para añadir una clave ajena a la tabla `clientes` para realizar la relación uno a uno. La clave ajena es necesario crearla en la pestaña `Foreign Keys`. Quedará de la siguiente forma:

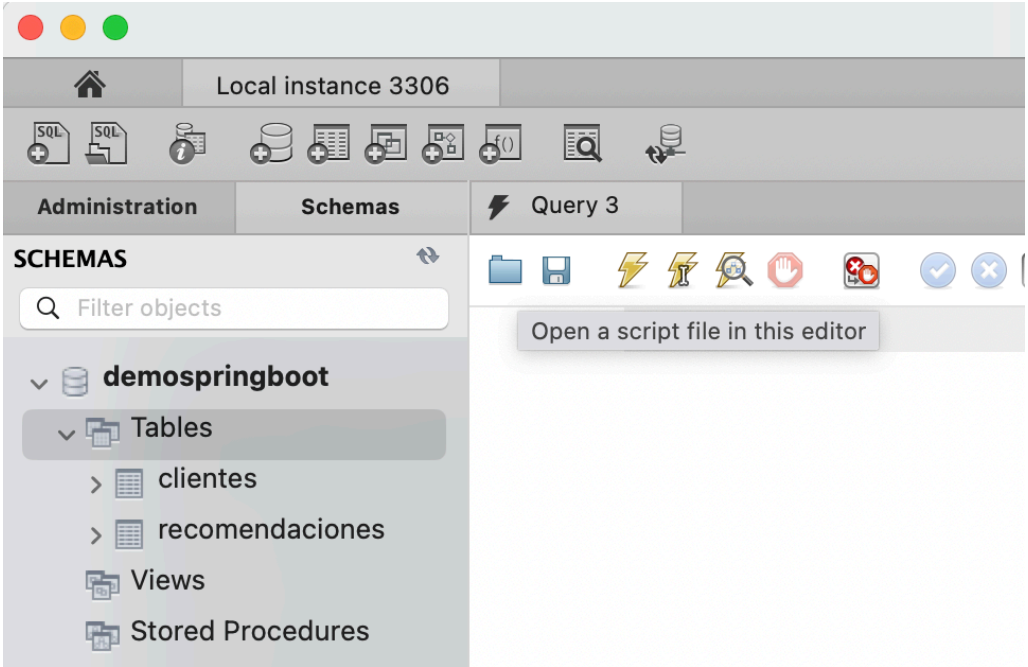



8. Aparte de esto le crearemos un índice único sobre la columna `idcliente`, de forma que solo puede haber un registro único con cada cliente, cumpliendo así la relación 1 a 1 en base de datos.

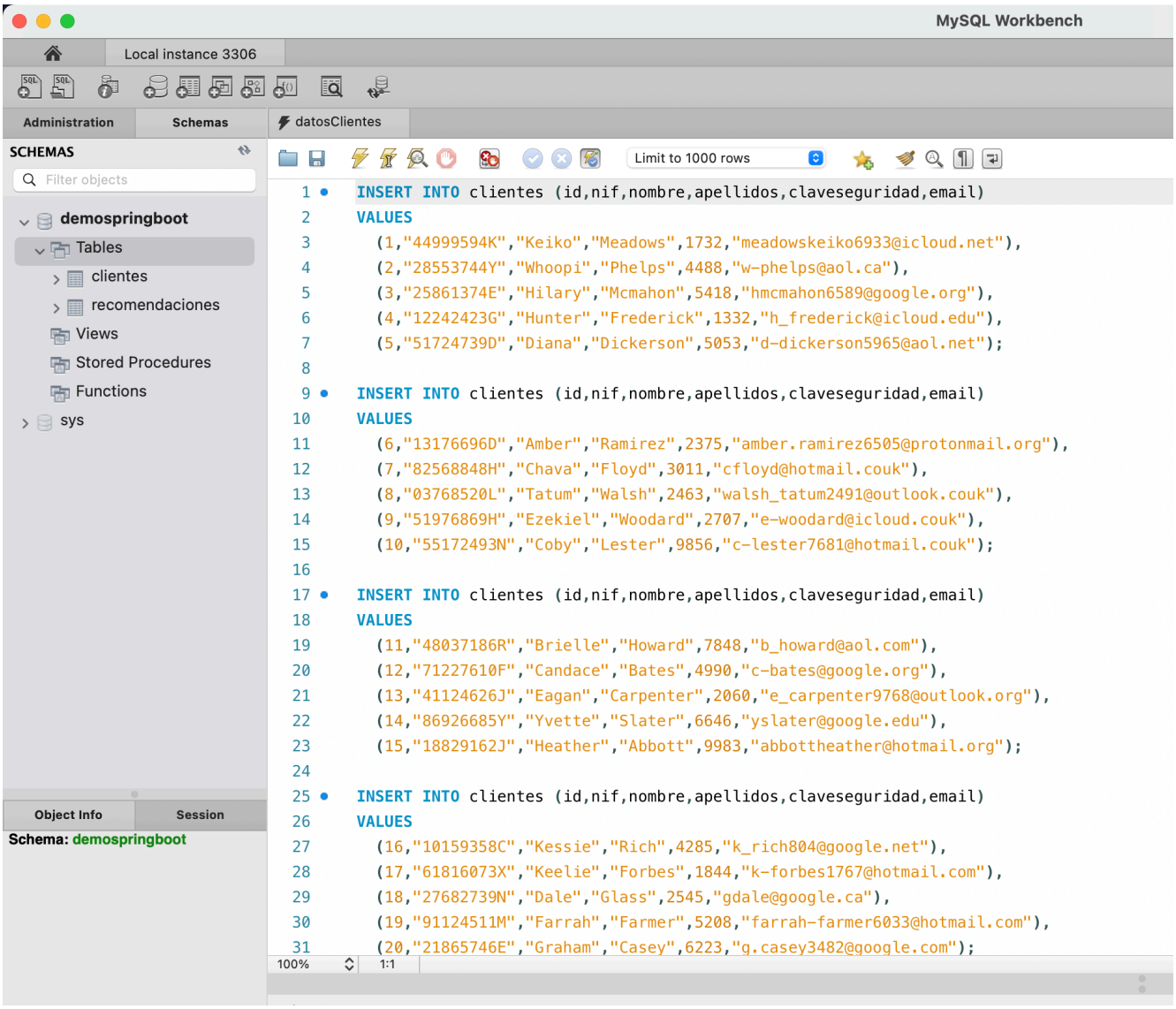



9. Tras ello ya tenemos creadas las estructuras en la BD que nos van a mantener los datos. Ahora vamos a proceder a cargar los datos de ejemplo que tenemos en el fichero `datosClientes.sql`. Dicho fichero se encuentra en los ficheros auxiliares proporcionados con la unidad. Para ello, desde una pestaña de query seleccionamos el icono de la carpeta, que nos va a permitir abrir un fichero y cargarlo en la

pestaña.




10. Tras seleccionar el fichero nos aparece en la pantalla de la siguiente forma:



11. Tras ello pulsamos el icono del rayo (el que solo es el rayo) y ejecutamos el script. Hemos de asegurarnos de ni haber seleccionado ninguna línea, ya que ejecutará solo hasta lo seleccionado. Tras esto se habran insertado los datos dentro de la tabla `clientes`.
12. Ahora hacemos lo mismo para la tabla `recomendaciones`, donde los datos están en el fichero `datosRecomendaciones.sql`. Tras realizarlo se habran insertado los datos dentro de la tabla. Si comprobáis dichas tablas veréis que tienen datos.

# Preparando la conexión entre la aplicación Spring Boot y MySQL

El siguiente paso es configurar la aplicación para que pueda conectarse a la base de datos. Para ello el fichero `application.properties` nos permite realizar las configuraciones de aplicaciones Spring Boot. Por ello, añadiremos lo siguiente al fichero:

```
aplicacion.nombre = Gesti\u00f3n integral
asignatura = Servidor
server.port = 8888

# ================================
# = DATA SOURCE
# ================================
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/demospringboot
spring.datasource.username=usuariodemo
spring.datasource.password=IesCampMorvedre01%

# ================================
# = JPA / HIBERNATE
# ================================
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto=none
spring.jpa.show-sql=true

# ================================
# = LOGGING. Sentencias SQL y parametros
# ================================
#logging.level.org.hibernate.SQL=DEBUG
#logging.level.org.hibernate.type=TRACE
```

Los parámetros que hemos añadido indican lo siguiente:

- `spring.datasource.driver-class-name`: Indica el driver o librería para conectar java con la base de datos correspondiente, en este caso MySQL.
- `spring.datasource.url`: Url donde esta el servicio del servidor de BD y el nombre del esquema.
- `spring.datasource.username`: Usuario de conexión al esquema de base de datos.

- `spring.datasource.password`: Contraseña del usuario para conectarse al esquema de base de datos.
- `spring.jpa.properties.hibernate.dialect`: Permite a Hibernate generar SQL optimizado para un sistema gestor de base de datos en particular, acelerando de esta forma el acceso a la BD.
- `spring.jpa.hibernate.ddl-auto`: Hibernate puede generar (o actualizar) de forma automática el esquema de base de datos de acuerdo a las entidades java encontradas en el proyecto. Los valores son: `create`, `create-drop`, `update` y `none`.
- `spring.jpa.show-sql`: Este parámetro es opcional, pero imprime en la consola las instrucciones SQL hechas para el esquema de base de datos. Como log para realizar comprobaciones es muy interesante, pero resulta incompleto ya que no muestra los parámetros. En este caso lo vamos a dejar comentado ya que lo sustituiremos por los dos siguientes parámetros.
- `logging.level.org.hibernate.sql`: Registra todas las declaraciones DML de SQL a medida que se ejecutan. Por ello el valor será `DEBUG`. Si muestra demasiado lo mejor es comentar esta opción. Por defecto yo la dejo comentada.
- `logging.level.org.hibernate.type`: Registra todos los parámetros JDBC. Se debe indicar la opción TRACE. Si muestra demasiado lo mejor es comentar esta opción. Por defecto yo la dejo comentada.

Tras definir como va a ser la conexión con la base de datos es momento de comenzar a mapear las entidades.

## Comenzando a mapear entidades con JPA

Vamos a comenzar a mapear la entidad `Cliente.java`. Para explicar como funciona el mapeo lo que vamos a hacer es mapearla al completo e ir explicando para que sirve cada anotación. Por ello, lo que tendremos al final es la siguiente clase mapeada:

```
1  package com.example.demo.repository.entity;
2
3  import java.util.Objects;
4
5  import jakarta.persistence.CascadeType;
6  import jakarta.persistence.Column;
7  import jakarta.persistence.Entity;
8  import jakarta.persistence.FetchType;
9  import jakarta.persistence.GeneratedValue;
10 import jakarta.persistence.GenerationType;
11 import jakarta.persistence.Id;
12 import jakarta.persistence.OneToOne;
13 import jakarta.persistence.Table;
14 import lombok.Data;
15 import lombok.ToString;
16
17 @Data
18 @Entity
19 @Table(name="clientes")
20 public class Cliente {
```

```
21
22    @Id
23    @GeneratedValue(strategy=GenerationType.IDENTITY)
24    private Long id;
25
26    @Column(name = "nif")
27    private String nif;
28
29    @Column(name = "nombre")
30    private String nombre;
31
32    private String apellidos;
33
34    @Column(name = "claveseguridad")
35    private String claveSeguridad;
36
37    private String email;
38
39    @OneToOne(fetch = FetchType.LAZY, cascade =  CascadeType.ALL, orphanRemoval = true,
mappedBy = "cliente")
40    @ToString.Exclude
41    private Recomendacion recomendacion;
42
43    @Override
44    public boolean equals(Object obj) {
45      if (this == obj)
46        return true;
47      if (obj == null)
48        return false;
49      if (getClass() != obj.getClass())
50        return false;
51      Cliente other = (Cliente) obj;
52      return Objects.equals(id, other.id);
53    }
54
55    @Override
56    public int hashCode() {
57      return Objects.hash(id);
58    }
59  }
```

Las anotaciones usadas son las siguientes:

- `@Entity`: Se aplica a la clase e indica que esta clase Java es una entidad a persistir.
- `@Table(name="clientes")`: Se aplica a la clase e indica el nombre de la tabla de la base de datos donde se persistirá la clase. Es opcional si el nombre de la clase coincide con el de la tabla.
- `@Id`: Se aplica a una propiedad Java e indica que este atributo es la clave primaria.

- `@GeneratedValue(strategy=GenerationType.IDENTITY)`: Esta anotación es para obligar a que el campo sea autoincremental. Tienes varias opciones, pero vamos a resaltar 2: `Identity` que sirve para MySQL y `Sequence` que sirve para Oracle. Como las opciones son bastantes, y hay mucha información de esto en internet, simplemente dejamos comentadas estas dos.
- `@Column(name="nif")`: Se aplica a una propiedad Java e indica el nombre de la columna de la base de datos en la que se persistirá la propiedad. Es opcional si el nombre de la propiedad Java coincide con el de la columna de la base de datos como pasa con el atributo `apellidos`.

A continuación, tendríamos el mapeo para la relación uno a uno. Simplemente lo ponemos y después explicaremos las relaciones.

Para el mapeo de la entidad `Recomendacion.java` tendremos lo siguiente:

```
package com.example.demo.repository.entity;

import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.OneToOne;
import jakarta.persistence.Table;
import lombok.Data;
import lombok.ToString;

@Data
@Entity
@Table(name="recomendaciones")
public class Recomendacion {

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

  private String observaciones;

  @OneToOne(fetch = FetchType.LAZY, optional = false)
  @JoinColumn(name = "idcliente")
  @ToString.Exclude
  private Cliente cliente;
}
```

Para el mapeo de la relación lo mismo, simplemente la ponemos para tenerla y después pasaremos a explicarla.

# Modificando los repositorios de datos

Ahora los repositorios de datos ya no pueden ser un `ArrayList` estático como teníamos, sino que debe ser el origen la base de datos con la que trabajamos.

Por ello, si recordáis tenemos un interfaz de operaciones para el repositorio, que se encuentra en el fichero `ClienteRepository.java`. El interfaz es el siguiente:

```
package com.example.demo.repository.dao;

import java.util.List;

import com.example.demo.repository.entity.Cliente;

public interface ClienteRepository {

  List<Cliente> findAll();
  Cliente findById(Cliente cliente);
  void save(Cliente cliente);
  void delete(Cliente cliente);
}
```

Bueno, pues resulta que `Spring Data` tiene su propio interfaz que proporciona todas estas operaciones ya implementadas, que es `JpaRepository`, de forma que, la implementación que realizamos en la clase `ClienteRepositoryImpl.java` nos la ahorramos. La convención a seguir para nombrar el repositorio/interfaz es utilizar el nombre de la entidad con el sufijo “Repository” (que esto ya lo cumplimos). Asimismo, hay que indicar la entidad para la que se crea el repositorio y el tipo de su clave primaria. Por ello, el fichero `ClienteRepository.java` quedará de la siguiente forma:

```
package com.example.demo.repository.dao;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import jakarta.transaction.Transactional;

import com.example.demo.repository.entity.Cliente;

@Repository
@Transactional
public interface ClienteRepository extends JpaRepository<Cliente, Long>{

}
```

La anotación `@Repository` sirve para indicar que dicha clase gestiona operaciones de persistencia contra la base de datos, de forma que indica que dicha anotación debe de ser empleada en la capa de persistencia.

La anotación `@Transactional` indica, según si está a nivel de clase o a nivel de método, que se trabajará con transacciones. Por defecto se trabaja con la opción `PROPAGATION_REQUIRED`, aunque existen otras opciones:

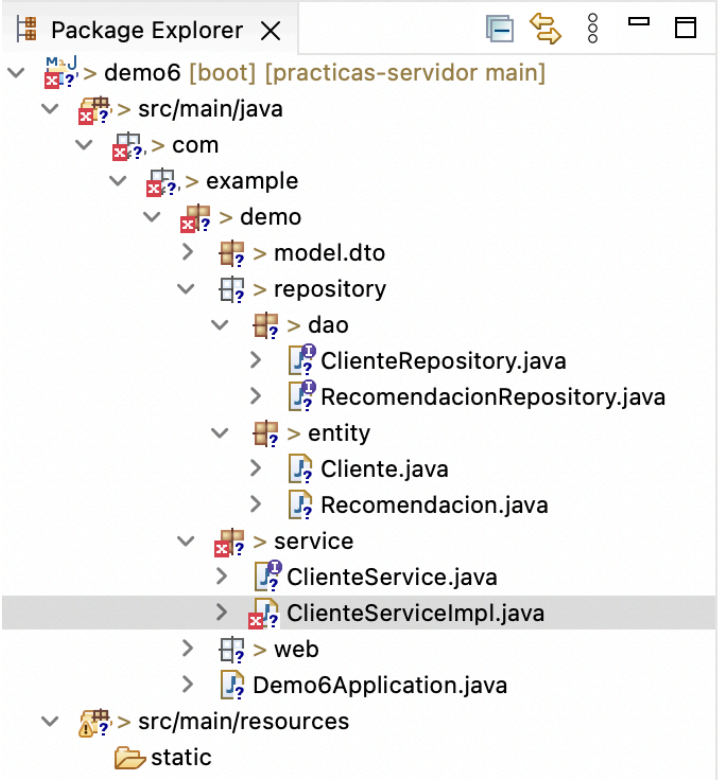
- `PROPAGATION_REQUIRED` – Es la que viene por defecto, así que no es necesaria especificarla. Si existe transacción la aprovecha y sino la crea.
- `REQUIRES_NEW` – Abre una transacción nueva y pone en suspenso la anterior. Una vez el método marcado como `REQUIRES_NEW` termina se vuelve a la transacción anterior.
- `PROPAGATION_SUPPORTS` – Soports una transacción concurrente. Si existe una transacción la aprovecha, sino no crea ninguna.
- `PROPAGATION_MANDATORY` – Si no existe una transacción abierta se lanza una excepción.
- `PROPAGATION_NEVER` – Si existe una transacción abierta se lanza una excepción.
- `PROPAGATION_NOT_SUPPORTED` – Si existe una transacción la pone en suspenso, la transacción se reactiva al salir del método.

Con esto tendremos todas las operaciones que teníamos en la clase `ClienteRepositoryImpl.java` ya implementadas. Por ello, borramos la clase `ClienteRepositoryImpl.java` ya que no tiene sentido tenerla, puesto que están todos los métodos implementados.

Por supuesto, si tenemos el repositorio para la entidad `Cliente` también tenemos que definirla para la entidad `Recomendacion`, por lo que crearemos la interfaz `RecomendacionRepository.java` siguiendo los mismos pasos que `ClienteRepository.java`. Quedará de la siguiente forma:

```
package com.example.demo.repository.dao;

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

import com.example.demo.repository.entity.Recomendacion;

import jakarta.transaction.Transactional;

@Repository
@Transactional
public interface RecomendacionRepository extends JpaRepository<Recomendacion, Long>{

}
```

Tras ello, si comprobáis el árbol veréis que tenemos unos errores en la clase `ClienteServiceImpl.java`:




Si abrimos la clase y verificamos el error nos encontraremos con que el método `clienteRepository.findById(Cliente)` ya no devuelve un objeto de la clase `Cliente`, sino que devuelve un valor `\<Optional\> Cliente`. Además ya no recibe como parámetro un objeto de tipo `Cliente` sino un `Long`, que es el id.

La clase `Optional` es una clase que aparecen en Java 8 cuyo objetivo es eliminar los errores por null. De ahí que el patrón `Option` (implementado en Java 8 en la clase `Optional`) nos permite mostrar la posibilidad de que un método pueda no devolver el valor deseado. Esto nos obliga a controlar la posible ausencia de valor, permitiéndonos elegir un valor alternativo en caso de dicha ausencia o simplemente realizar otra acción.

Por ello `Optional` es un objeto contenedor que puede o no contener un valor no nulo. Si el valor es presente, `isPresent()` devuelve `true` y `get()` retorna el valor mientras que `orElse()` retorna un valor por defecto si el valor no se encuentra presente.

A partir de esto, tenemos que modificar el método `public ClienteDTO findById(ClienteDTO clienteDTO)` de la clase `ClienteServiceImpl.java` quedando de la siguiente forma:

```
package com.example.demo.service;

import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.example.demo.model.dto.ClienteDTO;
import com.example.demo.repository.dao.ClienteRepository;
import com.example.demo.repository.entity.Cliente;
```

```
15
16 @Service
17 public class ClienteServiceImpl implements ClienteService{
18
19   private static final Logger log =
LoggerFactory.getLogger(ClienteServiceImpl.class);
20
21   @Autowired
22   private ClienteRepository clienteRepository;
23
24   @Override
25   public List<ClienteDTO> findAll() {
26     log.info("ClienteServiceImpl - findAll: Lista de todos los cliente");
27
28     List<ClienteDTO> listaClientesDTO = clienteRepository.findAll()
29         .stream()
30         .map(p->ClienteDTO.convertToDTO(p))
31         .collect(Collectors.toList());
32
33     return listaClientesDTO;
34   }
35
36   @Override
37   public ClienteDTO findById(ClienteDTO clienteDTO) {
38     log.info("ClienteServiceImpl - findById: Buscar cliente por id: " +
clienteDTO.getId());
39
40     Optional<Cliente> cliente = clienteRepository.findById(clienteDTO.getId());
41     if(cliente.isPresent()) {
42       clienteDTO = ClienteDTO.convertToDTO(cliente.get());
43       return clienteDTO;
44     }else {
45       return null;
46     }
47   }
48
49   @Override
50   public void save(ClienteDTO clienteDTO) {
51     log.info("ClienteServiceImpl - save: Salvamos el cliente: " +
clienteDTO.toString());
52
53     Cliente cliente = ClienteDTO.convertToEntity(clienteDTO);
54     clienteRepository.save(cliente);
55   }
56
57   @Override
58   public void delete(ClienteDTO clienteDTO) {
59     log.info("ClienteServiceImpl - delete: Borramos el cliente: " +
clienteDTO.getId());
60
61     Cliente cliente = new Cliente();
62     cliente.setId(clienteDTO.getId());
```

```
    clienteRepository.delete(cliente);
  }
}
```

Con esto el error quedará solucionado.

Si probamos en este momento la aplicación funcionaria pero no podriamos dar de alta (o modificar) clientes. Esto es debido a que, cuando recoge el cliente en el Post recoge también la recomendación, pero no tenemos id de la misma. Por ello, antes de continuar modificaremos la vista `clienteform.html` para añadir el id de la recomendación para que, a la hora de recogerlo en el controller, recoja dicho dato también. Quedará de la siguiente forma:

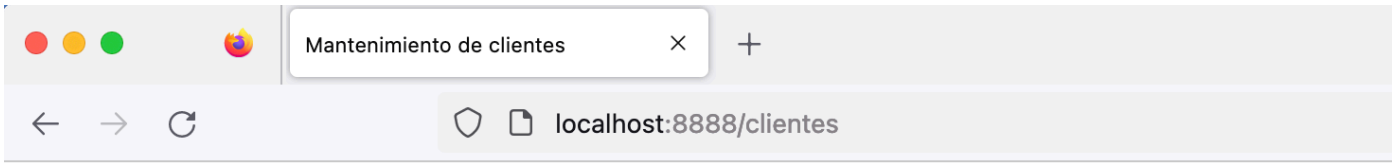
```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<title th:text="${add} ? 'Nuevo cliente' : 'Actualizar cliente'"></title>
</head>
<body>
  <h1 th:text="${add} ? 'Nuevo cliente' : 'Actualizar cliente'" />
  <br>
  <br>
  <form th:action="@{/clientes/save}" th:object="${clienteDTO}"
    method="POST">
    <table border="0">
      <tr th:if="${clienteDTO.id}">
        <td>id</td>
        <td>:</td>
        <td> <input type="text" th:field="*{id}" readonly/> </td>
      </tr>
      <tr>
        <td>Nombre</td>
        <td>:</td>
        <td> <input type="text" th:field="*{nombre}" /> </td>
      </tr>
      <tr>
        <td>Apellidos</td>
        <td>:</td>
        <td> <input type="text" th:field="*{apellidos}" /> </td>
      </tr>
      <tr>
        <td>NIF</td>
        <td>:</td>
        <td> <input type="text" th:field="*{nif}" /> </td>
      </tr>
      <tr>
        <td>Correo electrónico</td>
        <td>:</td>
        <td> <input type="text" th:field="*{email}" /> </td>
      </tr>
      <tr>
```

```
        <td>Clave seguridad</td>
        <td>:</td>
        <td> <input type="text" th:field="*{claveSeguridad}" /> </td>
      </tr>
      <tr>
        <td>Recomendación</td>
        <td>:</td>
        <td> <textarea th:field="*{recomendacionDTO.observaciones}" rows="4" cols="50" /> </td>
      </tr>

      <!-- Ocultamos el id de la recomendacion del cliente -->
      <input type="hidden" th:field="*{recomendacionDTO.id}">

    </table>
    <input type="submit" th:value="${add} ? 'Nuevo' : 'Actualizar'" />
  </form>
  <br>
  <a th:href="@{/clientes}">Volver a la lista de clientes</a>
</body>
</html>
```

# Probando la aplicación.

Llegados a este punto es momento de probar nuestra aplicación:

- Comenzamos visualizando los clientes que tenemos en la base de datos.

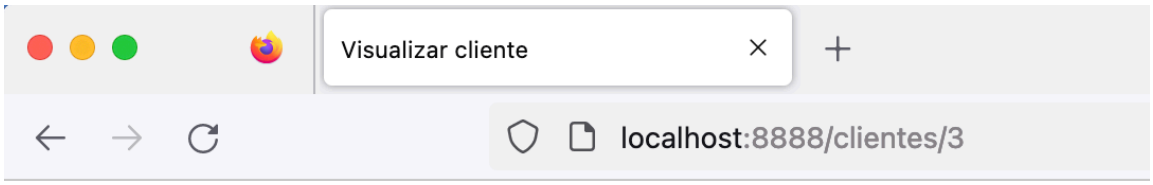



# Mantenimiento de clientes

| id | Nombre | Apellidos | NIF | Correo electrónico | Editar | Eliminar |
|---|---|---|---|---|---|---|
| 1 | Filemón | Pí | 11111111A | filemon.pi@tia.es | Editar | Eliminar |
| 2 | Mortadelo | Ibáñez | 22222222B | mortadelo.ibanez@tia.es | Editar | Eliminar |
| 3 | Vicente | Mondragón | 33333333C | vicente.mondragon@tia.es | Editar | Eliminar |
| 4 | Ayrton | Senna | 44444444D | ayrton.senna@f1.es | Editar | Eliminar |
| 5 | Ibertrola | - | B1111111A | - | Editar | Eliminar |
| 6 | Gas Natural | - | B2222222B | - | Editar | Eliminar |
| 7 | Telefónica | - | B3333333C | - | Editar | Eliminar |
| 8 | Aguas de Valencia | - | B4444444D | - | Editar | Eliminar |
| 9 | Audi | - | B5555555E | - | Editar | Eliminar |
| 10 | BMW | - | B6666666F | - | Editar | Eliminar |
| 11 | PayPal | - | B7777777G | - | Editar | Eliminar |
| 12 | Ayuntamiento de Valencia | - | B8888888H | - | Editar | Eliminar |

Nuevo cliente
Volver al inicio

- Visualizamos la información de “Vicente Mondragón” con id 3.

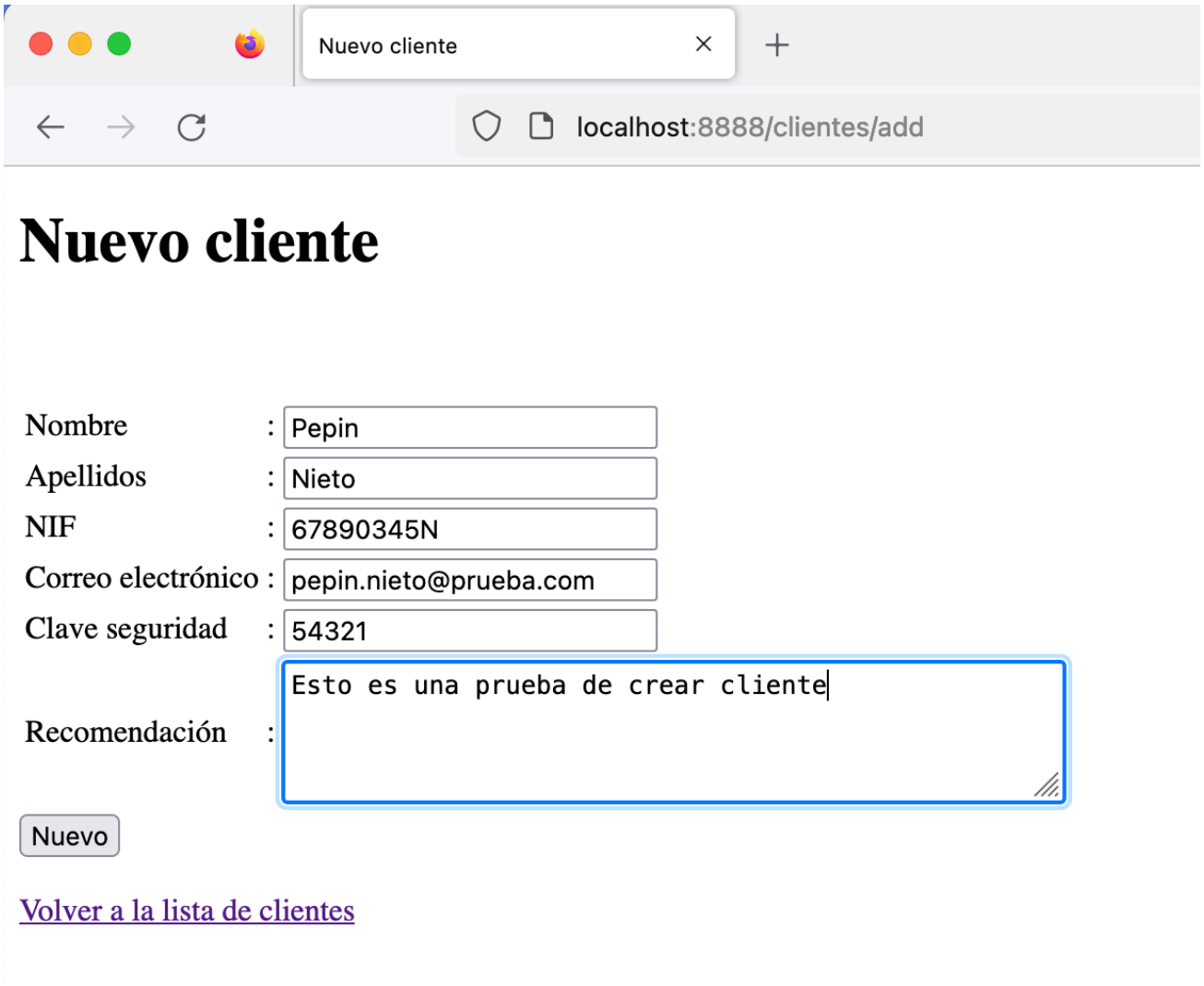



# Visualizar cliente

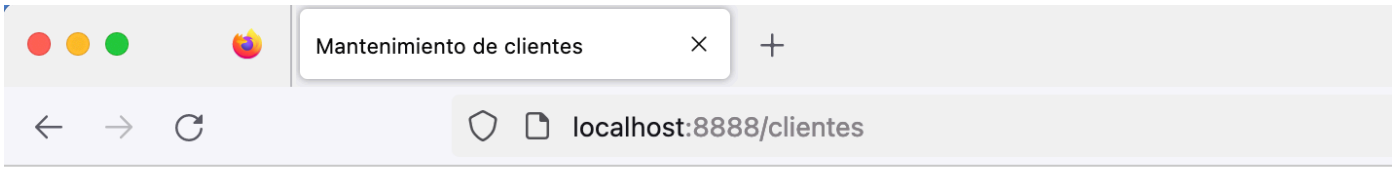
| | |
|---|---|
| id | : 3 |
| Nombre | : Vicente |
| Apellidos | : Mondragón |
| NIF | : 33333333C |
| Correo electrónico | : vicente.mondragon@tia.es |
| Clave seguridad | : 1234 |
| Recomendación | : Realiza muchos pedidos |

Volver a la lista de clientes

- Creamos un cliente nuevo:




- Tras crear el cliente nuevo, volvemos a la lista de clientes.

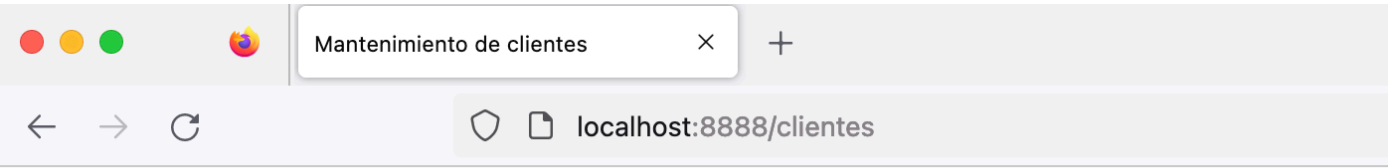



# Mantenimiento de clientes

| id | Nombre | Apellidos | NIF | Correo electrónico | Editar | Eliminar |
|---|---|---|---|---|---|---|
| 1 | Filemón | Pí | 11111111A | filemon.pi@tia.es | Editar | Eliminar |
| 2 | Mortadelo | Ibáñez | 22222222B | mortadelo.ibanez@tia.es | Editar | Eliminar |
| 3 | Vicente | Mondragón | 33333333C | vicente.mondragon@tia.es | Editar | Eliminar |
| 4 | Ayrton | Senna | 44444444D | ayrton.senna@f1.es | Editar | Eliminar |
| 5 | Ibertrola | - | B1111111A | - | Editar | Eliminar |
| 6 | Gas Natural | - | B2222222B | - | Editar | Eliminar |
| 7 | Telefónica | - | B3333333C | - | Editar | Eliminar |
| 8 | Aguas de Valencia | - | B4444444D | - | Editar | Eliminar |
| 9 | Audi | - | B5555555E | - | Editar | Eliminar |
| 10 | BMW | - | B6666666F | - | Editar | Eliminar |
| 11 | PayPal | - | B7777777G | - | Editar | Eliminar |
| 12 | Ayuntamiento de Valencia | - | B8888888H | - | Editar | Eliminar |
| 15 | Pepin | Nieto | 67890345N | pepin.nieto@prueba.com | Editar | Eliminar |

Nuevo cliente
Volver al inicio

- Eliminamos el cliente “Pepe Nieto” con id 13:




## Mantenimiento de clientes

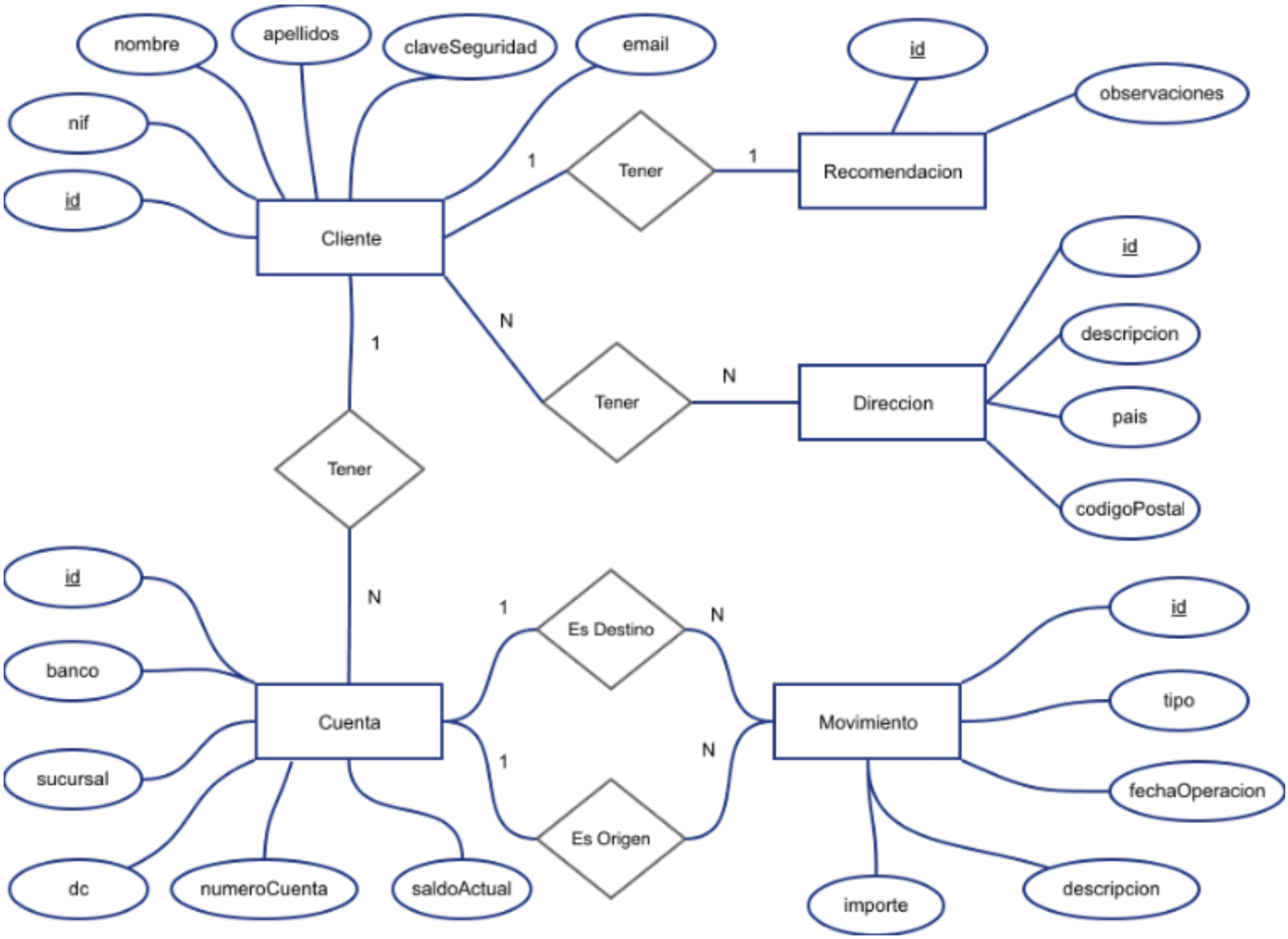
| id | Nombre | Apellidos | NIF | Correo electrónico | Editar | Eliminar |
|---|---|---|---|---|---|---|
| 1 | Filemón | Pí | 11111111A | filemon.pi@tia.es | Editar | Eliminar |
| 2 | Mortadelo | Ibáñez | 22222222B | mortadelo.ibanez@tia.es | Editar | Eliminar |
| 3 | Vicente | Mondragón | 33333333C | vicente.mondragon@tia.es | Editar | Eliminar |
| 4 | Ayrton | Senna | 44444444D | ayrton.senna@f1.es | Editar | Eliminar |
| 5 | Ibertrola | - | B1111111A | - | Editar | Eliminar |
| 6 | Gas Natural | - | B2222222B | - | Editar | Eliminar |
| 7 | Telefónica | - | B3333333C | - | Editar | Eliminar |
| 8 | Aguas de Valencia | - | B4444444D | - | Editar | Eliminar |
| 9 | Audi | - | B5555555E | - | Editar | Eliminar |
| 10 | BMW | - | B6666666F | - | Editar | Eliminar |
| 11 | PayPal | - | B7777777G | - | Editar | Eliminar |
| 12 | Ayuntamiento de Valencia | - | B8888888H | - | Editar | Eliminar |

Nuevo cliente
Volver al inicio

# Mapeando relaciones (Parte 1 de 2)

## Introducción

Ahora es el momento de comenzar a construir la base de datos completa que definimos en el tema anterior, que es la siguiente:




Las relaciones con las que vamos a trabajar son las siguientes:

- Relaciones 1 a 1.
- Relaciones 1 a N.
- Relaciones N a N.

# Relaciones 1 a 1

Aunque la hemos implementado para probar el primer ejemplo, es momento de explicar como implementar las relaciones 1 a 1 en Spring Boot mediante JPA.

Está claro que la relación de la que vamos a hablar es la que se produce entre las tablas `clientes` y `recomendaciones`. La relación se produce mediante una clave ajena sobre el campo `idcliente` de la tabla `recomendaciones`.

Nuestra clase `Cliente` tendrá conocimiento de la clase entidad con la que se relaciona, que será `Recomendacion`, y viceversa. De esta forma implementamos una relación bidireccional, de forma que podremos navegar de una entidad a otra sin problemas.

Comenzaremos hablando de la clase `Cliente.java`, donde indicaremos que está relacionada con la clase `Recomendacion.java` mediante un atributo de tipo `Recomendacion` dentro de la clase `Cliente.java`. A dicho atributo le añadiremos la anotación `@OneToOne` con los siguientes parámetros:

- `fetch`: Cuando recuperamos información de diferentes objetos de entidad (información de distintas tablas de base de datos), en ocasiones, no hay necesidad de recuperar toda la información y sobrecargar la memoria Java, ya que el usuario NO necesita ver la información completa en ese momento, sino en consultas subsecuentes. Por ello JPA maneja el concepto de `Lazy Loading` (carga retardada), con lo cual logramos NO cargar las colecciones de entidades asociadas a cierta clase de entidad y recuperar sólo la información que sea relevante para esa transacción. Tenemos dos opciones:
    - `Lazy Fetch ó lectura demorada`: si NO queremos que la consulta recupere todos los objetos de entidad `Recomendacion` asociado al objeto `Cliente`, deberemos configurar nuestra clase con el valor `FetchType.LAZY`. Esto viene a reflejar la característica de JPA que permite que, cuando existen relaciones entre distintas entidades, las que son dependientes de otras no sean inicializadas con sus valores almacenados en base de datos hasta que no sean explícitamente accedidas (leídas).
    - `Eager Fetch ó lectura inmediate`: La entidad `Recomendacion` es inicializada en el mismo instante que es la entidad contenedora, que es `Cliente`. Esto implica una sobrecarga para la aplicación, puesto que recuperaremos todos los objetos y los tendremos en memoria.
- `cascade`: Queremos que la entidad `Recomendacion` se guarde en la base de datos en todo momento que se haga una actualización en la entidad `Cliente`. De la misma forma, queremos que cuando una entidad de tipo `Cliente` sea eliminada de la base de datos también se elimine su correspondiente entidad `Recomendacion`. Esto es indicado con este parámetro. Las operaciones en cascada son operaciones que se realizan en los hijos al mismo momento que en los padres (o en las entidades relacionadas con la entidad en la que estamos realizando la operación). Los valores permitidos para esta opción son: `persist`, `merge`, `save-update`, `delete`, `lock`, `refresh`, `evict`, `replicate`, `all`, `none`. Nosotros utilizaremos `all`, de esta forma siempre será en cascada cualquier operación.
- `orphanRemoval`: Indicará que, en caso de que sea borrada la entidad principal, es decir `Cliente`, será eliminada la entidad secundaria, es decir `Recomendacion`, para que no quede huerfana. Por defecto se encuenta a False y lo activaremos a True.
- `mappedBy`: Indicará el nombre del atributo con el que está representado el objeto contenedor, que es `Cliente.java`, en la clase contenida, que es `Recomendacion.java`. En este caso, el atributo se llamará `cliente`, y se encontrará en la clase `Recomendacion.java`.

Podemos ver que el mapeo de la relación en la clase `Cliente.java` quedará de la siguiente forma:

```
package com.example.demo.repository.entity;

import java.util.Objects;

import jakarta.persistence.CascadeType;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.OneToOne;
import jakarta.persistence.Table;
import lombok.Data;
```

```java
import lombok.ToString;

@Data
@Entity
@Table(name="clientes")
public class Cliente {

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

  @Column(name = "nif")
  private String nif;

  @Column(name = "nombre")
  private String nombre;

  private String apellidos;

  @Column(name = "claveseguridad")
  private String claveSeguridad;

  private String email;

  @OneToOne(fetch = FetchType.LAZY, cascade =  CascadeType.ALL, orphanRemoval = true, mappedBy = "cliente")
  @ToString.Exclude
  private Recomendacion recomendacion;

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Cliente other = (Cliente) obj;
    return Objects.equals(id, other.id);
  }

  @Override
  public int hashCode() {
    return Objects.hash(id);
  }
}
```

En cuanto a la clase `Recomendacion.java` indicaremos que está relacionada con la clase `Cliente.java` mediante un atributo de tipo `Cliente` dentro de la clase `Recomendacion.java`. Dentro de esto, en una relación bidireccional, especificamos la anotación `@OneToOne` en ambas entidades, pero solo una entidad es la propietaria de la relación. Muy a menudo, la entidad secundaria (que es `Recomendacion`) es la propietaria

de la relación y la entidad principal (que es `Cliente`) es el lado inverso de la relación.

El propietario de la relación contiene una anotación `@JoinColumn` para especificar la columna de base de datos que es clave externa (en este caso se llama `idcliente` que está en la tabla `Recomendaciones`), y el lado inverso de la relación contiene un atributo `mappedBy` para indicar que la relación está asignada por la otra entidad (ya lo hemos comentado con anterioridad).

Al atributo que indica la relación en la clase `Recomendacion.java` le añadiremos la anotación `@OneToOne` con los siguientes parámetros:

- `fetch`: Ya lo hemos explicado en el punto anterior, pero sÍ indicamos que será de tipo `EAGER`.
- `optional`: La anotación `@OneToOne` admite el atributo `optional` que permite indicar si la relación debe existir siempre (por defecto siempre está en `true`) o puede ser nula. Si indicamos que la relación no puede ser nula (ponerla en valor `false`), Hibernate ya no tiene que comprobar su existencia y la relación puede obtenerse de forma `Lazy`. Esto sólo funciona en el extremo de la relación en el que tenemos la clave ajena, es decir, donde no tenemos definido el atributo `mappedBy`.

A continuación, tras la notación `@OneToOne` pondremos la notación `@JoinColumn` con los siguientes parámetros:

- `name`: ya lo hemos explicado con anterioridad, pero por resumir, indica en nombre de la columna en BD.

Con todo esto, el mapeo en la entidad `Recomendacion.java` quedará de la siguiente forma:

```
package com.example.demo.repository.entity;

import java.util.Objects;

import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.OneToOne;
import jakarta.persistence.Table;
import lombok.Data;
import lombok.ToString;

@Data
@Entity
@Table(name="recomendaciones")
public class Recomendacion {

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

  private String observaciones;

  @OneToOne(fetch = FetchType.LAZY, optional = false)
```

```
28    @JoinColumn(name = "idcliente")
29    @ToString.Exclude
30    private Cliente cliente;
31
32    @Override
33    public boolean equals(Object obj) {
34      if (this == obj)
35        return true;
36      if (obj == null)
37        return false;
38      if (getClass() != obj.getClass())
39        return false;
40      Recomendacion other = (Recomendacion) obj;
41      return Objects.equals(id, other.id);
42    }
43
44    @Override
45    public int hashCode() {
46      return Objects.hash(id);
47    }
48  }
```

Aprovecharemos, de paso, para implementar los métodos `equals` y `hashCode` para cada clase entidad que definamos, a partir de este momento. También pondremos la anotación `@ToString.Exclude` en los atributos que representen relaciones.

La aplicación ya la hemos probado en el punto anterior, por lo que vamos a pasar a definir el siguiente tipo de relación.

Antes de ello, vamos a explicar un poco cuando hacer uso de `Lazy Fetch` y/o `Eager Fetch`.

# ¿Cuándo utilizar Lazy Fetch y cuando Eager Fetch?

`Lazy Fetch` tiene un cometido concreto, y esto por si sólo ya es motivo para su existencia. El coste de crear objetos puede ser inmenso si todas las relaciones dentro de una entidad (relaciones que a su vez pueden contener sus propias relaciones, y así indefinidamente) fueran iniciadas en cascada desde el momento de creación de la entidad contenedora original. Otra situación en la cual `Lazy Fetch` es sumamente útil es cuando definimos una relación con objetos de gran tamaño (como BLOBs y CLOBs). Esta clase de relaciones suelen definirse como `Lazy` y, en caso de ser necesario el/los objeto/s referenciados, accedidos mediante lógica adicional. Todo esto representa el buen uso de `Lazy Fetch`.

Por supuesto podemos inicializar explícitamente todas las relaciones como `Eager` y olvidarnos de todo lo demás, y esto representa el mal uso de la arquitectura. Tarde o temprano terminaremos incurriendo en costes extra, ya que entre todos los objetos que estaremos inicializando, multitud de ellos serán accedidos en muy raras ocasiones (o incluso ninguna).

En este punto podríamos pensar que "`Lazy Fetch` es un mecanismo mejor que `Eager Fetch`"; nada más lejos de la realidad, pues podemos terminar cometiendo un pecado distinto pero igual de grave, el de marcar toda relación viviente como `Lazy`: puesto que ninguna relación estará inicializada dentro de la entidad contenedora, todos y cada una de los accesos que realicemos sobre ellas conllevará el coste extra de una nueva petición a base de datos, situándonos de nuevo frente a posibles futuros problemas de

agotamiento de recursos.

La solución, como casi siempre, se encuentra en encontrar el equilibrio (en nuestro caso determinar que componentes deben ser `Lazy` y cuales `Eager`). Si bien esto es algo que no puede ser definido con exactitud en el momento de iniciar el desarrollo de una aplicación, lo será en mayor o menor medida durante su periodo de vida (en parte gracias a herramientas externas y en parte gracias al propio comportamiento y/o respuesta de la aplicación ante situaciones de estrés).

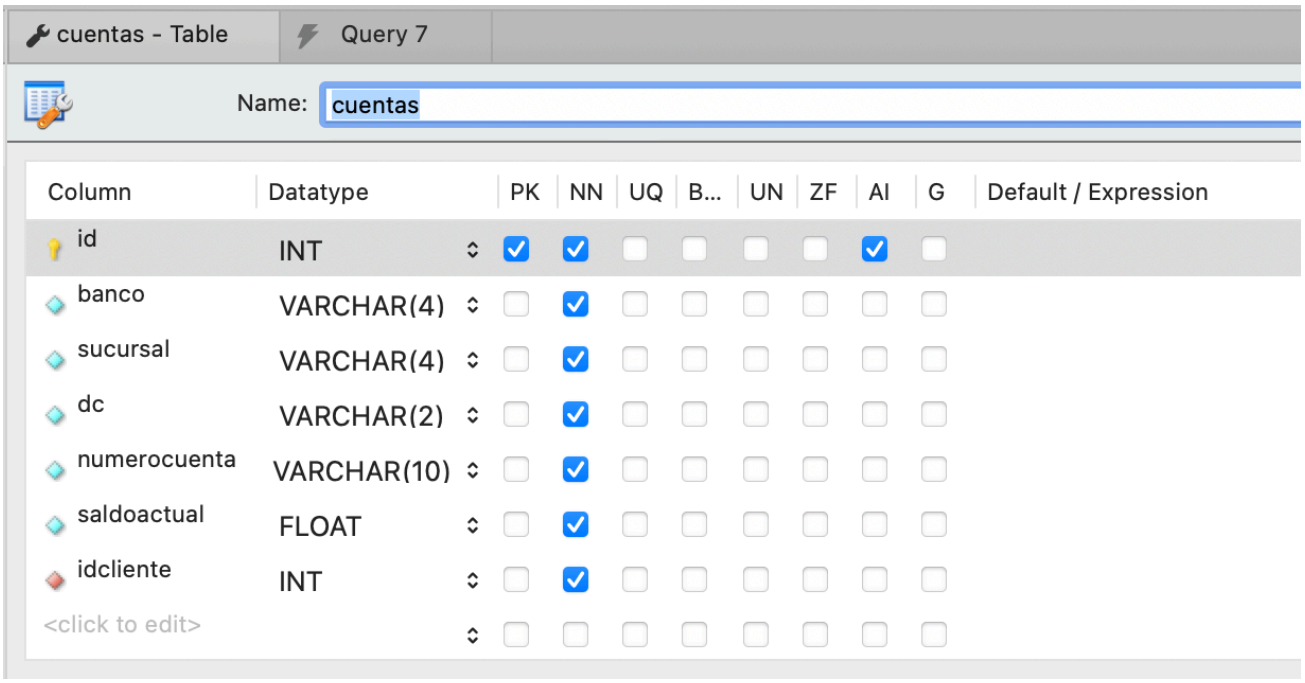
Aún así cada tipo de relación existente en Hibernate aplica, de forma predeterminada, una forma de recuperar los datos:

- OneToMany: LAZY
- ManyToOne: EAGER
- ManyToMany: LAZY
- OneToOne: EAGER

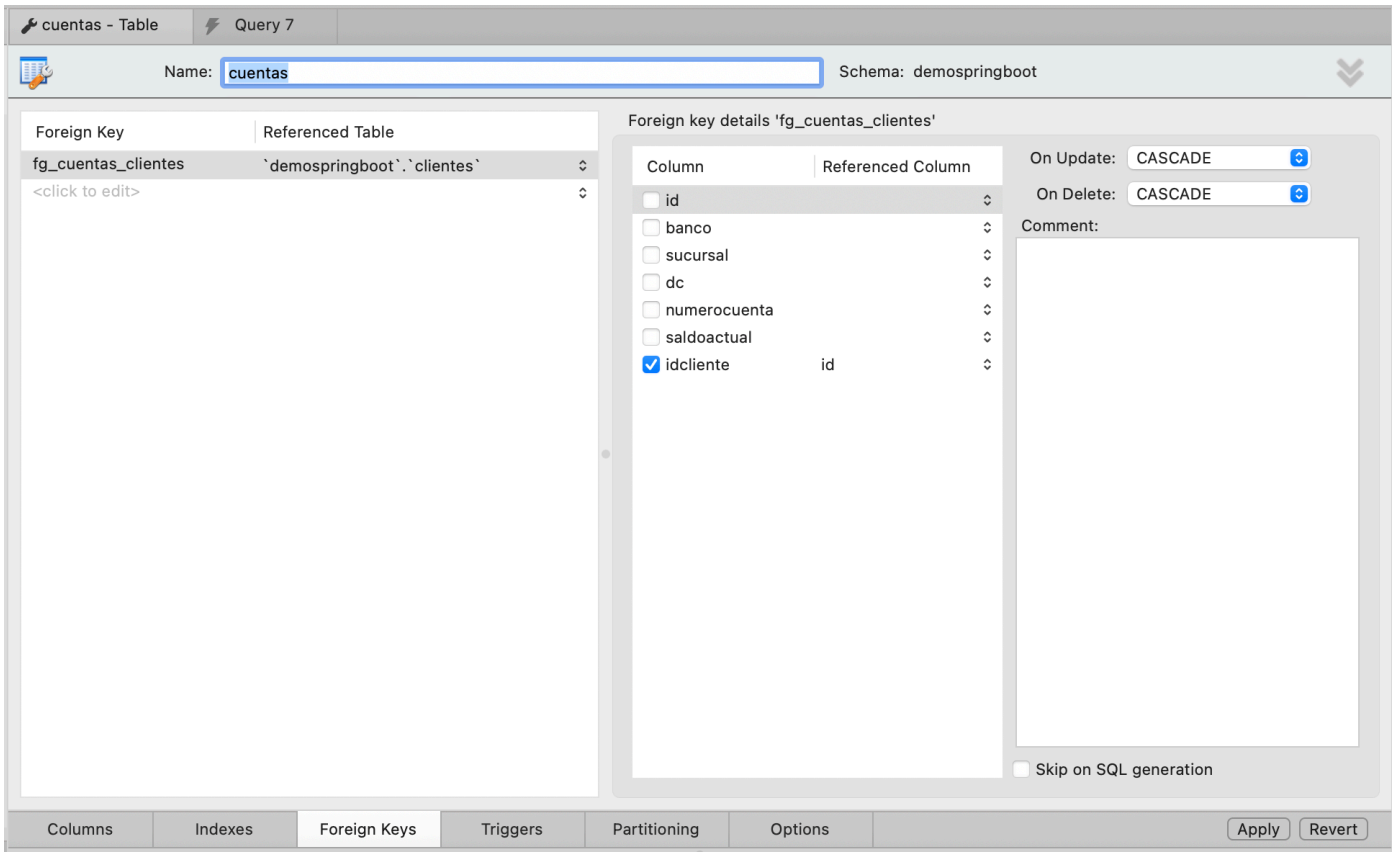
# Relaciones 1 a N

Una vez que ya hemos aprendido a implementar relaciones 1 a 1 es momento de continuar avanzando, y por ello vamos a aprender a implementar relaciones 1 a N. En nuestro ejemplo tenemos varias, pero vamos a centrarnos en implementar la relación que existe entre `Cliente` y `Cuenta`. Por ello, lo primero de todo es crear la estructura en la base de datos, para así poder trabajar sobre ella.

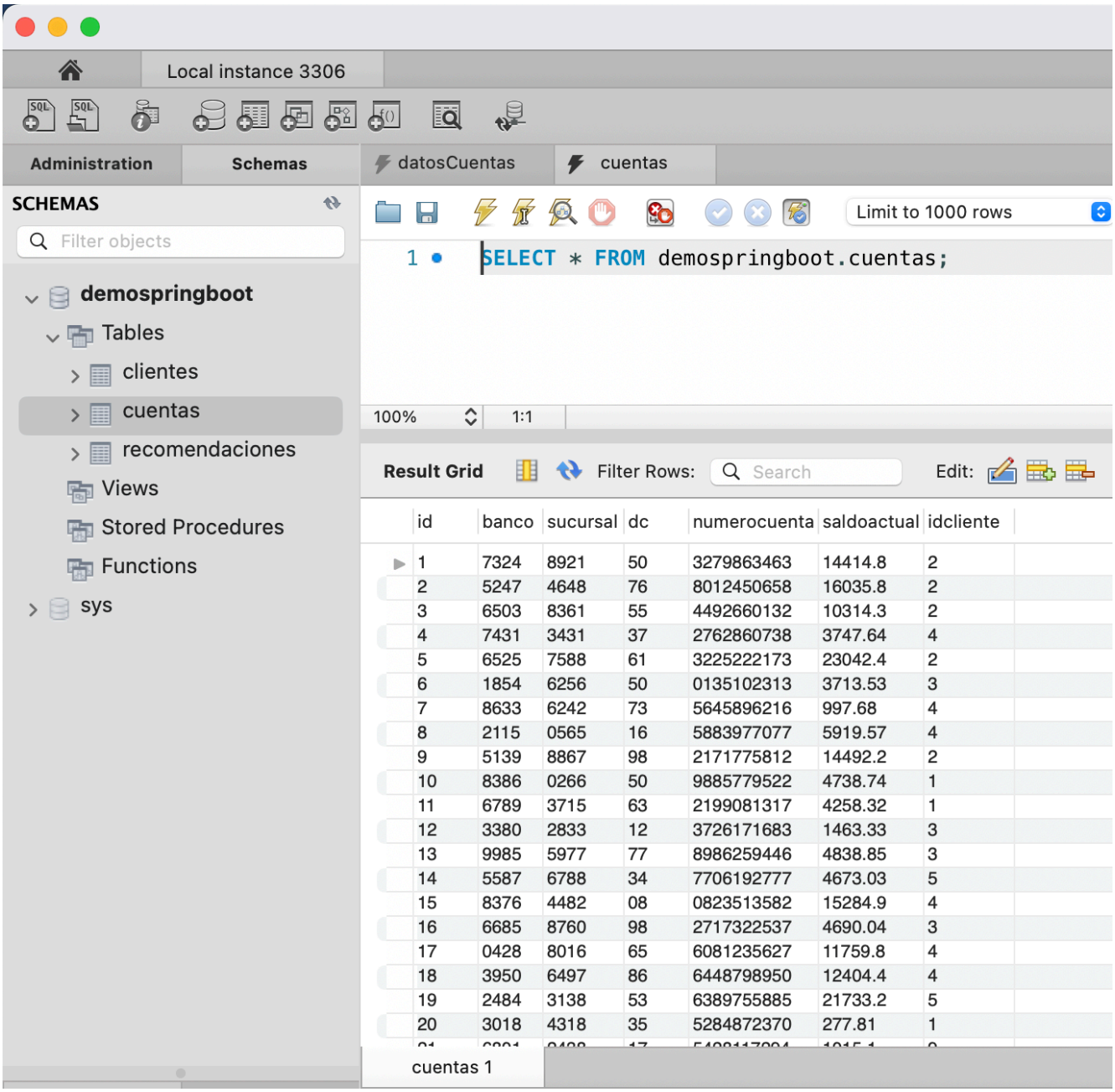
1. Abrimos MySQL WorkBench y nos conectamos con el usuario usuariodemo, que ha sido el que hemos creado.
2. Creamos la tabla `cuentas` que tendrá la siguiente estructura:




3. Recordar que tenemos que crear la clave ajena a la tabla `clientes` sobre la columna `idcliente`.




4. Tras ello poblamos la tabla `cuentas` con datos cargando el fichero `datosCuentas.sql`. Tras ello podremos visualizarlos de la siguiente forma:




5. Con esto ya podemos comenzar a realizar el mapeo en java.

En las relaciones Uno a Muchos, un objeto de la entidad "A" (el lado uno) está relacionado con muchos objetos de la entidad "B" (el lado muchos). En el caso de que la relación sea unidireccional, solo la entidad "A" tiene una referencia a los objetos de tipo "B" y esta relación está representada por una colección (un `List` o un `Set`) y la entidad "A" puede acceder a cada uno de los objetos de tipo "B" de esa colección, pero ojo la entidad "B" no tiene referencia a la entidad "A". Un ejemplo de este tipo de relaciones puede ser un cliente que puede tener muchas cuentas, donde el cliente es el lado Uno de la relación, y las cuentas son el lado Muchos.

Si queremos saber, desde la clase `Cuenta`, quien es su propietario debemos agregar un mapeo bidireccional (`@ManyToOne`) hacia la clase `Cliente`. Y en la clase `Cliente` especificar una relación de uno a muchos (`@OneToMany`) hacia una colección de objetos de tipo `Cuenta`, el cual puede ser una estructura de tipo `List` o `Set` o, dependiendo si queremos manejar orden o no, respectivamente.

La relación bidireccional se destaca por llevar las dos anotaciones, `@OneToMany` y `@ManyToOne`. Si eliminámos una de ellas, se convertiría en una relación unidireccional.

Aunque existen relaciones unidireccionales y bidireccionales, en este tipo de casos, tiene mucho más sentido hablar de relaciones bidireccionales, por lo que vamos a explicar únicamente las relaciones bidireccionales.

Por ello, los pasos a seguir son los siguientes:

1. Creamos la clase `Cuenta.java` en el paquete `com.example.demo.repository.entity` para definir la entidad.
2. Definimos la entidad quedando de la siguiente forma (nos faltará definir la relación):

```java
package com.example.demo.repository.entity;

import java.util.Objects;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.Data;

@Data
@Entity
@Table(name = "cuentas")
public class Cuenta {

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

  private String banco;
  private String sucursal;
  private String dc;

  @Column(name = "numerocuenta")
  private String numeroCuenta;

  @Column(name = "saldoactual")
  private float saldoActual;

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Cuenta other = (Cuenta) obj;
    return Objects.equals(id, other.id);
```

```
  }

  @Override
  public int hashCode() {
    return Objects.hash(id);
  }
}
```

3. Como ya hemos indicado, las relaciones uno a muchos (`@OneToMany`) se caracterizan por tener una entidad como objeto principal y una colección de objetos de otra entidad relacionados directamente. Estas relaciones se definen mediante colecciones, pues tendremos una serie de objetos pertenecientes al objeto principal.
4. Por ello, lo que vamos a hacer es irnos a la clase `Cliente.java` y definir la colección de objetos de la entidad `Cuenta.java`. De esta forma estamos comenzando a definir la relación 1 a N. Quedaría de la siguiente forma:

```
package com.example.demo.repository.entity;

import java.util.List;
import java.util.Objects;

import jakarta.persistence.CascadeType;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.OneToOne;
import jakarta.persistence.Table;
import lombok.Data;
import lombok.ToString;

@Data
@Entity
@Table(name="clientes")
public class Cliente {

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

  @Column(name = "nif")
  private String nif;

  @Column(name = "nombre")
  private String nombre;

  private String apellidos;

```

```
35    @Column(name = "claveseguridad")
36    private String claveSeguridad;
37
38    private String email;
39
40    @OneToOne(fetch = FetchType.LAZY, cascade =  CascadeType.ALL, orphanRemoval =
true, mappedBy = "cliente")
41    @ToString.Exclude
42    private Recomendacion recomendacion;
43
44    private List<Cuenta> listaCuentas;
45
46    @Override
47    public boolean equals(Object obj) {
48      if (this == obj)
49        return true;
50      if (obj == null)
51        return false;
52      if (getClass() != obj.getClass())
53        return false;
54      Cliente other = (Cliente) obj;
55      return Objects.equals(id, other.id);
56    }
57
58    @Override
59    public int hashCode() {
60      return Objects.hash(id);
61    }
62  }
```

5. Ahora tenemos que definir la propiedad que indica que es una relación bidireccional en la entidad `Cuenta.java`, de forma que el atributo que vamos a declarar en dicha clase definirá la relación N a 1. Quedará de la siguiente forma:

```
1  package com.example.demo.repository.entity;
2
3  import java.util.Objects;
4
5  import jakarta.persistence.Column;
6  import jakarta.persistence.Entity;
7  import jakarta.persistence.GeneratedValue;
8  import jakarta.persistence.GenerationType;
9  import jakarta.persistence.Id;
10 import jakarta.persistence.Table;
11 import lombok.Data;
12
13 @Data
14 @Entity
15 @Table(name = "cuentas")
16 public class Cuenta {
17
```

```
18    @Id
19    @GeneratedValue( strategy=GenerationType.IDENTITY)
20    private Long id;
21
22    private String banco;
23    private String sucursal;
24    private String dc;
25
26    @Column(name = "numerocuenta")
27    private String numeroCuenta;
28
29    @Column(name = "saldoactual")
30    private float saldoActual;
31
32    private Cliente cliente;
33
34    @Override
35    public boolean equals(Object obj) {
36      if (this == obj)
37        return true;
38      if (obj == null)
39        return false;
40      if (getClass() != obj.getClass())
41        return false;
42      Cuenta other = (Cuenta) obj;
43      return Objects.equals(id, other.id);
44    }
45
46    @Override
47    public int hashCode() {
48      return Objects.hash(id);
49    }
50  }
```

6. Tras esto comenzamos a mapear la entidad `Cliente.java`, con lo que pondremos la anotación `@OneToMany` y le añadiremos los siguientes parámetros, y modificaremos el constructor del mismo:

```
1  package com.example.demo.repository.entity;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.Objects;
6
7  import jakarta.persistence.CascadeType;
8  import jakarta.persistence.Column;
9  import jakarta.persistence.Entity;
10 import jakarta.persistence.FetchType;
11 import jakarta.persistence.GeneratedValue;
12 import jakarta.persistence.GenerationType;
13 import jakarta.persistence.Id;
14 import jakarta.persistence.OneToMany;
```

```java
import jakarta.persistence.OneToOne;
import jakarta.persistence.Table;
import lombok.Data;
import lombok.ToString;

@Data
@Entity
@Table(name="clientes")
public class Cliente {

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

  @Column(name = "nif")
  private String nif;

  @Column(name = "nombre")
  private String nombre;

  private String apellidos;

  @Column(name = "claveseguridad")
  private String claveSeguridad;

  private String email;

  @OneToOne(fetch = FetchType.LAZY, cascade =  CascadeType.ALL, orphanRemoval = true, mappedBy = "cliente")
  @ToString.Exclude
  private Recomendacion recomendacion;

  @OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL, mappedBy = "cliente")
  private List<Cuenta> listaCuentas;

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Cliente other = (Cliente) obj;
    return Objects.equals(id, other.id);
  }

  @Override
  public int hashCode() {
    return Objects.hash(id);
  }
```

```

  public Cliente() {
    super();
    this.recomendacion = new Recomendacion();
    this.listaCuentas = new ArrayList<Cuenta>();
    this.listaDirecciones = new ArrayList<Direccion>();
  }
}
```

7. Los parámetros que hemos indicado son los siguientes:
    1. `fetch`: aunque ya lo hemos descrito con anterioridad, este parámetro se encarga de definir como recuperar la información. En este caso vamos a usar la opción LAZY.
    2. `cascade`: ya descrito con anterioridad.
    3. `mappedBy`: Indicará el nombre del atributo con el que está representado el objeto en la clase hija. En este caso, el atributo se llamará `cliente`, y se encontrará en la clase `Cuenta`.
8. Tras ello vamos a definir la relación N a 1 en la clase `Cuenta.java`. Quedará de la siguiente forma (ponemos el fragmento de código que modificamos):

```
package com.example.demo.repository.entity;

import java.util.Objects;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;
import lombok.Data;

@Data
@Entity
@Table(name = "cuentas")
public class Cuenta {

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

  private String banco;
  private String sucursal;
  private String dc;

  @Column(name = "numerocuenta")
  private String numeroCuenta;

  @Column(name = "saldoactual")
```

```
32    private float saldoActual;
33
34    @ManyToOne
35    @JoinColumn(name = "idcliente")
36    private Cliente cliente;
37
38    @Override
39    public boolean equals(Object obj) {
40      if (this == obj)
41        return true;
42      if (obj == null)
43        return false;
44      if (getClass() != obj.getClass())
45        return false;
46      Cuenta other = (Cuenta) obj;
47      return Objects.equals(id, other.id);
48    }
49
50    @Override
51    public int hashCode() {
52      return Objects.hash(id);
53    }
54  }
```

9. Si os fijáis hemos declarado el atributo de tipo `Cliente`, el cual lleva dos anotaciones: `@ManyToOne` y `@JoinColumn`. El primero no necesita ningún parámetro de comportamiento (`cascade`, `fetch`,...) puesto que lo vamos a definir en la clase "padre" (en toda relación 1 a muchos hay una tabla padre, que es el que se lleva la cardinalidad 1 y una tabla hija, que es la que se lleva la cardinalidad muchos), y no en la tabla "hija", ya que `cuentas` es la tabla hija.
10. En cambio, en la anotación `@JoinColumn` indicaremos el parámetro `name`, que será el nombre de la columna donde se almacenará el valor la clave ajena en la tabla `cuentas`. En este caso la columna de la tabla `cuentas` que mantiene la relación se denomina `idcliente`.
11. Tras esto hemos terminado con el mapeo de las entidades.

Tras ello es momento de crear el DTO que mapeará la entidad, así como de modificar el DTO de la clase `Cliente.java` para su mapeo. Quedarán de la siguiente forma:

```
1   package com.example.demo.model.dto;
2
3   import java.io.Serializable;
4
5   import com.example.demo.repository.entity.Cliente;
6   import com.example.demo.repository.entity.Cuenta;
7
8   import lombok.Data;
9   import lombok.ToString;
10
11  @Data
12  public class CuentaDTO implements Serializable{
13
```

```
14    private static final long serialVersionUID = 1L;
15    private Long id;
16    private String banco;
17    private String sucursal;
18    private String dc;
19    private String numeroCuenta;
20    private float saldoActual;
21    @ToString.Exclude
22    private ClienteDTO clienteDTO;
23
24    public static CuentaDTO convertToDTO(Cuenta cuenta, ClienteDTO cliente) {
25
26      CuentaDTO cuentaDTO = new CuentaDTO();
27      cuentaDTO.setId(cuenta.getId());
28      cuentaDTO.setBanco(cuenta.getBanco());
29      cuentaDTO.setSucursal(cuenta.getSucursal());
30      cuentaDTO.setDc(cuenta.getDc());
31      cuentaDTO.setNumeroCuenta(cuenta.getNumeroCuenta());
32      cuentaDTO.setSaldoActual(cuenta.getSaldoActual());
33      cuentaDTO.setClienteDTO(cliente);
34
35      return cuentaDTO;
36    }
37
38    public static Cuenta convertToEntity(CuentaDTO cuentaDTO) {
39
40      Cuenta cuenta = new Cuenta();
41      cuenta.setId(cuentaDTO.getId());
42      cuenta.setBanco(cuentaDTO.getBanco());
43      cuenta.setSucursal(cuentaDTO.getSucursal());
44      cuenta.setDc(cuentaDTO.getDc());
45      cuenta.setNumeroCuenta(cuentaDTO.getNumeroCuenta());
46      cuenta.setSaldoActual(cuentaDTO.getSaldoActual());
47
48      Cliente cliente = ClienteDTO.convertToEntity(cuentaDTO.getClienteDTO());
49      cuenta.setCliente(cliente);
50
51      return cuenta;
52    }
53  }
```

Y en cuanto a `ClienteDTO.java` quedará de la siguiente forma:

```
1  package com.example.demo.model.dto;
2
3  import java.io.Serializable;
4  import java.util.ArrayList;
5  import java.util.List;
6
7  import com.example.demo.repository.entity.Cliente;
8  import com.example.demo.repository.entity.Cuenta;
```

```java
import com.example.demo.repository.entity.Recomendacion;

import lombok.Data;
import lombok.ToString;

@Data
public class ClienteDTO implements Serializable{

  private static final long serialVersionUID = 1L;
  private Long id;
  private String nif;
  private String nombre;
  private String apellidos;
  private String claveSeguridad;
  private String email;
  @ToString.Exclude
  private RecomendacionDTO recomendacionDTO;
  @ToString.Exclude
  private List<CuentaDTO> listaCuentasDTO;

  // Convertir una entidad a un DTO
  public static ClienteDTO convertToDTO(Cliente cliente) {

    ClienteDTO clienteDTO = new ClienteDTO();
    clienteDTO.setId(cliente.getId());
    clienteDTO.setNif(cliente.getNif());
    clienteDTO.setNombre(cliente.getNombre());
    clienteDTO.setApellidos(cliente.getApellidos());
    clienteDTO.setClaveSeguridad(cliente.getClaveSeguridad());
    clienteDTO.setEmail(cliente.getEmail());

    RecomendacionDTO rec = RecomendacionDTO.converToDTO(cliente.getRecomendacion(), clienteDTO);
    clienteDTO.setRecomendacionDTO(rec);
    // Cargamos la lista de cuentas
    for(int i=0;i<cliente.getListaCuentas().size();i++) {
      CuentaDTO cuentadto = CuentaDTO.convertToDTO(cliente.getListaCuentas().get(i), clienteDTO);
      clienteDTO.getListaCuentasDTO().add(cuentadto);
    }

    return clienteDTO;
  }

  // Convertir un DTO a una entidad
  public static Cliente convertToEntity(ClienteDTO clienteDTO) {

    Cliente cliente = new Cliente();
    cliente.setId(clienteDTO.getId());
    cliente.setNif(clienteDTO.getNif());
    cliente.setNombre(clienteDTO.getNombre());
    cliente.setApellidos(clienteDTO.getApellidos());
```

```
    cliente.setClaveSeguridad(clienteDTO.getClaveSeguridad());
    cliente.setEmail(clienteDTO.getEmail());

    Recomendacion rec = RecomendacionDTO.convertToEntity(clienteDTO.getRecomendacionDTO(), cliente);
    cliente.setRecomendacion(rec);

    // Cargamos la lista de cuentas
    for(int i=0;i<clienteDTO.getListaCuentasDTO().size();i++) {
      Cuenta cuenta = CuentaDTO.convertToEntity(clienteDTO.getListaCuentasDTO().get(i));
      cliente.getListaCuentas().add(cuenta);
    }

    return cliente;
  }

  // Constructor vacio
  public ClienteDTO() {
    super();
    this.recomendacionDTO = new RecomendacionDTO();
    this.listaCuentasDTO = new ArrayList<CuentaDTO>();
  }
}
```

Ya tenemos los DTO's. Con ello, lo primero que haremos será modificar la vista `clientes.html` de forma que añadiremos una opción más a cada cliente que nos permitirá poder visualizar las cuentas de cada uno de ellos.

La URL donde redireccionaremos al controlador será `/clientes/{idCliente}/cuentas`, donde `idCliente` será el identificador del cliente. Por ejemplo `/clientes/1/cuentas` mostrará las cuentas del cliente 1, siendo la URL similar a la operación de mostrar los datos del cliente, que era `/clientes/1`.

El código de `clientes.html` quedará de la siguiente forma:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <title>Mantenimiento de clientes</title>
</head>
<body>
  <div>
    <br>
    <h2 th:if="${listaClientesDTO.isEmpty()}">No hay clientes</h2>
    <div th:if="${!listaClientesDTO.isEmpty()}">
      <h2>Mantenimiento de clientes</h2>
      <table border="1">
        <thead>
          <tr>
            <td>id</td>
            <td>Nombre</td>
```

```
            <td>Apellidos</td>
            <td>NIF</td>
            <td>Correo electrónico</td>
            <td>Editar</td>
            <td>Eliminar</td>
            <th>Cuentas</th>
            <th>Direcciones</th>
          </tr>
        </thead>
        <tbody>
          <tr th:each="clienteDTO : ${listaClientesDTO}">
            <td><a
              th:href="@{/clientes/{idCliente}(idCliente=${clienteDTO.id})}"
              th:text="${clienteDTO.id}"></a></td>
            <td th:text="${clienteDTO.nombre}" />
            <td th:text="${clienteDTO.apellidos}" />
            <td th:text="${clienteDTO.nif}" />
            <td th:text="${clienteDTO.email}" />
            <td><a
              th:href="@{/clientes/update/{idCliente}(idCliente=${clienteDTO.id})}"
/>Editar</a></td>
            <td><a
              th:href="@{/clientes/delete/{idCliente}(idCliente=${clienteDTO.id})}"
/>Eliminar</a></td>
            <td><a
              th:href="@{/clientes/{idCliente}/cuentas
(idCliente=${clienteDTO.id})}">Cuentas</a></td>
            <td><a
              th:href="@{/clientes/{idCliente}/direcciones
(idCliente=${clienteDTO.id})}">Direcciones</a></td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
  <br>
  <div>
    <a th:href="@{/clientes/add}">Nuevo cliente</a>
    <br>
    <a th:href="@{/}">Volver al inicio</a>
  </div>
</body>
</html>
```

Ahora crearemos un controlador, tendrá el nombre de `CuentaController.java`, que mantendrá las operaciones que se pueden hacer con las cuentas. Es cierto que podríamos implementar estas operaciones dentro de la clase `ClienteController.java`, pero es mejor especializar los controladores y no crear un cajón desastre.

El controlador `CuentaController.java` tendrá el siguiente código:

```java
package com.example.demo.web.controller;

import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.servlet.ModelAndView;

import com.example.demo.model.dto.ClienteDTO;
import com.example.demo.model.dto.CuentaDTO;
import com.example.demo.service.ClienteService;
import com.example.demo.service.CuentaService;

@Controller
public class CuentaController {

  private static final Logger log = LoggerFactory.getLogger(CuentaController.class);

  @Autowired
  private CuentaService cuentaService;

  @Autowired
  private ClienteService clienteService;

  @GetMapping("/clientes/{idCliente}/cuentas")
  public ModelAndView findAllByCliente(@PathVariable Long idCliente) {

    log.info("ClienteController - findAll: Mostramos todos los clientes");

    // Obtenemos el cliente
    ClienteDTO clienteDTO = new ClienteDTO();
    clienteDTO.setId(idCliente);
    clienteDTO = clienteService.findById(clienteDTO);

    // Obtenemos la lista de cuentas
    List<CuentaDTO> listaCuentasDTO = cuentaService.findAllByCliente(clienteDTO);

    ModelAndView mav = new ModelAndView("cuentas");
    mav.addObject("clienteDTO", clienteDTO);
    mav.addObject("listaCuentasDTO", listaCuentasDTO);

    return mav;
  }
}
```

La operación `findAllByCliente` será la encargada de mapear la URL `/clientes/{idCliente}/cuentas`, de forma que recibirá el id del cliente. Vamos a obtener el cliente para mostrar sus datos por pantalla (objeto de tipo `ClienteDTO`). Ya solo nos queda que con el objeto de tipo `ClienteDTO` obtengamos la lista

de cuentas mediante el servicio `CuentaService.java` (que lo crearemos a continuación). Tras obtener estos datos creamos un objeto de tipo `ModelAndView`, le pasamos el clienteDTO y la lista de cuentas al mismo y devolveremos la vista que mostrará la lista de cuentas y el cliente.

Es el momento de definir las clases de la capa de servicios que van a trabajar con las cuentas. Siguiendo el mismo patrón que hemos usado para crear la interfaz `ClienteService.java` y la implementación del mismo, que es `ClienteServiceImpl.java`, vamos a definir, como interfaz, `CuentaService.java` y como implementación `CuentaServiceImpl.java`.

La interfaz `CuentaService.java` contendrá lo siguiente:

```
package com.example.demo.service;

import java.util.List;

import com.example.demo.model.dto.ClienteDTO;
import com.example.demo.model.dto.CuentaDTO;

public interface CuentaService {

  public List<CuentaDTO> findAllByCliente(ClienteDTO clienteDTO);

}
```

Fijaros que solo hemos definido un método en la interfaz, puesto que no necesitamos más, por ahora. Ahora creamos la clase `CuentaServiceImpl.java` y procederemos a realizar la llamada a `CuentaRepository` como hemos hecho con los clientes. Pero esto tiene un problema y es que el repositorio define métodos estándar, como obtener todas las cuentas, pero no define un método que nos permita obtener las cuentas de un cliente en concreto.

```
package com.example.demo.service;

import java.util.ArrayList;
import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.example.demo.model.dto.ClienteDTO;
import com.example.demo.model.dto.CuentaDTO;
import com.example.demo.repository.dao.CuentaRepository;
import com.example.demo.repository.entity.Cuenta;

@Service
public class CuentaServiceImpl implements CuentaService{

  private static final Logger log = LoggerFactory.getLogger(CuentaServiceImpl.class);

  @Autowired
```

```
22    private CuentaRepository cuentaRepository;
23
24    @Override
25    public List<CuentaDTO> findAllByCliente(ClienteDTO clienteDTO) {
26      log.info("CuentaServiceImpl - finfindAllByClientedAll: Lista de todas las cuentas
    del cliente: " + clienteDTO.getId());
27
28      // Obtenemos la lista de cuentas del cliente
29      List<Cuenta> lista = (List<Cuenta>)
    cuentaRepository.findAllByCliente(clienteDTO.getId());
30      // Creamos una lista de CuentaDTO que será la que devolvamos al controlador
31      List<CuentaDTO> listaResultadoDTO = new ArrayList<CuentaDTO>();
32      // Recorremos la lista de cuentas y las mapeamos a DTO
33      for (int i = 0; i < lista.size(); ++i) {
34          listaResultadoDTO.add(CuentaDTO.convertToDTO(lista.get(i), clienteDTO));
35      }
36      // Devolvemos la lista de DTO's
37      return listaResultadoDTO;
38    }
39  }
```

Por ello lo que vamos a hacer es extender la funcionalidad del repositorio `CuentaRepository.java`, añadiendo una nueva operación denominada `findAllByCliente`, de forma que obtendremos la lista de cuentas de un cliente, y esto lo haremos mediante la anotación `@Query`.

La anotación `@Query` nos permite definir las consultas que han de lanzar los métodos, incluyendo parámetros. Lo comprenderéis enseguida, en cuanto veáis como queda el interfaz `CuentaRepository.java`:

```
1   package com.example.demo.repository.dao;
2
3   import java.util.List;
4
5   import org.springframework.data.jpa.repository.JpaRepository;
6   import org.springframework.data.jpa.repository.Query;
7   import org.springframework.stereotype.Repository;
8
9   import com.example.demo.repository.entity.Cuenta;
10
11  import jakarta.transaction.Transactional;
12
13  @Repository
14  @Transactional
15  public interface CuentaRepository extends JpaRepository<Cuenta, Long>{
16
17    @Query(value = "SELECT * FROM cuentas WHERE idcliente = ?1", nativeQuery = true)
18    public List<Cuenta> findAllByCliente(Long idCliente);
19  }
```

Si os fijáis en la consulta que hemos definido realizamos una selección sobre la tabla `cuentas`, donde nos devolverá las cuentas que sean del cliente que le pasamos por parámetro, que es le mismo que recibimos nosotros por parámetro. Se podía haber definido, también, de la siguiente forma:

```
package com.example.demo.repository.dao;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import com.example.demo.repository.entity.Cuenta;

import jakarta.transaction.Transactional;

@Repository
@Transactional
public interface CuentaRepository extends JpaRepository<Cuenta, Long>{

  /*
  @Query(value = "SELECT * FROM cuentas WHERE idcliente = ?1", nativeQuery = true)
  public List<Cuenta> findAllByCliente(Long idCliente);
  */

  @Query(value = "SELECT * FROM cuentas WHERE idcliente = :idc", nativeQuery = true)
  public List<Cuenta> findAllByCliente(@Param("idc") Long idCliente);

}
```

Lo que sí es muy importante es el parámetro `nativeQuery`, puesto que estamos ejecutando código SQL en modo nativo, y no código SQL propio de Hibernate, que se denomina `JPQL`. Ya hablaremos más adelante del mismo.

Ya solo nos queda definir la vista `cuentas.html`, que mostrará los datos básicos del cliente y la lista de cuentas. El código será el siguiente:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <title>Mantenimiento de cuentas</title>
</head>
<body>
  <div>
    <br>
    <h2 th:if="${listaCuentasDTO.isEmpty()}">No hay cuentas para el cliente
      <td th:utext="${clienteDTO.nombre}"></td> <td
th:utext="${clienteDTO.apellidos}"></td>
    </h2>
```

```
13      <div th:if="${!listaCuentasDTO.isEmpty()}">
14        <h2>Mantenimiento de cuentas del cliente
15          <td th:utext="${clienteDTO.nombre}"></td> <td
th:utext="${clienteDTO.apellidos}"></td>
16        </h2>
17        <br>
18        <table border="1">
19          <thead>
20            <tr>
21              <th>id</th>
22              <th>Banco</th>
23              <th>Sucursal</th>
24              <th>dc</th>
25              <th>Número de cuenta</th>
26              <th>Saldo actual</th>
27              <th>Movimientos</th>
28              <th>Eliminar</th>
29            </tr>
30          </thead>
31          <tbody>
32            <tr th:each="cuentaDTO : ${listaCuentasDTO}">
33              <td th:utext="${cuentaDTO.id}">...</td>
34              <td th:utext="${cuentaDTO.banco}">...</td>
35              <td th:utext="${cuentaDTO.sucursal}">...</td>
36              <td th:utext="${cuentaDTO.dc}">...</td>
37              <td th:utext="${cuentaDTO.numeroCuenta}">...</td>
38              <td th:utext="${cuentaDTO.saldoActual}">...</td>
39              <td><a th:href="@{/clientes/{idCliente}/cuentas/movimientos
(idCliente=${clienteDTO.id})}">Visualizar movimientos</a></td>
40              <td><a th:href="@{/clientes/{idCliente}/cuentas/delete/{idCuenta}
(idCliente=${clienteDTO.id},idCuenta=${cuentaDTO.id})}">Eliminar</a></td>
41            </tr>
42          </tbody>
43        </table>
44      </div>
45      <br>
46      <div>
47        <a th:href="@{/clientes/{idCliente}/cuentas/add
(idCliente=${clienteDTO.id})}">Nueva cuenta</a>
48        <br>
49        <a th:href="@{/clientes}">Volver a la lista de clientes</a>
50      </div>
51    </div>
52  </body>
53  </html>
54
```

Con esto ya solo nos queda ejecutar el programa y ver el resultado:

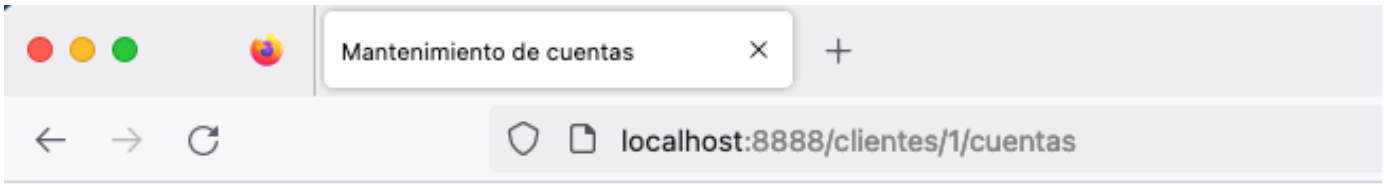
Mantenimiento de clientes

localhost:8888/clientes

# Mantenimiento de clientes

| id | Nombre | Apellidos | NIF | Correo electrónico | Editar | Eliminar | **Cuentas** |
|---|---|---|---|---|---|---|---|
| 1 | Keiko | Meadows | 44999594K | meadowskeiko6933@icloud.net | Editar | Eliminar | Cuentas |
| 2 | Whoopi | Phelps | 28553744Y | w-phelps@aol.ca | Editar | Eliminar | Cuentas |
| 3 | Hilary | Mcmahon | 25861374E | hmcmahon6589@google.org | Editar | Eliminar | Cuentas |
| 4 | Hunter | Frederick | 12242423G | h_frederick@icloud.edu | Editar | Eliminar | Cuentas |
| 5 | Diana | Dickerson | 51724739D | d-dickerson5965@aol.net | Editar | Eliminar | Cuentas |
| 6 | Amber | Ramirez | 13176696D | amber.ramirez6505@protonmail.org | Editar | Eliminar | Cuentas |
| 7 | Chava | Floyd | 82568848H | cfloyd@hotmail.couk | Editar | Eliminar | Cuentas |
| 8 | Tatum | Walsh | 03768520L | walsh_tatum2491@outlook.couk | Editar | Eliminar | Cuentas |
| 9 | Ezekiel | Woodard | 51976869H | e-woodard@icloud.couk | Editar | Eliminar | Cuentas |
| 10 | Coby | Lester | 55172493N | c-lester7681@hotmail.couk | Editar | Eliminar | Cuentas |
| 11 | Brielle | Howard | 48037186R | b_howard@aol.com | Editar | Eliminar | Cuentas |
| 12 | Candace | Bates | 71227610F | c-bates@google.org | Editar | Eliminar | Cuentas |
| 13 | Eagan | Carpenter | 41124626J | e_carpenter9768@outlook.org | Editar | Eliminar | Cuentas |
| 14 | Yvette | Slater | 86926685Y | yslater@google.edu | Editar | Eliminar | Cuentas |
| 15 | Heather | Abbott | 18829162J | abbottheather@hotmail.org | Editar | Eliminar | Cuentas |
| 16 | Kessie | Rich | 10159358C | k_rich804@google.net | Editar | Eliminar | Cuentas |
| 17 | Keelie | Forbes | 61816073X | k-forbes1767@hotmail.com | Editar | Eliminar | Cuentas |
| 18 | Dale | Glass | 27682739N | gdale@google.ca | Editar | Eliminar | Cuentas |
| 19 | Farrah | Farmer | 91124511M | farrah-farmer6033@hotmail.com | Editar | Eliminar | Cuentas |
| 20 | Graham | Casey | 21865746E | g.casey3482@google.com | Editar | Eliminar | Cuentas |

Nuevo cliente
Volver al inicio




# Mantenimiento de cuentas del cliente Keiko Meadows

| id | Banco | Sucursal | dc | Número de cuenta | Saldo actual | Movimientos | Eliminar |
|---|---|---|---|---|---|---|---|
| 10 | 8386 | 0266 | 50 | 9885779522 | 4738.74 | Visualizar movimientos | Eliminar |
| 11 | 6789 | 3715 | 63 | 2199081317 | 4258.32 | Visualizar movimientos | Eliminar |
| 20 | 3018 | 4318 | 35 | 5284872370 | 277.81 | Visualizar movimientos | Eliminar |

Volver a la lista de clientes

## Eliminando elementos hijos de una relación 1 a N

Entre todo esto vamos a implementar la operación de eliminar una cuenta, ya que se trata de una operación sencilla pero que puede causar problemas a la hora de implementarla.

La cosa es que si borramos un elemento padre, tal como tenemos configurado el sistema, borraria todos los hijos, por lo que en esto no tenemos problema. El problema nos surge cuando borramos un elemento hijo que se encuentra dentro de un padre, por ejemplo borramos una cuenta que se encuentra dentro del cliente. El problema surge cuando localizamos el cliente y mandamos borrar la cuenta mediante el método `deleteById(Long id)` que está implementando en el repositorio. Como existe una relación entre ambos, si ejecutamos dicho método no borrará la cuenta, puesto que tenemos la cuenta en la relación. Por ello proponemos 2 soluciones al problema:

- Método 1: Mandar borrar la cuenta mediante `deleteById(Long id)` sin haber localizado el cliente (de esta forma no se carga la relación antes de borrar la cuenta).
- Método 2: Cargar el cliente y la cuenta, eliminar la cuenta de la lista de cuentas que mantiene el cliente y, como se ha eliminado la relación entre ellos, podemos ejecutar el método `deleteById(Long id)` eliminando así la cuenta, y tras ello, ejecutar `save()` sobre el cliente, de forma que almacenamos de esta forma la situación nueva del cliente.

Vamos a proceder a implementar los dos casos, aunque solo podremos usar uno al mismo tiempo (usaremos, por defecto, el primer método que es más sencillo, aunque implementaremos los dos).

**Método 1**

Este método es el más sencillo, puesto que, sin recuperar el objeto cliente procedemos al borrado de la cuenta por id diréctamente.

Modificaremos la clase `CuentaController.java` de forma que quedará de la siguiente forma:

```java
package com.example.demo.web.controller;

import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.servlet.ModelAndView;

import com.example.demo.model.dto.ClienteDTO;
import com.example.demo.model.dto.CuentaDTO;
import com.example.demo.service.ClienteService;
import com.example.demo.service.CuentaService;

@Controller
public class CuentaController {

  private static final Logger log = LoggerFactory.getLogger(CuentaController.class);

  @Autowired
  private CuentaService cuentaService;

  @Autowired
  private ClienteService clienteService;

  @GetMapping("/clientes/{idCliente}/cuentas")
  public ModelAndView findAllByCliente(@PathVariable Long idCliente) {

    log.info("CuentaController - findAllByCliente: Mostramos todas las cuentas del cliente: " + idCliente);

    // Obtenemos el cliente
    ClienteDTO clienteDTO = new ClienteDTO();
    clienteDTO.setId(idCliente);
    clienteDTO = clienteService.findById(clienteDTO);

    // Obtenemos la lista de cuentas
    List<CuentaDTO> listaCuentasDTO = cuentaService.findAllByCliente(clienteDTO);

    ModelAndView mav = new ModelAndView("cuentas");
    mav.addObject("clienteDTO", clienteDTO);
    mav.addObject("listaCuentasDTO", listaCuentasDTO);

    return mav;
  }

  // Borrar una cuenta
```

```
52    @GetMapping("/clientes/{idCliente}/cuentas/delete/{idCuenta}")
53    public ModelAndView delete(@PathVariable("idCliente") Long idCliente,
54        @PathVariable("idCuenta") Long idCuenta) {
55
56      log.info("CuentaController - delete: Usamos el metodo 1 para borrar la cuenta:" +
    idCuenta);
57
58      // Creamos la cuentaDTO
59      CuentaDTO cuentaDTO = new CuentaDTO();
60      cuentaDTO.setId(idCuenta);
61
62      // Invocamos al metodo de borrar
63      cuentaService.delete(cuentaDTO);
64
65      // Redireccionamos para volver a invocar al metodo que escucha /clientes
66      ModelAndView mav = new ModelAndView("redirect:/clientes/{idCliente}/cuentas");
67
68      return mav;
69    }
70  }
```

La interfaz `CuentaService.java` quedará de la siguiente forma:

```
1  package com.example.demo.service;
2
3  import java.util.List;
4
5  import com.example.demo.model.dto.ClienteDTO;
6  import com.example.demo.model.dto.CuentaDTO;
7
8  public interface CuentaService {
9
10   public List<CuentaDTO> findAllByCliente(ClienteDTO clienteDTO);
11   public void delete(CuentaDTO cuentaDTO);
12 }
```

En cuanto a la clase `CuentaServiceImpl.java` quedará de la siguiente forma:

```
1  package com.example.demo.service;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.Optional;
6
7  import org.slf4j.Logger;
8  import org.slf4j.LoggerFactory;
9  import org.springframework.beans.factory.annotation.Autowired;
```

```java
import org.springframework.stereotype.Service;

import com.example.demo.model.dto.ClienteDTO;
import com.example.demo.model.dto.CuentaDTO;
import com.example.demo.repository.dao.ClienteRepository;
import com.example.demo.repository.dao.CuentaRepository;
import com.example.demo.repository.entity.Cliente;
import com.example.demo.repository.entity.Cuenta;

@Service
public class CuentaServiceImpl implements CuentaService {

  private static final Logger log = LoggerFactory.getLogger(CuentaServiceImpl.class);

  @Autowired
  private CuentaRepository cuentaRepository;

  @Autowired
  private ClienteRepository clienteRepository;

  @Override
  public List<CuentaDTO> findAllByCliente(ClienteDTO clienteDTO) {
    log.info("CuentaServiceImpl - finfindAllByClientedAll: Lista de todas las cuentas del cliente: "
        + clienteDTO.getId());

    // Obtenemos la lista de cuentas del cliente
    List<Cuenta> lista = (List<Cuenta>) cuentaRepository.findAllByCliente(clienteDTO.getId());
    // Creamos una lista de CuentaDTO que será la que devolvamos al controlador
    List<CuentaDTO> listaResultadoDTO = new ArrayList<CuentaDTO>();
    // Recorremos la lista de cuentas y las mapeamos a DTO
    for (int i = 0; i < lista.size(); ++i) {
      listaResultadoDTO.add(CuentaDTO.convertToDTO(lista.get(i), clienteDTO));
    }
    // Devolvemos la lista de DTO's
    return listaResultadoDTO;
  }

  /*
   * Metodo 1 de borrado de un hijo en una relacion 1 a N:
   * ejecutamos directamente el borrado por medio del id
   */
  @Override
  public void delete(CuentaDTO cuentaDTO) {
    log.info("CuentaServiceImpl - delete: Metodo 1: borramos la cuenta: " + cuentaDTO.toString());

    cuentaRepository.deleteById(cuentaDTO.getId());
  }
}
```

Con esto, si probamos el borrado podemos ver que funciona correctamente.

### Método 2

Este caso es algo más complicado, puesto que hay que obtener tanto el cliente como la cuenta, desvincular la cuenta del cliente, borrar la cuenta y salvar los cambios del cliente.

Por ello, modificaremos la clase `CuentaController.java` que quedará de la siguiente forma:

```
package com.example.demo.web.controller;

import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.servlet.ModelAndView;

import com.example.demo.model.dto.ClienteDTO;
import com.example.demo.model.dto.CuentaDTO;
import com.example.demo.service.ClienteService;
import com.example.demo.service.CuentaService;

@Controller
public class CuentaController {

  private static final Logger log = LoggerFactory.getLogger(CuentaController.class);

  @Autowired
  private CuentaService cuentaService;

  @Autowired
  private ClienteService clienteService;

  @GetMapping("/clientes/{idCliente}/cuentas")
  public ModelAndView findAllByCliente(@PathVariable Long idCliente) {

    log.info("CuentaController - findAllByCliente: Mostramos todas las cuentas del cliente: " + idCliente);

    // Obtenemos el cliente
    ClienteDTO clienteDTO = new ClienteDTO();
    clienteDTO.setId(idCliente);
    clienteDTO = clienteService.findById(clienteDTO);
```

```
40
41      // Obtenemos la lista de cuentas
42      List<CuentaDTO> listaCuentasDTO = cuentaService.findAllByCliente(clienteDTO);
43
44      ModelAndView mav = new ModelAndView("cuentas");
45      mav.addObject("clienteDTO", clienteDTO);
46      mav.addObject("listaCuentasDTO", listaCuentasDTO);
47
48      return mav;
49    }
50
51    // Borrar una cuenta
52    @GetMapping("/clientes/{idCliente}/cuentas/delete/{idCuenta}")
53    public ModelAndView delete(@PathVariable("idCliente") Long idCliente,
54        @PathVariable("idCuenta") Long idCuenta) {
55
56      //log.info("CuentaController - delete: Usamos el metodo 1 para borrar la cuenta:"
    + idCuenta);
57      log.info("CuentaController - delete: Usamos el metodo 2 para borrar la cuenta:" +
    idCuenta);
58
59      // Creamos el clienteDTO
60      ClienteDTO clienteDTO = new ClienteDTO();
61      clienteDTO.setId(idCliente);
62
63      // Creamos la cuentaDTO
64      CuentaDTO cuentaDTO = new CuentaDTO();
65      cuentaDTO.setId(idCuenta);
66
67      // Invocamos al metodo de borrar
68      // cuentaService.delete(cuentaDTO);
69      cuentaService.delete(cuentaDTO, clienteDTO);
70
71      // Redireccionamos para volver a invocar al metodo que escucha /clientes
72      ModelAndView mav = new ModelAndView("redirect:/clientes/{idCliente}/cuentas");
73
74      return mav;
75    }
76  }
```

La interfaz `CuentaService.java` quedará de la siguiente forma (sobrecargamos el método `delete()`):

```java
package com.example.demo.service;

import java.util.List;

import com.example.demo.model.dto.ClienteDTO;
import com.example.demo.model.dto.CuentaDTO;

public interface CuentaService {

  public List<CuentaDTO> findAllByCliente(ClienteDTO clienteDTO);
  public void delete(CuentaDTO cuentaDTO);
  public void delete(CuentaDTO cuentaDTO, ClienteDTO clienteDTO);
}
```

La clase `CuentaServiceImpl.java` quedará de la siguiente forma:

```java
package com.example.demo.service;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.example.demo.model.dto.ClienteDTO;
import com.example.demo.model.dto.CuentaDTO;
import com.example.demo.repository.dao.ClienteRepository;
import com.example.demo.repository.dao.CuentaRepository;
import com.example.demo.repository.entity.Cliente;
import com.example.demo.repository.entity.Cuenta;

@Service
public class CuentaServiceImpl implements CuentaService {

  private static final Logger log = LoggerFactory.getLogger(CuentaServiceImpl.class);

  @Autowired
  private CuentaRepository cuentaRepository;

  @Autowired
  private ClienteRepository clienteRepository;

  @Override
  public List<CuentaDTO> findAllByCliente(ClienteDTO clienteDTO) {
    log.info("CuentaServiceImpl - finfindAllByClientedAll: Lista de todas las cuentas del cliente: "
```

```
33          + clienteDTO.getId());
34
35      // Obtenemos la lista de cuentas del cliente
36      List<Cuenta> lista = (List<Cuenta>)
cuentaRepository.findAllByCliente(clienteDTO.getId());
37      // Creamos una lista de CuentaDTO que será la que devolvamos al controlador
38      List<CuentaDTO> listaResultadoDTO = new ArrayList<CuentaDTO>();
39      // Recorremos la lista de cuentas y las mapeamos a DTO
40      for (int i = 0; i < lista.size(); ++i) {
41        listaResultadoDTO.add(CuentaDTO.convertToDTO(lista.get(i), clienteDTO));
42      }
43      // Devolvemos la lista de DTO's
44      return listaResultadoDTO;
45    }
46
47    /*
48     * Metodo 1 de borrado de un hijo en una relacion 1 a N:
49     * ejecutamos directamente el borrado por medio del id
50     */
51    @Override
52    public void delete(CuentaDTO cuentaDTO) {
53      log.info("CuentaServiceImpl - delete: Metodo 1: borramos la cuenta: " +
cuentaDTO.toString());
54
55      cuentaRepository.deleteById(cuentaDTO.getId());
56    }
57
58    /*
59     * Metodo 2 de borrado de un hijo en una relacion 1 a N:
60     * ejecutamos directamente el borrado por medio del id
61     */
62    @Override
63    public void delete(CuentaDTO cuentaDTO, ClienteDTO clienteDTO) {
64      log.info("CuentaServiceImpl - delete: Metodo 1: borramos la cuenta: " +
cuentaDTO.toString());
65      // Localizamos ambos objetos
66      Optional<Cliente> cliente = clienteRepository.findById(clienteDTO.getId());
67      Optional<Cuenta> cuenta = cuentaRepository.findById(cuentaDTO.getId());
68
69      // Eliminamos la cuenta de la lista de cuentas del cliente.
70      // Con esto eliminamos la relacion y podemos borrar la cuenta
71      cliente.get().getListaCuentas().remove(cuenta.get());
72
73      // Eliminamos la cuenta
74      cuentaRepository.deleteById(cuenta.get().getId());
75
76      // Salvamos el cliente con la nueva situacioni
77      clienteRepository.save(cliente.get());
78    }
79  }
```

Si nos fijamos borramos del cliente, dentro del conjunto de cuentas (que se almacena en la variable listaCuentas) la cuenta en cuestión. El problema es que la variable ya no es una variable de tipo `List` sino de tipo `Set`, por lo que no podemos usar el `remove()` directamente, puesto que cuando la implementación es de tipo `Set` la comparación no se realiza mediante los métodos `equals()` y `hashcode()` de la clase `Cuenta.java`.

Para que esto funcione hemos de implementar el interfaz `Comparable` dentro de la clase `Cuenta.java`, de forma que nos permitirá realizar esta comparación de forma correcta, funcionando así el borrado. Por ello modificaremos la clase `Cuenta.java` de la siguiente forma:

```
1  package com.example.demo.repository.entity;
2
3  import java.util.Objects;
4
5  import jakarta.persistence.Column;
6  import jakarta.persistence.Entity;
7  import jakarta.persistence.GeneratedValue;
8  import jakarta.persistence.GenerationType;
9  import jakarta.persistence.Id;
10 import jakarta.persistence.JoinColumn;
11 import jakarta.persistence.ManyToOne;
12 import jakarta.persistence.Table;
13 import lombok.Data;
14
15 @Data
16 @Entity
17 @Table(name = "cuentas")
18 public class Cuenta implements Comparable {
19
20   @Id
21   @GeneratedValue( strategy=GenerationType.IDENTITY)
22   private Long id;
23
24   private String banco;
25   private String sucursal;
26   private String dc;
27
28   @Column(name = "numerocuenta")
29   private String numeroCuenta;
30
31   @Column(name = "saldoactual")
32   private float saldoActual;
33
34   @ManyToOne
35   @JoinColumn(name = "idcliente")
36   private Cliente cliente;
37
38   @Override
39   public boolean equals(Object obj) {
40     if (this == obj)
41       return true;
```

```
42      if (obj == null)
43        return false;
44      if (getClass() != obj.getClass())
45        return false;
46      Cuenta other = (Cuenta) obj;
47      return Objects.equals(id, other.id);
48    }
49
50    @Override
51    public int hashCode() {
52      return Objects.hash(id);
53    }
54
55    @Override
56    public int compareTo(Object o) {
57      Cuenta c = (Cuenta)o;
58      if(this.id == c.id)
59        return 0;
60      else
61        return 1;
62    }
63  }
```

Como se puede observar en el nuevo método `delete()` hemos implementado los pasos indicados. Si probamos de nuevo el borrado podemos comprobar que funciona correctamente.

Hay diversas operaciones que no hemos implementado en este ejemplo, pero la vamos a dejar como ejercicio para el alumnado.

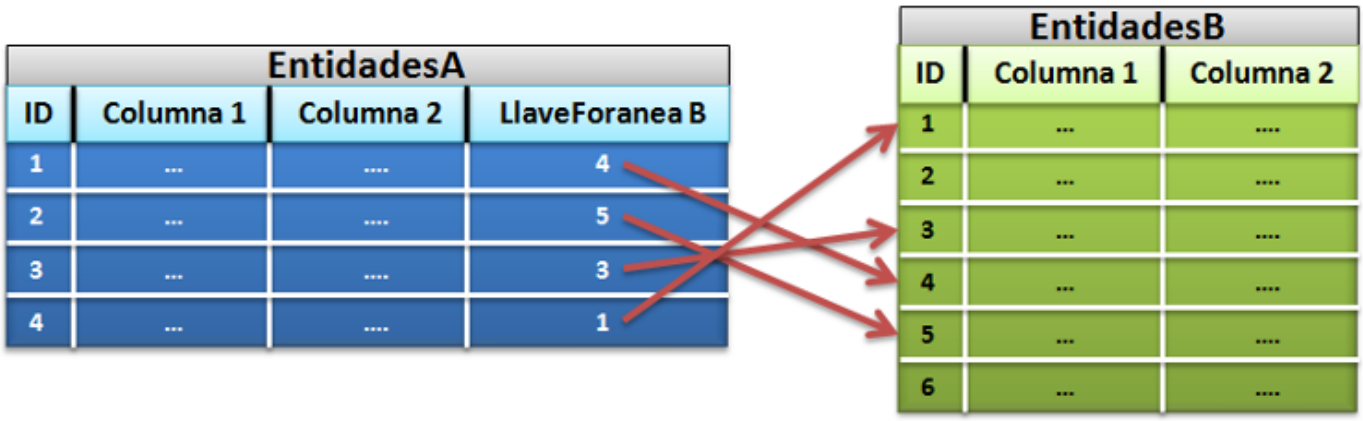
Ya solo nos queda implementar el siguiente tipo de relación que existe  y la variante que nos encontraremos de la misma.

# Relaciones N a N sin atributo en medio de la relación y sin mapear entidad del medio

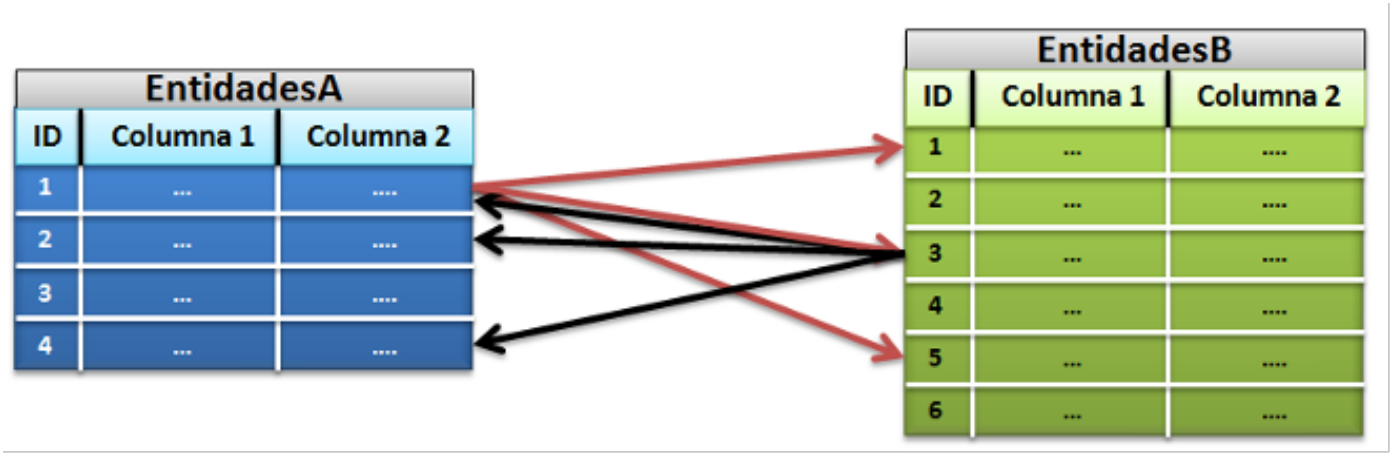
En ocasiones necesitamos relacionar muchas entidades de un tipo con muchas entidades de otro tipo. Este tipo de relaciones es muy común cuando trabajamos con bases de datos relacionales, por eso es importante saber cómo trabajar con ellas.

Este tipo de relaciones es especial ya que tenemos muchos registros de la entidad tipo A que están relacionadas con muchos registros de la entidad tipo B.

Normalmente los registros en base de datos se relacionan usando una llave foránea de una tabla con el identificador de otra tabla, como en la siguiente imagen:

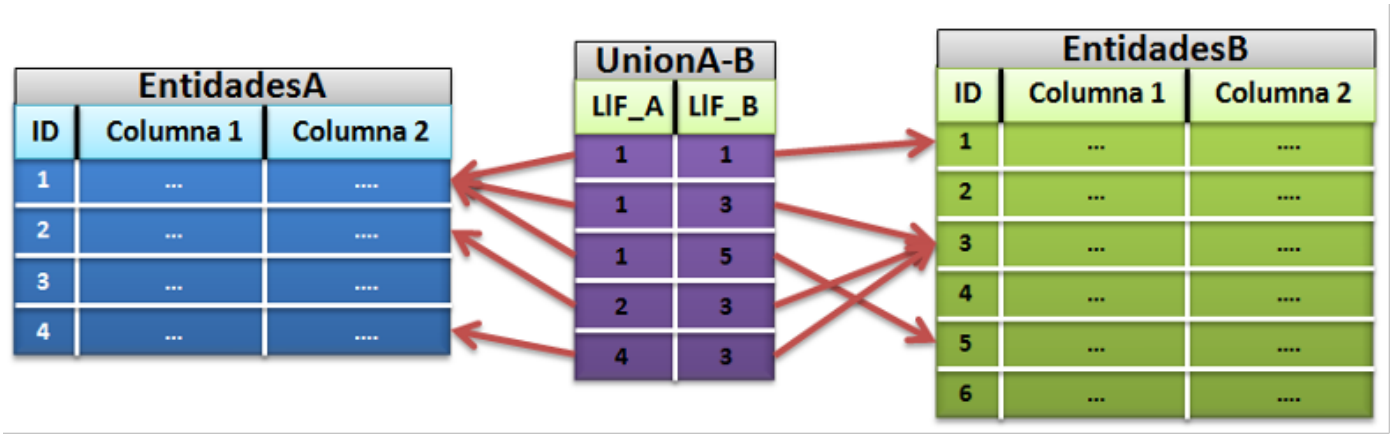



Sin embargo, como en las relaciones muchos a muchos necesitamos relacionar muchos registros de la entidad A, con muchos registros de la entidad B, y es por esto que NO nos basta con una llave foránea en una de las tablas ya que, por ejemplo, el registro con ID 1 de la entidad A podría estar relacionado con los registros con IDs 1, 3, y 5 de la entidad B (hasta aquí sería una relación uno a muchos). Sin embargo, el registro 3 de la entidad B puede estas relacionado con los registros 1, 2, y 4 de la entidad A, como lo muestra la siguiente figura:




Esto no puede ser logrado utilizando simplemente llaves foráneas en las tablas de las entidades A y B. En estos casos se utiliza una tercera tabla conocida como tabla de join, o tabla de enlace, o tabla de unión.

Esta tercera tabla lo único que hace es mantener las relaciones de las entidades A que están relacionadas con las entidades B, y las entidades B que están relacionadas con las entidades A. Como lo único que necesita para esto son los identificadores de ambas tablas como llaves foráneas, esto es lo que mantiene esta tabla de unión:

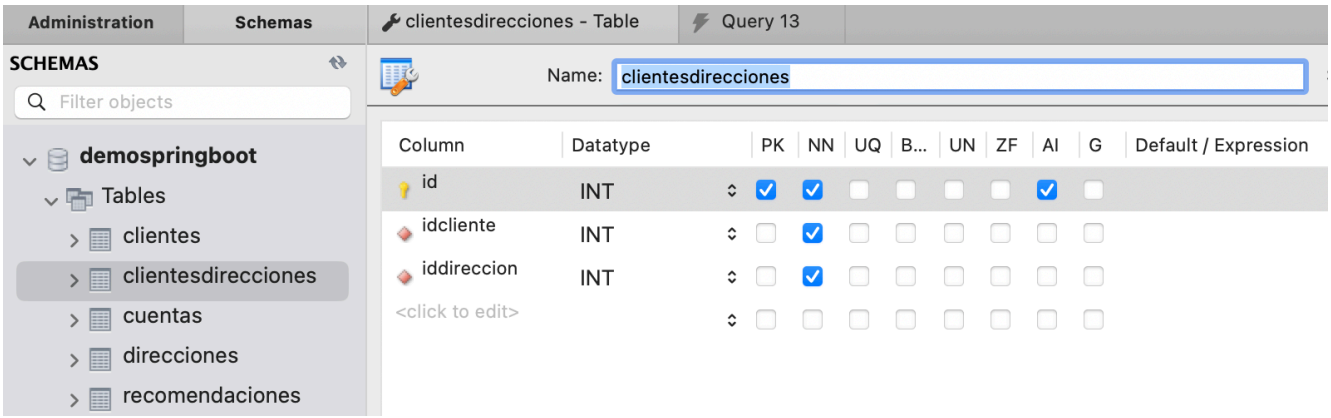


Afortunadamente con Hibernate no debemos preocuparnos de esta tabla de unión, ya que es el propio framework el que se encarga de hacerlo. De lo único que debemos preocuparnos es de hacer de forma correcta los mapeos para representar esta relación (ya que tiene algunos trucos).

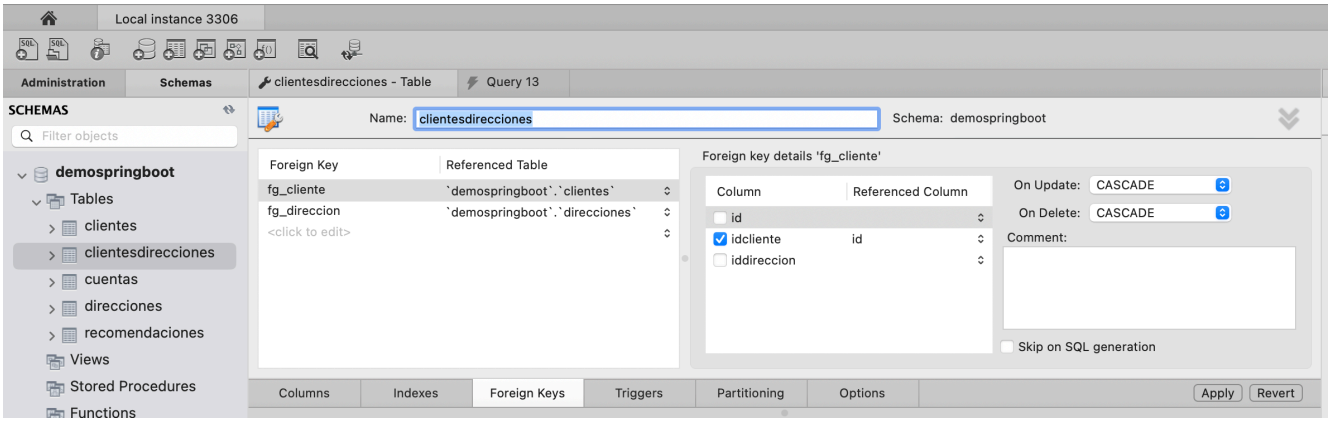
Lo primero que vamos a hacer es crear la estructura necesaria para soportar la relación `Clientes` y `Direcciones`. Por ello:

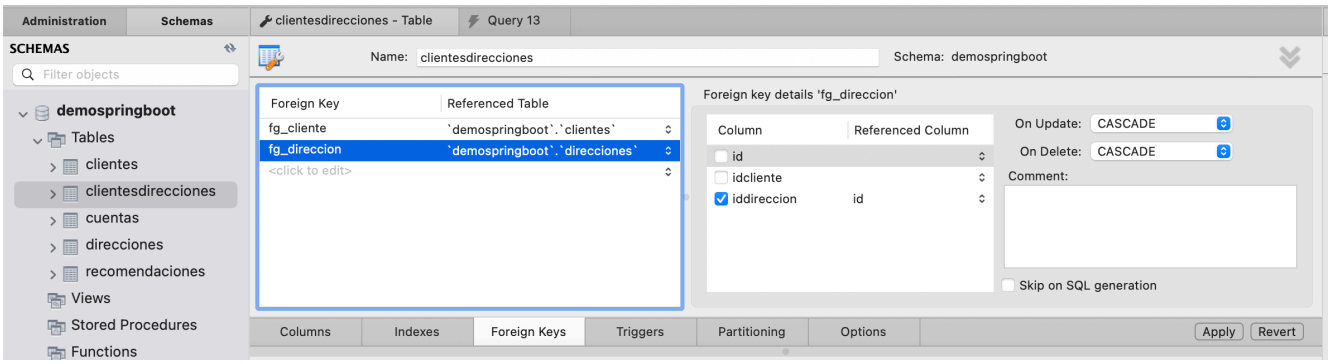
1. Abrimos MySQL WorkBench y nos conectamos con el usuario usuariodemo, que ha sido el que hemos creado.
2. Creamos la tabla `direcciones` que tendrá la siguiente estructura:

3. Ahora creamos la tabla `clientesdirecciones`, que se va a encargar de representar la relación N a N.

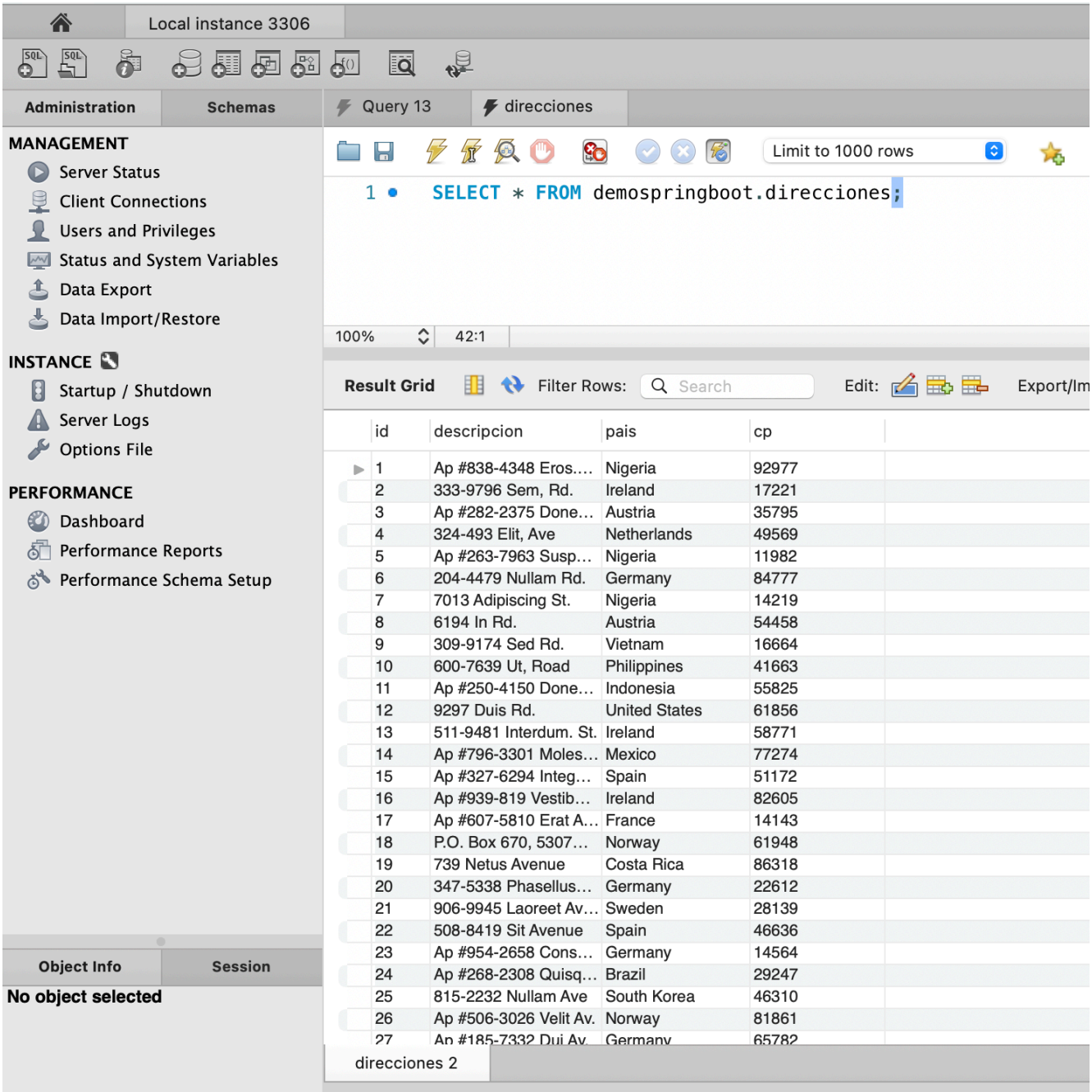


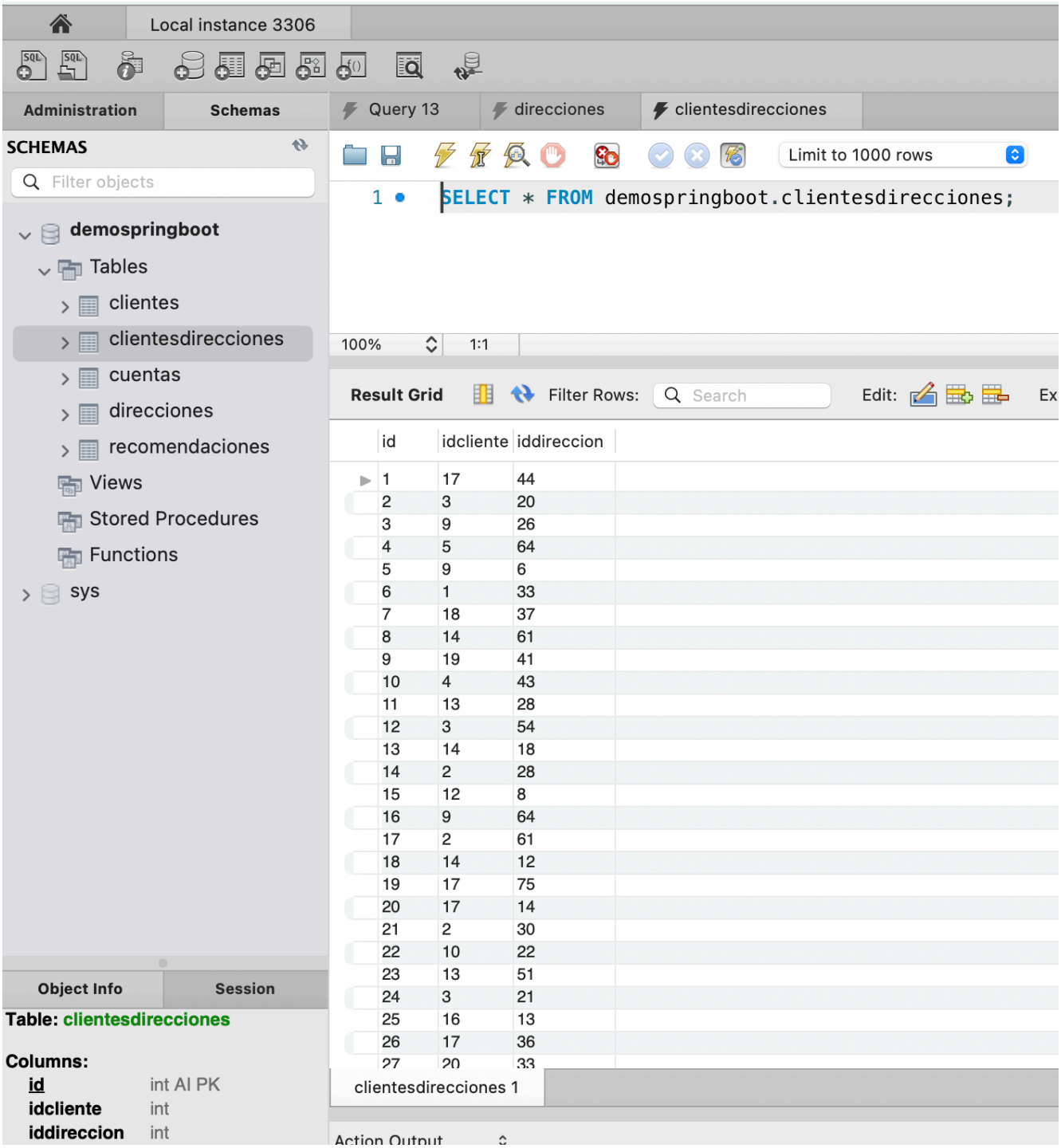

4. Creamos las claves ajenas respecto a la tabla `clientes` y a la tabla `direcciones`.






5. Tras esto, vamos a poblar la tabla `direcciones` con los datos que hay en el fichero `datosDirecciones.sql` y, a continuación, poblamos la tabla `clientesdirecciones` con los datos que hay en el fichero `datosClientesDirecciones.sql`:






6. Tras ello es el momento de comenzar a mapear.

Existen dos formas de poder dar solución a este problema:

- Convertir la relación Muchos a Muchos a relaciones 1 a Muchos. Cuando tenemos un atributo (o varios) en la tabla que representa la relación N a N esta es la solución correcta, es decir que la tabla intermedia contenga algún atributo más aparte del id y de las claves ajenas. Esto lo vamos a ver en el siguiente punto.
- Utilizar la anotación `@ManyToMany` como solución al problema, en caso de que no tengamos una columna en la tabla de relación N a N. Esta es la que vamos a explicar en este apartado.

Vamos a implementar la relación del segundo caso entre las entidades `Cliente.java` y `Direccion.java`. Tras ello explicaremos que uso de anotaciones hemos hecho. Comenzaremos implementando como debe ser la relación en la entidad Cliente.java:

```java
package com.example.demo.repository.entity;

import java.util.ArrayList;
import java.util.List;
import java.util.Objects;

import jakarta.persistence.CascadeType;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinTable;
import jakarta.persistence.ManyToMany;
import jakarta.persistence.OneToMany;
import jakarta.persistence.OneToOne;
import jakarta.persistence.Table;
import jakarta.persistence.JoinColumn;
import lombok.Data;
import lombok.ToString;

@Data
@Entity
@Table(name = "clientes")public class Cliente {

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

  @Column(name = "nif")
  private String nif;

  @Column(name = "nombre")
  private String nombre;

  private String apellidos;

  @Column(name = "claveseguridad")
  private String claveSeguridad;

  private String email;

  @OneToOne(fetch = FetchType.LAZY, cascade =  CascadeType.ALL, orphanRemoval = true, mappedBy = "cliente")
  @ToString.Exclude
  private Recomendacion recomendacion;

  @OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL, mappedBy = "cliente")
  private List<Cuenta> listaCuentas;

  @ManyToMany(fetch = FetchType.EAGER, cascade = { CascadeType.ALL })
```

```
52    // Tabla que mantiene la relacion N-N
53    @JoinTable(
54        // Nombre de la tabla
55        name = "clientesdirecciones",
56        // columna que almacena el id de cliente en la tabla clientesdirecciones
57        joinColumns = @JoinColumn(name = "idcliente"),
58        // columna que almacena el id de la direccion en la tabla clientesdirecciones
59        inverseJoinColumns = @JoinColumn(name = "iddireccion"))
60    @ToString.Exclude
61    private List<Direccion> listaDirecciones;
62
63    @Override
64    public boolean equals(Object obj) {
65      if (this == obj)
66        return true;
67      if (obj == null)
68        return false;
69      if (getClass() != obj.getClass())
70        return false;
71      Cliente other = (Cliente) obj;
72      return Objects.equals(id, other.id);
73    }
74
75    @Override
76    public int hashCode() {
77      return Objects.hash(id);
78    }
79
80    public Cliente() {
81      super();
82      this.recomendacion = new Recomendacion();
83      this.listaCuentas = new ArrayList<Cuenta>();
84      this.listaDirecciones = new ArrayList<Direccion>();
85    }
86  }
```

Pasaremos a crear una nueva entidad denominada `Direccion.java` y a implementarla.

```
1   package com.example.demo.repository.entity;
2
3   import java.util.ArrayList;
4   import java.util.List;
5   import java.util.Objects;
6
7   import jakarta.persistence.CascadeType;
8   import jakarta.persistence.Entity;
9   import jakarta.persistence.GeneratedValue;
10  import jakarta.persistence.GenerationType;
11  import jakarta.persistence.Id;
12  import jakarta.persistence.ManyToMany;
13  import jakarta.persistence.Table;
```

```java
import lombok.Data;
import lombok.ToString;

@Data
@Entity
@Table(name = "direcciones")
public class Direccion {

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

  private String descripcion;
  private String pais;
  private String cp;

  @ManyToMany(cascade = { CascadeType.ALL }, mappedBy = "listaDirecciones")
  @ToString.Exclude
  private List<Cliente> listaClientes;

  public Direccion() {
    super();
    this.listaClientes = new ArrayList<Cliente>();
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Direccion other = (Direccion) obj;
    return Objects.equals(id, other.id);
  }

  @Override
  public int hashCode() {
    return Objects.hash(id);
  }
}
```

Vamos a comenzar a explicar cada parte del código nuevo puesto aquí:

- `Cliente`: Aparece la anotación `@ManyToMany` en el atributo `listaDirecciones`, que es de tipo `List`. Fijaros que va acompañado de los parámetros `cascade` y `fetch`, que ya han sido explicados con anterioridad. También lleva la anotación `@JoinTable` la cual nos sirve para definir la estructura de la tabla intermedia que contendrá la relación entre los clientes y las direcciones. Lleva los siguientes parámetros:

- Con el atributo `name` de la anotación `@JoinTable` indicamos el nombre de la tabla de relación, que es `clientesdirecciones`.
- Con el atributo `joinColumns` definimos la clave foránea en la tabla `clientesdirecciones` a la tabla `clientes` usando el atributo `idcliente` de la tabla `clientesdirecciones`. Para ello usamos la anotación `@JoinColumn`.
- Con el atributo `inverseJoinColumns` definimos la clave foránea en la tabla `clientesdirecciones` a la tabla `direcciones` usando el atributo `iddireccion` de la tabla `clientesdirecciones`.
- Si alguna de las clases tuviera una clave compuesta, en estos atributos deberíamos especificar todas las columnas de dicha clave.
- `Direccion`: Aparece la anotación `@ManyToMany` en el atributo `listaClientes`, que es de tipo `List`. En este caso va acompañado solo de `cascade`, y sólo tenemos que anotar la relación con `@ManyToMany`, y utilizar el atributo `mappedBy` para indicar que la relación está definida en el atributo `listaDirecciones` de la clase `Clientes.java`.

Tras tener los mapeos realizados vamos a mapear los DTO's. Por ello crearemos `DireccionDTO.java`:

```java
package com.example.demo.model.dto;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import com.example.demo.repository.entity.Cliente;
import com.example.demo.repository.entity.Direccion;

import lombok.Data;
import lombok.ToString;

@Data
public class DireccionDTO implements Serializable {

  private static final long serialVersionUID = 1L;
  private Long id;
  private String descripcion;
  private String pais;
  private String cp;
  @ToString.Exclude
  private List<ClienteDTO> listaClientesDTO;

  public static DireccionDTO convertToDTO(Direccion direccion, ClienteDTO clienteDTO) {
    DireccionDTO direccionDTO = new DireccionDTO();
    direccionDTO.setId(direccion.getId());
    direccionDTO.setDescripcion(direccion.getDescripcion());
    direccionDTO.setPais(direccion.getPais());
    direccionDTO.setCp(direccion.getCp());

    // No tiene sentido mapear todos los clientes que tiene la direccion, puesto que
    // al mapear cada cliente volveriamos a mapear sus direcciones, y así
```

```
33      // sucesivamente.
34      direccionDTO.getListaClientesDTO().add(clienteDTO);
35
36      return direccionDTO;
37    }
38
39    public static Direccion convertToEntity(DireccionDTO direccionDTO, Cliente cliente)
  {
40      Direccion direccion = new Direccion();
41      direccion.setId(direccionDTO.getId());
42      direccion.setDescripcion(direccionDTO.getDescripcion());
43      direccion.setPais(direccionDTO.getPais());
44      direccion.setCp(direccionDTO.getCp());
45
46      // No tiene sentido mapear todos los clientes que tiene la direccion, puesto que
47      // al mapear cada cliente volveriamos a mapear sus direcciones, y así
48      // sucesivamente.
49      direccion.getListaClientes().add(cliente);
50
51      return direccion;
52
53    }
54
55    // Constructor vacio
56    public DireccionDTO() {
57      super();
58      this.listaClientesDTO = new ArrayList<ClienteDTO>();
59    }
60  }
```

Tras esto modificaremos `ClienteDTO.java`:

```
1  package com.example.demo.model.dto;
2
3  import java.io.Serializable;
4  import java.util.ArrayList;
5  import java.util.List;
6
7  import com.example.demo.repository.entity.Cliente;
8  import com.example.demo.repository.entity.Cuenta;
9  import com.example.demo.repository.entity.Direccion;
10 import com.example.demo.repository.entity.Recomendacion;
11
12 import lombok.Data;
13 import lombok.ToString;
14
15 @Data
16 public class ClienteDTO implements Serializable{
17
18   private static final long serialVersionUID = 1L;
19   private Long id;
```

```java
  private String nif;
  private String nombre;
  private String apellidos;
  private String claveSeguridad;
  private String email;
  @ToString.Exclude
  private RecomendacionDTO recomendacionDTO;
  @ToString.Exclude
  private List<CuentaDTO> listaCuentasDTO;
  @ToString.Exclude
  private List<DireccionDTO> listaDireccionesDTO;

  // Convertir una entidad a un DTO
  public static ClienteDTO convertToDTO(Cliente cliente) {

    ClienteDTO clienteDTO = new ClienteDTO();
    clienteDTO.setId(cliente.getId());
    clienteDTO.setNif(cliente.getNif());
    clienteDTO.setNombre(cliente.getNombre());
    clienteDTO.setApellidos(cliente.getApellidos());
    clienteDTO.setClaveSeguridad(cliente.getClaveSeguridad());
    clienteDTO.setEmail(cliente.getEmail());

    RecomendacionDTO rec = RecomendacionDTO.converToDTO(cliente.getRecomendacion(), clienteDTO);
    clienteDTO.setRecomendacionDTO(rec);
    // Cargamos la lista de cuentas
    for(int i=0;i<cliente.getListaCuentas().size();i++) {
      CuentaDTO cuentadto = CuentaDTO.convertToDTO(cliente.getListaCuentas().get(i), clienteDTO);
      clienteDTO.getListaCuentasDTO().add(cuentadto);
    }
    // Cargamos la lista de direcciones
    for(int i=0;i<cliente.getListaDirecciones().size();i++) {
      DireccionDTO direccindto = DireccionDTO.convertToDTO(cliente.getListaDirecciones().get(i), clienteDTO);
      clienteDTO.getListaDireccionesDTO().add(direccindto);
    }

    return clienteDTO;
  }

  // Convertir un DTO a una entidad
  public static Cliente convertToEntity(ClienteDTO clienteDTO) {

    Cliente cliente = new Cliente();
    cliente.setId(clienteDTO.getId());
    cliente.setNif(clienteDTO.getNif());
    cliente.setNombre(clienteDTO.getNombre());
    cliente.setApellidos(clienteDTO.getApellidos());
    cliente.setClaveSeguridad(clienteDTO.getClaveSeguridad());
    cliente.setEmail(clienteDTO.getEmail());
```

```

    Recomendacion rec = RecomendacionDTO.convertToEntity(clienteDTO.getRecomendacionDTO(), cliente);
    cliente.setRecomendacion(rec);

    // Cargamos la lista de cuentas
    for(int i=0;i<clienteDTO.getListaCuentasDTO().size();i++) {
      Cuenta cuenta = CuentaDTO.convertToEntity(clienteDTO.getListaCuentasDTO().get(i));
      cliente.getListaCuentas().add(cuenta);
    }
    // Cargamos la lista de direcciones
    for(int i=0;i<clienteDTO.getListaDireccionesDTO().size();i++) {
      Direccion direccion = DireccionDTO.convertToEntity(clienteDTO.getListaDireccionesDTO().get(i), cliente);
      cliente.getListaDirecciones().add(direccion);
    }

    return cliente;
  }

  // Constructor vacio
  public ClienteDTO() {
    super();
    this.recomendacionDTO = new RecomendacionDTO();
    this.listaCuentasDTO = new ArrayList<CuentaDTO>();
    this.listaDireccionesDTO = new ArrayList<DireccionDTO>();
  }
}
```

Lo único a destacar en estos DTO's es que, a la hora de mapear las direcciones solo nos traeremos un solo cliente, y no mapearemos todos los clientes que contiene la dirección, así no tendremos el problema de la referencia circular que produce un error de desbordamiento de pila.

Ahora crearemos un controlador que se va a encargar de la gestión de las direcciones. A este controlador le llamamos `DireccionController.java`, y vamos a implementar la operación de listar las direcciones y de dar de alta una nueva dirección asociada a un cliente. El controlador quedará de la siguiente forma:

```
package com.example.demo.web.controller;

import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.servlet.ModelAndView;

```

```java
import com.example.demo.model.dto.ClienteDTO;
import com.example.demo.model.dto.DireccionDTO;
import com.example.demo.service.ClienteService;
import com.example.demo.service.DireccionService;

@Controller
public class DireccionController {

  private static final Logger log = LoggerFactory.getLogger(DireccionController.class);

  @Autowired
  private DireccionService direccionService;

  @Autowired
  private ClienteService clienteService;

  @GetMapping("/clientes/{idCliente}/direcciones")
  public ModelAndView findAllByCliente(@PathVariable Long idCliente) {

    log.info("DireccionController - findAllByCliente: Listamos las direcciones del cliente: " + idCliente);

    // Obtenemos el cliente para luego poner sus datos en la pantalla
    ClienteDTO clienteDTO = new ClienteDTO();
    clienteDTO.setId(idCliente);
    clienteDTO = clienteService.findById(clienteDTO);
    // Obtenemos la lista de direcciones del cliente
    List<DireccionDTO> listaDireccionesDTO = direccionService.findAllByCliente(clienteDTO);
    // Pasamos los datos al modelo
    ModelAndView mav = new ModelAndView("direcciones");
    mav.addObject("listaDireccionesDTO", listaDireccionesDTO);
    mav.addObject("clienteDTO", clienteDTO);
    // retornamos el ModelAndView
    return mav;
  }

  @GetMapping("/clientes/{idCliente}/direcciones/add")
  public ModelAndView add(@PathVariable Long idCliente) {

    log.info("DireccionController - add: Anyadimos una nueva direccion al cliente: " + idCliente);

    ClienteDTO clienteDTO = new ClienteDTO();
    clienteDTO.setId(idCliente);
    clienteDTO = clienteService.findById(clienteDTO);
    // Pasamos los datos al modelo
    ModelAndView mav = new ModelAndView("direccionesform");
    mav.addObject("direccionDTO", new DireccionDTO());
    mav.addObject("clienteDTO", clienteDTO);
    // retornamos el ModelAndView
```

```
    return mav;
  }

  @GetMapping("/clientes/{idCliente}/direcciones/save")
  public ModelAndView save(@PathVariable Long idCliente, @ModelAttribute("direccionDTO") DireccionDTO direccionDTO) {

    log.info("DireccionController - save: Almacenamos la direccion " + direccionDTO.toString());

    ClienteDTO clienteDTO = new ClienteDTO();
    clienteDTO.setId(idCliente);
    clienteDTO = clienteService.findById(clienteDTO);
    // Anyadimos al cliente la direccion. Puede haber varias direcciones ya en el
    // clienteDTO
    clienteDTO.getListaDireccionesDTO().add(direccionDTO);
    // Anyadimos a la direccion el cliente. En este momento solo habra un clienteDTO
    direccionDTO.getListaClientesDTO().add(clienteDTO);
    direccionService.save(direccionDTO);
    // Redireccionamos para volver a invocar el metodo que escucha /clientes
    ModelAndView mav = new ModelAndView("redirect:/clientes/{idCliente}/direcciones");
    return mav;
  }
}
```

Tras esto creamos el servicio, que se llamará el interfaz `DireccionService.java` y `DireccionServiceImpl.java`:

```
package com.example.demo.service;

import java.util.List;

import com.example.demo.model.dto.ClienteDTO;
import com.example.demo.model.dto.DireccionDTO;

public interface DireccionService {

  public List<DireccionDTO> findAllByCliente(ClienteDTO clienteDTO);
  public void save(DireccionDTO direccionDTO);

}
```

```
package com.example.demo.service;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.example.demo.model.dto.ClienteDTO;
import com.example.demo.model.dto.DireccionDTO;
import com.example.demo.repository.dao.ClienteRepository;
import com.example.demo.repository.dao.DireccionRepository;
import com.example.demo.repository.entity.Cliente;
import com.example.demo.repository.entity.Direccion;

@Service
public class DireccionServiceImpl implements DireccionService {

  private static final Logger log = LoggerFactory.getLogger(DireccionServiceImpl.class);

  @Autowired
  private DireccionRepository direccionRepository;

  @Autowired
  private ClienteRepository clienteRepository;

  @Override
  public List<DireccionDTO> findAllByCliente(ClienteDTO clienteDTO) {

    log.info("DireccionServiceImpl - findAllByCliente: Lista de todas las direcciones del cliente: "
        + clienteDTO.getId());

    // Obtenemos la lista de direcciones del cliente
    List<Direccion> lista = (List<Direccion>) direccionRepository.findAllByCliente(clienteDTO.getId());
    // Creamos una lista de DireccionDTO que será la que devolvamos al controlador
    List<DireccionDTO> listaDireccionesDTO = new ArrayList<DireccionDTO>();
    // Recorremos la lista de direcciones y las mapeamos a DTO
    for (int i = 0; i < lista.size(); ++i) {
      listaDireccionesDTO.add(DireccionDTO.convertToDTO(lista.get(i), clienteDTO));
    }
    // Devolvemos la lista de DTO's
    return listaDireccionesDTO;
  }

  @Override
  public void save(DireccionDTO direccionDTO) {

    log.info("DireccionServiceImpl - save: Salva la direccion del cliente: "
        + direccionDTO.getListaClientesDTO().get(0).getId());

    // Solo hay un clienteDTO en la direccionDTO
    Optional<Cliente> cliente = clienteRepository.findById(direccionDTO.getListaClientesDTO().get(0).getId());
    if (cliente.isPresent()) {
```

```
57        // Creamos las relaciones en las entidades
58        Direccion direccion = DireccionDTO.convertToEntity(direccionDTO,
cliente.get());
59        cliente.get().getListaDirecciones().add(direccion);
60        // Almacenamos la direccion, y por la relacion que tiene, almacenará la
relacion N a N
61        direccionRepository.save(direccion);
62      }
63    }
64  }
```

Ya solo nos queda crear el repositorio, por lo que definiremos el interfaz `DireccionRepository.java`. En este definiremos el método `findAllByCliente` donde obtendremos las direcciones de un cliente en concreto a través de la tabla que define la relación N a N:

```
1  package com.example.demo.repository.dao;
2
3  import java.util.List;
4
5  import org.springframework.data.jpa.repository.JpaRepository;
6  import org.springframework.data.jpa.repository.Query;
7  import org.springframework.data.repository.query.Param;
8  import org.springframework.stereotype.Repository;
9  import com.example.demo.repository.entity.Direccion;
10
11 import jakarta.transaction.Transactional;
12
13 @Repository
14 @Transactional
15 public interface DireccionRepository extends JpaRepository<Direccion, Long>{
16
17   @Query(value = "SELECT A.* FROM DIRECCIONES A, CLIENTESDIRECCIONES B "
18       + " WHERE A.ID=B.IDDIRECCION AND B.IDCLIENTE = :idcliente", nativeQuery = true)
19   public List<Direccion> findAllByCliente(@Param("idcliente") Long idCliente);
20 }
```

La vista `clientes.html` ya tiene la opción para acceder a las direcciones, por lo que no es necesario implementar nada.

Tras esto vamos a crear la vista `direcciones.html`, que va a mostrar las direcciones de un cliente en concreto. En dicha vista vamos a permitir dar de alta una nueva dirección asociada al cliente (ya tenemos definida esta operación) y después implementaremos la opción de añadir una dirección ya existente al cliente. El código quedará de la siguiente forma:

```
1  <!DOCTYPE html>
2  <html xmlns:th="http://www.thymeleaf.org">
3  <head>
4    <meta charset="UTF-8">
5    <title>Mantenimiento de direcciones</title>
6  </head>
```

```
<body>
  <div>
    <br>
    <h2 th:if="${listaDireccionesDTO.isEmpty()}">No hay direcciones para el cliente
      <td th:utext="${clienteDTO.nombre}"></td> <td th:utext="${clienteDTO.apellidos}"></td>
    </h2>
    <div th:if="${!listaDireccionesDTO.isEmpty()}">
      <h2>Mantenimiento de direcciones del cliente
        <td th:utext="${clienteDTO.nombre}"></td> <td th:utext="${clienteDTO.apellidos}"></td>
      </h2>
      <br>
      <table border="1">
        <thead>
          <tr>
            <th>id</th>
            <th>Direccion</th>
            <th>País</th>
            <th>CP</th>
          </tr>
        </thead>
        <tbody>
          <tr th:each="direccionDTO : ${listaDireccionesDTO}">
            <td th:utext="${direccionDTO.id}">...</td>
            <td th:utext="${direccionDTO.descripcion}">...</td>
            <td th:utext="${direccionDTO.pais}">...</td>
            <td th:utext="${direccionDTO.cp}">...</td>
          </tr>
        </tbody>
      </table>
    </div>
    <br>
    <div>
      <a th:href="@{/clientes/{idCliente}/direcciones/add(idCliente=${clienteDTO.id})}" >Nueva direccion</a>
      </br>
      <a th:href="@{/clientes/{idCliente}/direcciones/addselect(idCliente=${clienteDTO.id})}" >Añadir dirección ya existente</a>
      </br>
      <a th:href="@{/clientes}">Volver a la lista de clientes</a>
    </div>
  </div>
</body>
</html>

```

Ya solo nos queda definir el formulario que permite dar de alta la nueva dirección, que sería `direccionform.html`. Quedará de la siguiente forma:

```
<!DOCTYPE HTML>
```

```html
<html xmlns:th="http://www.thymeleaf.org">

<head>
    <meta charset="UTF-8" />
    <title>Crear nueva direccion</title>
 </head>

<body>
    <h1>Crear nueva direccion</h1>
    <br/>
    <br/>
    <form th:action="@{/clientes/{idCliente}/direcciones/save
(idCliente=${clienteDTO.id})}" th:object="${direccionDTO}" method="POST">
        <table border="0">
            <tr th:if="${direccionDTO.id}">
                <td>ID</td>
                <td>:</td>
                <td>
                  <input type="text" th:field="*{id}" readonly />
                </td>
            </tr>
            <tr>
                <td>Direccion</td>
                <td>:</td>
                <td>
                    <input type="text" th:field="*{descripcion}" />
                </td>
            </tr>
            <tr>
                <td>País</td>
                <td>:</td>
                <td>
                    <input type="text" th:field="*{pais}" />
                </td>
            </tr>
            <tr>
                <td>Código postal</td>
                <td>:</td>
                <td>
                    <input type="text" th:field="*{cp}" />
                </td>
            </tr>
        </table>
        <input type="submit" th:value="Crear" /> </form>
    <br/>
    <br/>
    <a th:href="@{/clientes/{idCliente}/direcciones
(idCliente=${clienteDTO.id})}">Volver a la lista de direcciones</a>
</html>
```

Si probamos la aplicación obtenemos los siguientes resultados:

Mantenimiento de clientes

localhost:8888/clientes

# Mantenimiento de clientes

| id | Nombre | Apellidos | NIF | Correo electrónico | Editar | Eliminar | **Cuentas** | **Direcciones** |
|---|---|---|---|---|---|---|---|---|
| 1 | Keiko | Meadows | 44999594K | meadowskeiko6933@icloud.net | Editar | Eliminar | Cuentas | Direcciones |
| 2 | Whoopi | Phelps | 28553744Y | w-phelps@aol.ca | Editar | Eliminar | Cuentas | Direcciones |
| 3 | Hilary | Mcmahon | 25861374E | hmcmahon6589@google.org | Editar | Eliminar | Cuentas | Direcciones |
| 4 | Hunter | Frederick | 12242423G | h_frederick@icloud.edu | Editar | Eliminar | Cuentas | Direcciones |
| 5 | Diana | Dickerson | 51724739D | d-dickerson5965@aol.net | Editar | Eliminar | Cuentas | Direcciones |
| 6 | Amber | Ramirez | 13176696D | amber.ramirez6505@protonmail.org | Editar | Eliminar | Cuentas | Direcciones |
| 7 | Chava | Floyd | 82568848H | cfloyd@hotmail.couk | Editar | Eliminar | Cuentas | Direcciones |
| 8 | Tatum | Walsh | 03768520L | walsh_tatum2491@outlook.couk | Editar | Eliminar | Cuentas | Direcciones |
| 9 | Ezekiel | Woodard | 51976869H | e-woodard@icloud.couk | Editar | Eliminar | Cuentas | Direcciones |
| 10 | Coby | Lester | 55172493N | c-lester7681@hotmail.couk | Editar | Eliminar | Cuentas | Direcciones |
| 11 | Brielle | Howard | 48037186R | b_howard@aol.com | Editar | Eliminar | Cuentas | Direcciones |
| 12 | Candace | Bates | 71227610F | c-bates@google.org | Editar | Eliminar | Cuentas | Direcciones |
| 13 | Eagan | Carpenter | 41124626J | e_carpenter9768@outlook.org | Editar | Eliminar | Cuentas | Direcciones |
| 14 | Yvette | Slater | 86926685Y | yslater@google.edu | Editar | Eliminar | Cuentas | Direcciones |
| 15 | Heather | Abbott | 18829162J | abbottheather@hotmail.org | Editar | Eliminar | Cuentas | Direcciones |
| 16 | Kessie | Rich | 10159358C | k_rich804@google.net | Editar | Eliminar | Cuentas | Direcciones |
| 17 | Keelie | Forbes | 61816073X | k-forbes1767@hotmail.com | Editar | Eliminar | Cuentas | Direcciones |
| 18 | Dale | Glass | 27682739N | gdale@google.ca | Editar | Eliminar | Cuentas | Direcciones |
| 19 | Farrah | Farmer | 91124511M | farrah-farmer6033@hotmail.com | Editar | Eliminar | Cuentas | Direcciones |
| 20 | Graham | Casey | 21865746E | g.casey3482@google.com | Editar | Eliminar | Cuentas | Direcciones |

Nuevo cliente
Volver al inicio

Seleccionamos el cliente con id 4, que es Hunter Frederick:

Mantenimiento de direcciones

localhost:8888/clientes/4/direcciones

# Mantenimiento de direcciones del cliente Hunter Frederick

| id | Direccion | País | CP |
|---|---|---|---|
| 7 | 7013 Adipiscing St. | Nigeria | 14219 |
| 24 | Ap #268-2308 Quisque St. | Brazil | 29247 |
| 43 | Ap #270-8384 Mollis St. | India | 17133 |
| 70 | Ap #382-5932 Gravida. Avenue | Turkey | 79545 |
| 71 | 4818 Sapien Street | Netherlands | 42226 |

Nueva direccion
Añadir dirección ya existente
Volver a la lista de direcciones

Ahora pulsamos añadir una nueva dirección:

Crear nueva direccion

localhost:8888/clientes/4/direcciones/add

# Crear nueva direccion

Direccion : Camino al monte 27
País : España
Código postal : 43564
Crear

Volver a la lista de direcciones

Pulsamos el botón de Crear y comprobamos que se ha añadido correctamente:

Podemos ver que la ha creado perfectamente, pero si comprobamos la tabla en la base de datos podemos ver lo siguiente:

Local instance 3306

Administration | Schemas | Query 13 | clientesdirecciones

SCHEMAS

Filter objects

- demospringboot
  - Tables
    - clientes
    - clientesdirecciones
    - cuentas
    - direcciones
    - recomendaciones
  - Views
  - Stored Procedures
  - Functions
- sys

Limit to 1000 rows

```
1 • SELECT * FROM demospringboot.clientesdirecciones;
```

100% 1:1

Result Grid | Filter Rows: Search | Edit: | Export/Import:

| id | idcliente | iddireccion |
|---|---|---|
| 129 | 11 | 53 |
| 130 | 8 | 25 |
| 131 | 2 | 79 |
| 132 | 5 | 13 |
| 133 | 19 | 77 |
| 134 | 10 | 5 |
| 135 | 8 | 30 |
| 136 | 10 | 69 |
| 137 | 17 | 17 |
| 138 | 15 | 78 |
| 139 | 2 | 46 |
| 140 | 4 | 7 |
| 141 | 4 | 24 |
| 142 | 4 | 43 |
| 143 | 4 | 70 |
| 144 | 4 | 71 |
| 145 | 4 | 93 |
| NULL | NULL | NULL |

clientesdirecciones 1

Podemos ver que ha creado 6 registros nuevos, desde el 140 al 145. El problema que vemos en este desarrollo es que cada vez que añadimos una nueva dirección borra las anteriores que pertenecen a este cliente y vuelve a añadirlas de nuevo.

Si comprobamos el log de Hibernate en STS podemos ver lo siguiente:

Console | Problems | Debug Shell

```
demo6 copia - Demo6Application [Spring Boot App] /Library/Java/JavaVirtualMachines/jdk-17.0.4.1.jdk/Contents/Home/bin/java  (13 dic 2022 17:07:15) [pid: 17842]
clientes c1_0 left join cuentas l1_0 on c1_0.id=l1_0.idcliente where c1_0.id=?
Hibernate: select r1_0.id,r1_0.idcliente,r1_0.observaciones from recomendaciones r1_0 where r1_0.idcliente=?
Hibernate: select l1_0.idcliente,l1_1.id,l1_1.cp,l1_1.descripcion,l1_1.pais from clientesdirecciones l1_0 join direcciones l1_1 on l1_1.id=l1_0.iddireccion where l1_0.idcliente=?
2022-12-13T17:08:53.501+01:00  INFO 17842 --- [nio-8888-exec-6] c.e.d.w.controller.DireccionController    : DireccionController - save: Almacenamos la direccion DireccionDTO(id=null,
descripcion=Camino al monte 27, pais=España, cp=43564)
2022-12-13T17:08:53.501+01:00  INFO 17842 --- [nio-8888-exec-6] c.e.demo.service.ClienteServiceImpl       : ClienteServiceImpl - findById: Buscar cliente por id: 4
Hibernate: select c1_0.id,c1_0.apellidos,c1_0.claveseguridad,c1_0.email,l1_0.idcliente,l1_0.id,l1_0.banco,l1_0.dc,l1_0.numerocuenta,l1_0.saldoactual,l1_0.sucursal,c1_0.nif,c1_0.nombre from
clientes c1_0 left join cuentas l1_0 on c1_0.id=l1_0.idcliente where c1_0.id=?
Hibernate: select r1_0.id,r1_0.idcliente,r1_0.observaciones from recomendaciones r1_0 where r1_0.idcliente=?
Hibernate: select l1_0.idcliente,l1_1.id,l1_1.cp,l1_1.descripcion,l1_1.pais from clientesdirecciones l1_0 join direcciones l1_1 on l1_1.id=l1_0.iddireccion where l1_0.idcliente=?
2022-12-13T17:08:53.506+01:00  INFO 17842 --- [nio-8888-exec-6] c.e.demo.service.DireccionServiceImpl     : DireccionServiceImpl - save: Salva la direccion del cliente: 4
Hibernate: insert into direcciones (cp, descripcion, pais) values (?, ?, ?)
Hibernate: delete from clientesdirecciones where idcliente=?
Hibernate: insert into clientesdirecciones (idcliente, iddireccion) values (?, ?)
Hibernate: insert into clientesdirecciones (idcliente, iddireccion) values (?, ?)
Hibernate: insert into clientesdirecciones (idcliente, iddireccion) values (?, ?)
Hibernate: insert into clientesdirecciones (idcliente, iddireccion) values (?, ?)
Hibernate: insert into clientesdirecciones (idcliente, iddireccion) values (?, ?)
Hibernate: insert into clientesdirecciones (idcliente, iddireccion) values (?, ?)
2022-12-13T17:08:53.535+01:00  INFO 17842 --- [nio-8888-exec-7] c.e.d.w.controller.DireccionController    : DireccionController - findAllByCliente: Listamos las direcciones del cliente: 4
2022-12-13T17:08:53.535+01:00  INFO 17842 --- [nio-8888-exec-7] c.e.demo.service.ClienteServiceImpl       : ClienteServiceImpl - findById: Buscar cliente por id: 4
Hibernate: select c1_0.id,c1_0.apellidos,c1_0.claveseguridad,c1_0.email,l1_0.idcliente,l1_0.id,l1_0.banco,l1_0.dc,l1_0.numerocuenta,l1_0.saldoactual,l1_0.sucursal,c1_0.nif,c1_0.nombre from
clientes c1_0 left join cuentas l1_0 on c1_0.id=l1_0.idcliente where c1_0.id=?
Hibernate: select r1_0.id,r1_0.idcliente,r1_0.observaciones from recomendaciones r1_0 where r1_0.idcliente=?
Hibernate: select l1_0.idcliente,l1_1.id,l1_1.cp,l1_1.descripcion,l1_1.pais from clientesdirecciones l1_0 join direcciones l1_1 on l1_1.id=l1_0.iddireccion where l1_0.idcliente=?
2022-12-13T17:08:53.541+01:00  INFO 17842 --- [nio-8888-exec-7] c.e.demo.service.DireccionServiceImpl     : DireccionServiceImpl - findAllByCliente: Lista de todas las direcciones del cliente: 4
Hibernate: SELECT A.* FROM DIRECCIONES A, CLIENTESDIRECCIONES B  WHERE A.ID=B.IDDIRECCION AND B.IDCLIENTE = ?
```

El proceso que hace es el siguiente:

- Inserta la nueva dirección.
- Borra todas las relaciones que hay almacenadas en la tabla `clientesdirecciones` que sean del cliente con id 4.
- Tras ello vuelve a insertar todas las relaciones entre las direcciones y el cliente con id 4 que debe haber en la tabla `clientesdirecciones`.

Esto hace que exista un problema de rendimiento a la hora de utilizar la colección `List` para mapear las listas en una relación de muchos a muchos. Por ello vamos a cambiar la colecciones de tipo `List` por colecciones de tipo `Set`. Tendremos que modificar las clases `Cliente.java`, `Direccion.java` y `ClienteDTO.java`. El código de la clase `Cliente.java` será el siguiente:

```
package com.example.demo.repository.entity;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Objects;
import java.util.Set;

import jakarta.persistence.CascadeType;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinTable;
import jakarta.persistence.ManyToMany;
import jakarta.persistence.OneToMany;
import jakarta.persistence.OneToOne;
import jakarta.persistence.Table;
import jakarta.persistence.JoinColumn;
import lombok.Data;
import lombok.ToString;

@Data
@Entity
@Table(name = "clientes")
public class Cliente {

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

  @Column(name = "nif")
  private String nif;

  @Column(name = "nombre")
  private String nombre;

  private String apellidos;

  @Column(name = "claveseguridad")
  private String claveSeguridad;

  private String email;

```

```java
  @OneToOne(fetch = FetchType.EAGER, cascade = CascadeType.ALL, orphanRemoval = true, mappedBy = "cliente")
  @ToString.Exclude
  private Recomendacion recomendacion;

  /*
   * Mapeo de la lista de cuentas con List
  @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL, mappedBy = "cliente")
  @ToString.Exclude
  private List<Cuenta> listaCuentas;
    */

  // Mapeo de la lista de cuentas con Set
  @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL, mappedBy = "cliente")
  @ToString.Exclude
  private Set<Cuenta> listaCuentas;

  /*
  @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
  // Tabla que mantiene la relacion N-N
  @JoinTable(
      // Nombre de la tabla
      name = "clientesdirecciones",
      // columna que almacena el id de cliente en la tabla clientesdirecciones
      joinColumns = @JoinColumn(name = "idcliente"),
      // columna que almacena el id de la direccion en la tabla clientesdirecciones
      inverseJoinColumns = @JoinColumn(name = "iddireccion"))
  @ToString.Exclude
  private List<Direccion> listaDirecciones;
  */

  @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
  // Tabla que mantiene la relacion N-N
  @JoinTable(
      // Nombre de la tabla
      name = "clientesdirecciones",
      // columna que almacena el id de cliente en la tabla clientesdirecciones
      joinColumns = @JoinColumn(name = "idcliente"),
      // columna que almacena el id de la direccion en la tabla clientesdirecciones
      inverseJoinColumns = @JoinColumn(name = "iddireccion"))
  @ToString.Exclude
  private Set<Direccion> listaDirecciones;

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
```

```
      return false;
    Cliente other = (Cliente) obj;
    return Objects.equals(id, other.id);
  }

  @Override
  public int hashCode() {
    return Objects.hash(id);
  }

  public Cliente() {
    super();
    this.recomendacion = new Recomendacion();
    //this.listaDirecciones = new ArrayList<Direccion>();
    //this.listaCuentas = new ArrayList<Cuenta>();
    this.listaCuentas = new HashSet<Cuenta>();
    this.listaDirecciones = new HashSet<Direccion>();
  }
}
```

El código de la clase `Direccion.java` será el siguiente:

```
package com.example.demo.repository.entity;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Objects;
import java.util.Set;

import jakarta.persistence.CascadeType;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.ManyToMany;
import jakarta.persistence.Table;
import lombok.Data;
import lombok.ToString;

@Data
@Entity
@Table(name = "direcciones")
public class Direccion {

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

  private String descripcion;
```

```
29    private String pais;
30    private String cp;
31
32    /*
33    @ManyToMany(cascade = { CascadeType.ALL }, mappedBy = "listaDirecciones")
34    @ToString.Exclude
35    private List<Cliente> listaClientes;
36    */
37
38    @ManyToMany(cascade = { CascadeType.ALL }, mappedBy = "listaDirecciones")
39    @ToString.Exclude
40    private Set<Cliente> listaClientes;
41
42
43    public Direccion() {
44      super();
45      //this.listaClientes = new ArrayList<Cliente>();
46      this.listaClientes = new HashSet<Cliente>();
47    }
48
49    @Override
50    public boolean equals(Object obj) {
51      if (this == obj)
52        return true;
53      if (obj == null)
54        return false;
55      if (getClass() != obj.getClass())
56        return false;
57      Direccion other = (Direccion) obj;
58      return Objects.equals(id, other.id);
59    }
60
61    @Override
62    public int hashCode() {
63      return Objects.hash(id);
64    }
65  }
```

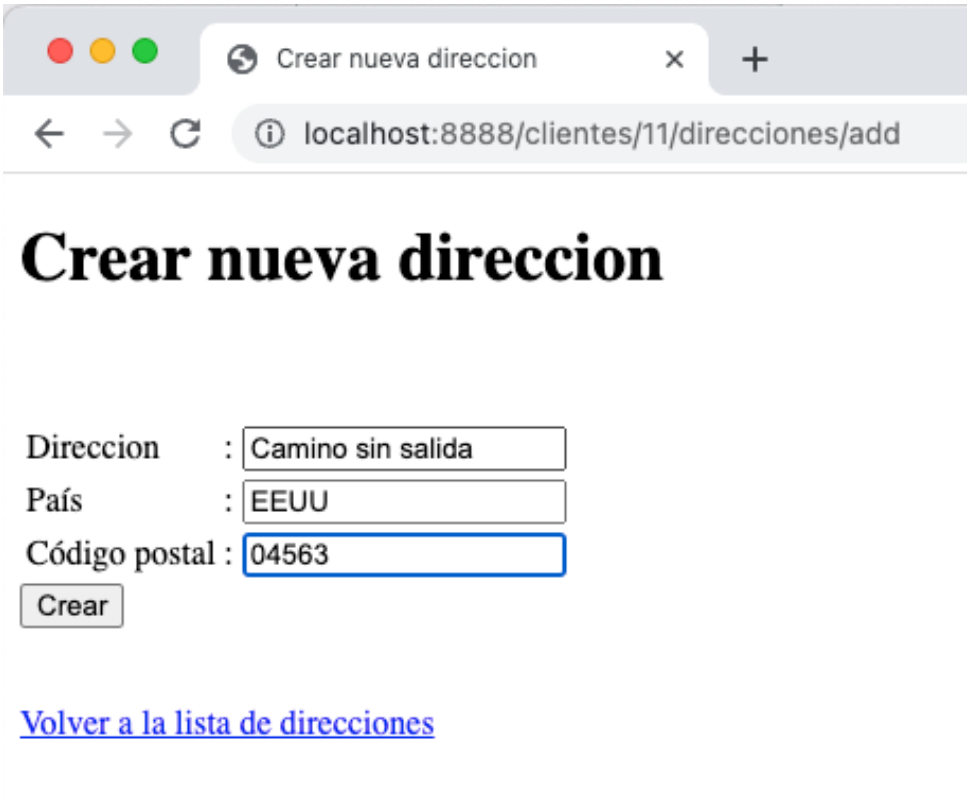
El código de la clase `ClienteDTO.java` será el siguiente:

```
1   package com.example.demo.model.dto;
2
3   import java.io.Serializable;
4   import java.util.ArrayList;
5   import java.util.List;
6   import java.util.Optional;
7
8   import com.example.demo.repository.entity.Cliente;
9   import com.example.demo.repository.entity.Cuenta;
10  import com.example.demo.repository.entity.Direccion;
```

```
11 import com.example.demo.repository.entity.Recomendacion;
12
13 import lombok.Data;
14 import lombok.ToString;
15
16 @Data
17 public class ClienteDTO implements Serializable{
18
19   private static final long serialVersionUID = 1L;
20   private Long id;
21   private String nif;
22   private String nombre;
23   private String apellidos;
24   private String claveSeguridad;
25   private String email;
26   @ToString.Exclude
27   private RecomendacionDTO recomendacionDTO;
28   @ToString.Exclude
29   private List<CuentaDTO> listaCuentasDTO;
30   @ToString.Exclude
31   private List<DireccionDTO> listaDireccionesDTO;
32
33   // Convertir una entidad a un DTO
34   public static ClienteDTO convertToDTO(Cliente cliente) {
35
36     ClienteDTO clienteDTO = new ClienteDTO();
37     clienteDTO.setId(cliente.getId());
38     clienteDTO.setNif(cliente.getNif());
39     clienteDTO.setNombre(cliente.getNombre());
40     clienteDTO.setApellidos(cliente.getApellidos());
41     clienteDTO.setClaveSeguridad(cliente.getClaveSeguridad());
42     clienteDTO.setEmail(cliente.getEmail());
43
44     RecomendacionDTO rec = RecomendacionDTO.converToDTO(cliente.getRecomendacion(),
clienteDTO);
45     clienteDTO.setRecomendacionDTO(rec);
46
47     // Cargamos la lista de cuentas, que como es un HashSet hemos de convertirla a
ArrayList
48     List<Cuenta> listaCuentas = new ArrayList<Cuenta>(cliente.getListaCuentas());
49     for(int i=0;i<listaCuentas.size();i++) {
50       CuentaDTO cuentadto = CuentaDTO.convertToDTO(listaCuentas.get(i), clienteDTO);
51       clienteDTO.getListaCuentasDTO().add(cuentadto);
52     }
53     // Cargamos la lista de direcciones, que como es un HashSet hemos de convertirla
a ArrayList
54     List<Direccion> listaDirecciones = new ArrayList<Direccion>
(cliente.getListaDirecciones());
55     for(int i=0;i<listaDirecciones.size();i++) {
56       DireccionDTO direccionDTO = DireccionDTO.convertToDTO(listaDirecciones.get(i),
clienteDTO);
57       clienteDTO.getListaDireccionesDTO().add(direccionDTO);
```

```
    }
    /*
    for(int i=0;i<cliente.getListaDirecciones().size();i++) {
      DireccionDTO direccionDTO = DireccionDTO.convertToDTO(cliente.getListaDirecciones().get(i), clienteDTO);
      clienteDTO.getListaDireccionesDTO().add(direccionDTO);
    }
    */

    return clienteDTO;
  }

  // Convertir un DTO a una entidad
  public static Cliente convertToEntity(ClienteDTO clienteDTO) {

    Cliente cliente = new Cliente();
    cliente.setId(clienteDTO.getId());
    cliente.setNif(clienteDTO.getNif());
    cliente.setNombre(clienteDTO.getNombre());
    cliente.setApellidos(clienteDTO.getApellidos());
    cliente.setClaveSeguridad(clienteDTO.getClaveSeguridad());
    cliente.setEmail(clienteDTO.getEmail());

    Recomendacion rec = RecomendacionDTO.convertToEntity(clienteDTO.getRecomendacionDTO(), cliente);
    cliente.setRecomendacion(rec);

    // Cargamos la lista de cuentas
    for(int i=0;i<clienteDTO.getListaCuentasDTO().size();i++) {
      Cuenta cuenta = CuentaDTO.convertToEntity(clienteDTO.getListaCuentasDTO().get(i));
      cliente.getListaCuentas().add(cuenta);
    }
    // Cargamos la lista de direcciones
    for(int i=0;i<clienteDTO.getListaDireccionesDTO().size();i++) {
      Direccion direccion = DireccionDTO.convertToEntity(clienteDTO.getListaDireccionesDTO().get(i), cliente);
      cliente.getListaDirecciones().add(direccion);
    }

    return cliente;
  }

  // Constructor vacio
  public ClienteDTO() {
    super();
    this.recomendacionDTO = new RecomendacionDTO();
    this.listaCuentasDTO = new ArrayList<CuentaDTO>();
    this.listaDireccionesDTO = new ArrayList<DireccionDTO>();
  }
}
```

De esta forma, al ejecutarlo si añadimos una nueva dirección al cliente con id 11 tenemos lo siguiente:



Y si comprobamos la tabla `clientesdirecciones` de la base de datos podemos ver que solo ha creado un registro solo, y que no ha borrado los demás y los ha vuelto a insertar.

Local instance 3306

Administration | Schemas | Query 13 | clientesdirecciones

SCHEMAS

Filter objects

- demospringboot
  - Tables
    - clientes
    - clientesdirecciones
    - cuentas
    - direcciones
    - recomendaciones
  - Views
  - Stored Procedures
  - Functions
- sys

Limit to 1000 rows

```
1 • SELECT * FROM demospringboot.clientesdirecciones;
```

100% 1:1

Result Grid Filter Rows: Search Edit:

| id | idcliente | iddireccion |
|---|---|---|
| 130 | 8 | 25 |
| 131 | 2 | 79 |
| 132 | 5 | 13 |
| 133 | 19 | 77 |
| 134 | 10 | 5 |
| 135 | 8 | 30 |
| 136 | 10 | 69 |
| 137 | 17 | 17 |
| 138 | 15 | 78 |
| 139 | 2 | 46 |
| 140 | 4 | 7 |
| 141 | 4 | 24 |
| 142 | 4 | 43 |
| 143 | 4 | 70 |
| 144 | 4 | 71 |
| 145 | 4 | 93 |
| 146 | 11 | 94 |
| NULL | NULL | NULL |

clientesdirecciones 1

De hecho, si comprobamos el log de STS podemos ver que la operación de borrado no se realiza:

```
Console  Problems  Debug Shell
demo6 copia - Demo6Application [Spring Boot App] /Library/Java/JavaVirtualMachines/jdk-17.0.4.1.jdk/Contents/Home/bin/java  (13 dic 2022 19:10:09) [pid: 19214]
2022-12-13T19:10:31.494+01:00  INFO 19214 --- [nio-8888-exec-4] c.e.demo.service.ClienteServiceImpl      : ClienteServiceImpl - findById: Buscar cliente por id: 11
Hibernate: select
c1_0.id,c1_0.apellidos,c1_0.claveseguridad,c1_0.email,l1_0.idcliente,l1_0.id,l1_0.banco,l1_0.dc,l1_0.numerocuenta,l1_0.saldoactual,l1_0.sucursal,l2_0.idcliente,l2_1.id,l2_1.cp,l2_1.descripcion,
l2_1.pais,c1_0.nif,c1_0.nombre from clientes c1_0 left join cuentas l1_0 on c1_0.id=l1_0.idcliente left join (clientesdirecciones l2_0 join direcciones l2_1 on l2_1.id=l2_0.iddireccion) on
c1_0.id=l2_0.idcliente where c1_0.id=?
Hibernate: select r1_0.id,r1_0.idcliente,r1_0.observaciones from recomendaciones r1_0 where r1_0.idcliente=?
2022-12-13T19:10:53.801+01:00  INFO 19214 --- [nio-8888-exec-5] c.e.d.w.controller.DireccionController   : DireccionController - save: Almacenamos la direccion DireccionDTO(id=null,
descripcion=Camino sin salida, pais=EEUU, cp=04563)
2022-12-13T19:10:53.801+01:00  INFO 19214 --- [nio-8888-exec-5] c.e.demo.service.ClienteServiceImpl      : ClienteServiceImpl - findById: Buscar cliente por id: 11
Hibernate: select
c1_0.id,c1_0.apellidos,c1_0.claveseguridad,c1_0.email,l1_0.idcliente,l1_0.id,l1_0.banco,l1_0.dc,l1_0.numerocuenta,l1_0.saldoactual,l1_0.sucursal,l2_0.idcliente,l2_1.id,l2_1.cp,l2_1.descripcion,
l2_1.pais,c1_0.nif,c1_0.nombre from clientes c1_0 left join cuentas l1_0 on c1_0.id=l1_0.idcliente left join (clientesdirecciones l2_0 join direcciones l2_1 on l2_1.id=l2_0.iddireccion) on
c1_0.id=l2_0.idcliente where c1_0.id=?
Hibernate: select r1_0.id,r1_0.idcliente,r1_0.observaciones from recomendaciones r1_0 where r1_0.idcliente=?
2022-12-13T19:10:53.807+01:00  INFO 19214 --- [nio-8888-exec-5] c.e.demo.service.DireccionServiceImpl    : DireccionServiceImpl - save: Salva la direccion del cliente: 11
Hibernate: insert into direcciones (cp, descripcion, pais) values (?, ?, ?)
Hibernate: insert into clientesdirecciones (idcliente, iddireccion) values (?, ?)
2022-12-13T19:10:53.834+01:00  INFO 19214 --- [nio-8888-exec-6] c.e.d.w.controller.DireccionController   : DireccionController - findAllByCliente: Listamos las direcciones del cliente: 11
2022-12-13T19:10:53.834+01:00  INFO 19214 --- [nio-8888-exec-6] c.e.demo.service.ClienteServiceImpl      : ClienteServiceImpl - findById: Buscar cliente por id: 11
Hibernate: select
c1_0.id,c1_0.apellidos,c1_0.claveseguridad,c1_0.email,l1_0.idcliente,l1_0.id,l1_0.banco,l1_0.dc,l1_0.numerocuenta,l1_0.saldoactual,l1_0.sucursal,l2_0.idcliente,l2_1.id,l2_1.cp,l2_1.descripcion,
l2_1.pais,c1_0.nif,c1_0.nombre from clientes c1_0 left join cuentas l1_0 on c1_0.id=l1_0.idcliente left join (clientesdirecciones l2_0 join direcciones l2_1 on l2_1.id=l2_0.iddireccion) on
c1_0.id=l2_0.idcliente where c1_0.id=?
Hibernate: select r1_0.id,r1_0.idcliente,r1_0.observaciones from recomendaciones r1_0 where r1_0.idcliente=?
2022-12-13T19:10:53.840+01:00  INFO 19214 --- [nio-8888-exec-6] c.e.demo.service.DireccionServiceImpl    : DireccionServiceImpl - findAllByCliente: Lista de todas las direcciones del cliente:
11
Hibernate: SELECT A.* FROM DIRECCIONES A, CLIENTESDIRECCIONES B  WHERE A.ID=B.IDDIRECCION AND B.IDCLIENTE = ?
```

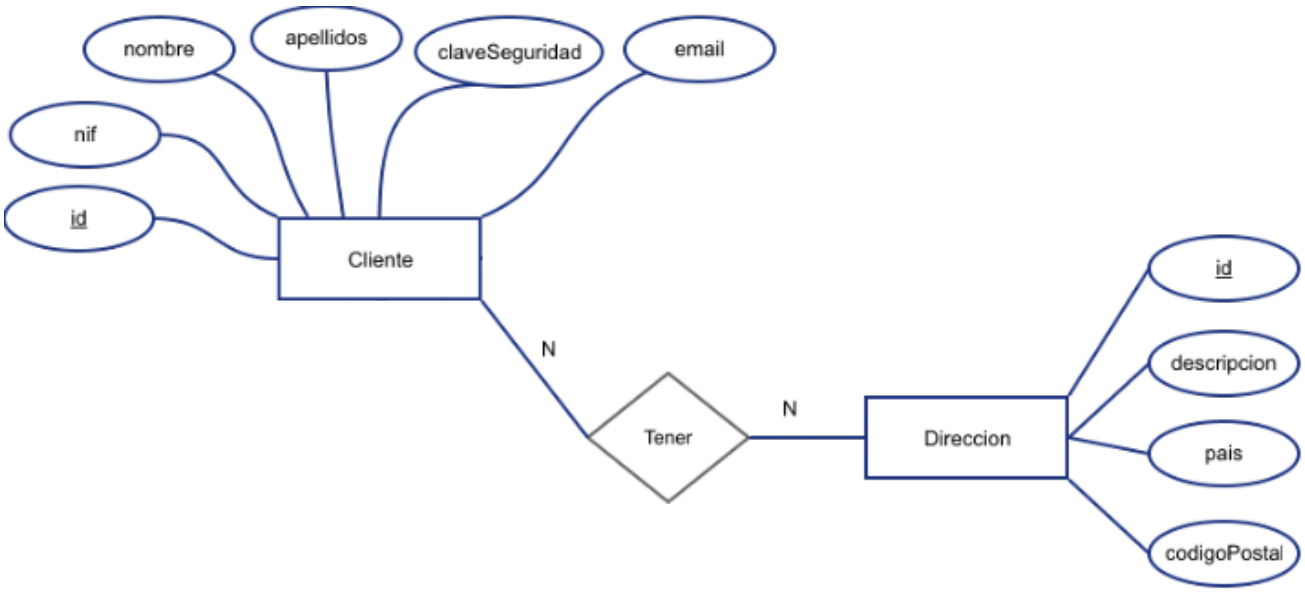
# Relaciones N a N sin atributo en medio de la relación y mapeando la entidad del medio.

Existe otra forma de mapear las relaciones muchos a muchos y es mapeando la tabla que se crea en la base de datos para mantener este tipo de relación.

Si la tabla no tiene ningún atributo que haya que almacenar en la relación (recordamos que ahora mismo solo almacenamos la clave primaria y las dos claves ajenas) no es necesario mapear la entidad del medio de la relación N a N, con lo que con el punto anterior podriamos solucionar el problema, pero esta no es la única solución al problema.

También es posible mapear la tabla del medio que existe en la relación N a N y crear relaciones de tipo 1 a N. Es más, en caso de que exista un atributo en medio de la relación es obligatorio hacer este tipo de mapeos.

De nuevo nos vamos a centrar en la siguiente relación:




Por ello lo primero que vamos a hacer es mapear la tabla que representa la relación `Tener` que existe entre `Cliente` y `Direccion`. Para ello vamos a crear una nueva entidad denominada `ClienteDireccion.java` que va a mapear la tabla `ClientesDirecciones`, que representa la relación `Tener`.

Al mapear `ClienteDireccion.java` nos vamos a encontrar que por un lado tiene una relación con `Cliente.java` y por otro lado tiene una relación con `Direccion.java`. Esto hace que se convierta en una relación de tipo 1 a N entre `ClienteDireccion.java` y `Cliente.java`. Además, existirá una relación de tipo 1 a N entre `ClienteDireccion.java` y `Direccion.java`. Esto hace que la relación N a N pase a transformasrse a 2 relaciones de tipo 1 a N.

Vamos a destacar de la implementación que, en este caso, la relación sí que se va a implementar mediante LAZY, ya que la relacion `ClienteDireccion.java` va a sobrecargar de manera innecesaria la aplicación, por lo que sera, de manera obligatoria, hacer esto cuando realicemos este tipo de implementación. También destacaremos la implementación de los métodos `equals()` y `hashCode()`, ya que van a trabajar con los identificadores de `Direccion` y `Cliente`.

Al implementar la clase se quedará de la siguiente forma:

```
package com.example.demo.repository.entity;

import java.util.Set;

```

```java
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;
import lombok.Data;

@Data
@Entity
@Table(name = "clientesdirecciones")
public class ClienteDireccion {

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

  @ManyToOne
  @JoinColumn(name="idcliente")
  private Cliente cliente;

  @ManyToOne
  @JoinColumn(name="iddireccion")
  private Direccion direccion;

  // Generamos el equals() y el hashcode() seleccionando cliente y direccion, pero
  // modificaremos lo que genera automaticamente anyadiendole tanto a cliente como a direccion
  // el metodo getId(), ya que nos interesa comparar sus ids y obtener el hashCode a partir de ellos.
  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    ClienteDireccion other = (ClienteDireccion) obj;
    return Objects.equals(cliente.getId(), other.cliente.getId()) && Objects.equals(direccion.getId(), other.direccion.getId());
  }

  @Override
  public int hashCode() {
    return Objects.hash(cliente.getId(), direccion.getId());
  }
}
```

El código de clase `Cliente.java` se queda de la siguiente forma:

```
package com.example.demo.repository.entity;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Objects;
import java.util.Set;

import jakarta.persistence.CascadeType;
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinTable;
import jakarta.persistence.ManyToMany;
import jakarta.persistence.OneToMany;
import jakarta.persistence.OneToOne;
import jakarta.persistence.Table;
import jakarta.persistence.JoinColumn;
import lombok.Data;
import lombok.ToString;

@Data
@Entity
@Table(name = "clientes")
public class Cliente {

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

  @Column(name = "nif")
  private String nif;

  @Column(name = "nombre")
  private String nombre;

  private String apellidos;

  @Column(name = "claveseguridad")
  private String claveSeguridad;

  private String email;

  @OneToOne(fetch = FetchType.EAGER, cascade = CascadeType.ALL, orphanRemoval = true, mappedBy = "cliente")
  @ToString.Exclude
  private Recomendacion recomendacion;
```

```
50
51    /*
52     * Mapeo de la lista de cuentas con List
53    @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL, mappedBy =
"cliente")
54    @ToString.Exclude
55    private List<Cuenta> listaCuentas;
56      */
57
58    // Mapeo de la lista de cuentas con Set
59    @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL, mappedBy =
"cliente")
60    @ToString.Exclude
61    private Set<Cuenta> listaCuentas;
62
63    /*
64    @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
65    // Tabla que mantiene la relacion N-N
66    @JoinTable(
67        // Nombre de la tabla
68        name = "clientesdirecciones",
69        // columna que almacena el id de cliente en la tabla clientesdirecciones
70        joinColumns = @JoinColumn(name = "idcliente"),
71        // columna que almacena el id de la direccion en la tabla clientesdirecciones
72        inverseJoinColumns = @JoinColumn(name = "iddireccion"))
73    @ToString.Exclude
74    private List<Direccion> listaDirecciones;
75    */
76
77    /*
78    @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
79    // Tabla que mantiene la relacion N-N
80    @JoinTable(
81        // Nombre de la tabla
82        name = "clientesdirecciones",
83        // columna que almacena el id de cliente en la tabla clientesdirecciones
84        joinColumns = @JoinColumn(name = "idcliente"),
85        // columna que almacena el id de la direccion en la tabla clientesdirecciones
86        inverseJoinColumns = @JoinColumn(name = "iddireccion"))
87    @ToString.Exclude
88    private Set<Direccion> listaDirecciones;
89    */
90
91
92    // Mapeo con la entidad ClienteDireccion con Set
93    // IMPORTATE: Poner a LAZY la relacion en este caso, ya que es la relacion N a N
94    @OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL, mappedBy =
"cliente")
95    @ToString.Exclude
96    private Set<ClienteDireccion> listaClientesDirecciones;
97
98    @Override
```

```
99    public boolean equals(Object obj) {
100     if (this == obj)
101       return true;
102     if (obj == null)
103       return false;
104     if (getClass() != obj.getClass())
105       return false;
106     Cliente other = (Cliente) obj;
107     return Objects.equals(id, other.id);
108   }
109
110   @Override
111   public int hashCode() {
112     return Objects.hash(id);
113   }
114
115   public Cliente() {
116     super();
117     this.recomendacion = new Recomendacion();
118     //this.listaCuentas = new ArrayList<Cuenta>();
119     //this.listaDirecciones = new ArrayList<Direccion>();
120     this.listaCuentas = new HashSet<Cuenta>();
121     //this.listaDirecciones = new HashSet<Direccion>();
122     this.listaClientesDirecciones = new HashSet<ClienteDireccion>();
123   }
124 }
```

El código de la clase `Direccion.java` queda de la siguiente forma:

```
1  package com.example.demo.repository.entity;
2
3  import java.util.ArrayList;
4  import java.util.HashSet;
5  import java.util.List;
6  import java.util.Objects;
7  import java.util.Set;
8
9  import jakarta.persistence.CascadeType;
10 import jakarta.persistence.Entity;
11 import jakarta.persistence.FetchType;
12 import jakarta.persistence.GeneratedValue;
13 import jakarta.persistence.GenerationType;
14 import jakarta.persistence.Id;
15 import jakarta.persistence.ManyToMany;
16 import jakarta.persistence.OneToMany;
17 import jakarta.persistence.Table;
18 import lombok.Data;
19 import lombok.ToString;
20
21 @Data
```

```java
@Entity
@Table(name = "direcciones")
public class Direccion {

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

  private String descripcion;
  private String pais;
  private String cp;

  /*
  @ManyToMany(cascade = { CascadeType.ALL }, mappedBy = "listaDirecciones")
  @ToString.Exclude
  private List<Cliente> listaClientes;
  */

  /*
  @ManyToMany(cascade = { CascadeType.ALL }, mappedBy = "listaDirecciones")
  @ToString.Exclude
  private Set<Cliente> listaClientes;
   */

  // Mapeo con la entidad ClienteDireccion con Set
  // IMPORTATE: Poner a LAZY la relacion en este caso, ya que es la relacion N a N
  @OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL, mappedBy = "direccion")
  @ToString.Exclude
  private Set<ClienteDireccion> listaClientesDirecciones;

  public Direccion() {
    super();
    //this.listaClientes = new ArrayList<Cliente>();
    //this.listaClientes = new HashSet<Cliente>();
    this.listaClientesDirecciones = new HashSet<ClienteDireccion>();
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Direccion other = (Direccion) obj;
    return Objects.equals(id, other.id);
  }

  @Override
  public int hashCode() {
```

```
    return Objects.hash(id);
  }
}
```

En cuanto a los DTO's que tenemos que modificar, lo que nos interesa es lo que se va a visualizar en la pantalla y lo que vamos a recoger de la misma, por lo que no tiene sentido crear un DTO de ClienteDireccion.java. Por ello dejaremos los DTO's como están, pero los adaptaremos a lo que necesitamos.

Por ello, el código de la clase `ClienteDTO.java` queda de la siguiente forma:

```java
package com.example.demo.model.dto;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

import com.example.demo.repository.entity.Cliente;
import com.example.demo.repository.entity.ClienteDireccion;
import com.example.demo.repository.entity.Cuenta;
import com.example.demo.repository.entity.Direccion;
import com.example.demo.repository.entity.Recomendacion;

import lombok.Data;
import lombok.ToString;

@Data
public class ClienteDTO implements Serializable{

  private static final long serialVersionUID = 1L;
  private Long id;
  private String nif;
  private String nombre;
  private String apellidos;
  private String claveSeguridad;
  private String email;
  @ToString.Exclude
  private RecomendacionDTO recomendacionDTO;
  @ToString.Exclude
  private List<CuentaDTO> listaCuentasDTO;
  @ToString.Exclude
  private List<DireccionDTO> listaDireccionesDTO;

  // Convertir una entidad a un DTO
  public static ClienteDTO convertToDTO(Cliente cliente) {

    ClienteDTO clienteDTO = new ClienteDTO();
    clienteDTO.setId(cliente.getId());
    clienteDTO.setNif(cliente.getNif());
```

```
    clienteDTO.setNombre(cliente.getNombre());
    clienteDTO.setApellidos(cliente.getApellidos());
    clienteDTO.setClaveSeguridad(cliente.getClaveSeguridad());
    clienteDTO.setEmail(cliente.getEmail());

    RecomendacionDTO rec = RecomendacionDTO.converToDTO(cliente.getRecomendacion(), clienteDTO);
    clienteDTO.setRecomendacionDTO(rec);

    // Cargamos la lista de cuentas, que como es un HashSet hemos de convertirla a ArrayList
    List<Cuenta> listaCuentas = new ArrayList<Cuenta>(cliente.getListaCuentas());
    for(int i=0;i<listaCuentas.size();i++) {
      CuentaDTO cuentadto = CuentaDTO.convertToDTO(listaCuentas.get(i), clienteDTO);
      clienteDTO.getListaCuentasDTO().add(cuentadto);
    }

    /*
    for(int i=0;i<cliente.getListaDirecciones().size();i++) {
      DireccionDTO direccionDTO = DireccionDTO.convertToDTO(cliente.getListaDirecciones().get(i), clienteDTO);
      clienteDTO.getListaDireccionesDTO().add(direccionDTO);
    }
    */

    // Cargamos la lista de clientes direcciones, que como es un HashSet hemos de convertirla a ArrayList
    List<ClienteDireccion> listaClientesDirecciones = new ArrayList<ClienteDireccion>(cliente.getListaClientesDirecciones());
    for(int i=0;i<listaClientesDirecciones.size();i++) {
      // Como solo nos interesa la direccion la lista que tenemos sera lista de direccionesDTO
      DireccionDTO direccionDTO = DireccionDTO.convertToDTO(listaClientesDirecciones.get(i).getDireccion(), clienteDTO);
      clienteDTO.getListaDireccionesDTO().add(direccionDTO);
    }


    return clienteDTO;
  }

  // Convertir un DTO a una entidad
  public static Cliente convertToEntity(ClienteDTO clienteDTO) {

    Cliente cliente = new Cliente();
    cliente.setId(clienteDTO.getId());
    cliente.setNif(clienteDTO.getNif());
    cliente.setNombre(clienteDTO.getNombre());
    cliente.setApellidos(clienteDTO.getApellidos());
    cliente.setClaveSeguridad(clienteDTO.getClaveSeguridad());
    cliente.setEmail(clienteDTO.getEmail());
```

```
84
85      Recomendacion rec =
RecomendacionDTO.convertToEntity(clienteDTO.getRecomendacionDTO(), cliente);
86      cliente.setRecomendacion(rec);
87
88      // Cargamos la lista de cuentas
89      for(int i=0;i<clienteDTO.getListaCuentasDTO().size();i++) {
90        Cuenta cuenta =
CuentaDTO.convertToEntity(clienteDTO.getListaCuentasDTO().get(i));
91        cliente.getListaCuentas().add(cuenta);
92      }
93      // Cargamos la lista de direcciones
94      for(int i=0;i<clienteDTO.getListaDireccionesDTO().size();i++) {
95        Direccion direccion =
DireccionDTO.convertToEntity(clienteDTO.getListaDireccionesDTO().get(i), cliente);
96        ClienteDireccion cd = new ClienteDireccion();
97        cd.setCliente(cliente);
98        cd.setDireccion(direccion);
99        cliente.getListaClientesDirecciones().add(cd);
100     }
101
102     return cliente;
103   }
104
105   // Constructor vacio
106   public ClienteDTO() {
107     super();
108     this.recomendacionDTO = new RecomendacionDTO();
109     this.listaCuentasDTO = new ArrayList<CuentaDTO>();
110     this.listaDireccionesDTO = new ArrayList<DireccionDTO>();
111   }
112 }
113
```

El código de la clase `DireccionDTO.java` queda de la siguiente forma:

```
1  package com.example.demo.model.dto;
2
3  import java.io.Serializable;
4  import java.util.ArrayList;
5  import java.util.List;
6
7  import com.example.demo.repository.entity.Cliente;
8  import com.example.demo.repository.entity.ClienteDireccion;
9  import com.example.demo.repository.entity.Direccion;
10
11 import lombok.Data;
12 import lombok.ToString;
13
14 @Data
```

```java
public class DireccionDTO implements Serializable {

  private static final long serialVersionUID = 1L;
  private Long id;
  private String descripcion;
  private String pais;
  private String cp;
  @ToString.Exclude
  private List<ClienteDTO> listaClientesDTO;

  public static DireccionDTO convertToDTO(Direccion direccion, ClienteDTO clienteDTO) {
    DireccionDTO direccionDTO = new DireccionDTO();
    direccionDTO.setId(direccion.getId());
    direccionDTO.setDescripcion(direccion.getDescripcion());
    direccionDTO.setPais(direccion.getPais());
    direccionDTO.setCp(direccion.getCp());

    // No tiene sentido mapear todos los clientes que tiene la direccion, puesto que
    // al mapear cada cliente volveriamos a mapear sus direcciones, y así
    // sucesivamente.
    direccionDTO.getListaClientesDTO().add(clienteDTO);

    return direccionDTO;
  }

  public static Direccion convertToEntity(DireccionDTO direccionDTO, Cliente cliente) {
    Direccion direccion = new Direccion();
    direccion.setId(direccionDTO.getId());
    direccion.setDescripcion(direccionDTO.getDescripcion());
    direccion.setPais(direccionDTO.getPais());
    direccion.setCp(direccionDTO.getCp());

    // No tiene sentido mapear todos los clientes que tiene la direccion, puesto que
    // al mapear cada cliente volveriamos a mapear sus direcciones, y así
    // sucesivamente.
    //direccion.getListaClientes().add(cliente);

    // Creamos una entidad de ClienteDireccion y asignamos la relacion de ambos.
    ClienteDireccion cd = new ClienteDireccion();
    cd.setCliente(cliente);
    cd.setDireccion(direccion);
    direccion.getListaClientesDirecciones().add(cd);
    cliente.getListaClientesDirecciones().add(cd);

    return direccion;

  }

  // Constructor vacio
  public DireccionDTO() {
```

```
    super();
    this.listaClientesDTO = new ArrayList<ClienteDTO>();
  }
}
```

En cuanto a la clase `DireccionServiceImpl.java` tendremos que adaptar el método `save()` para almacenar de forma correcta en la entidad `ClienteDireccion.java`. Quedará de la siguiente forma:

```
package com.example.demo.service;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.example.demo.model.dto.ClienteDTO;
import com.example.demo.model.dto.DireccionDTO;
import com.example.demo.repository.dao.ClienteRepository;
import com.example.demo.repository.dao.DireccionRepository;
import com.example.demo.repository.entity.Cliente;
import com.example.demo.repository.entity.ClienteDireccion;
import com.example.demo.repository.entity.Direccion;

@Service
public class DireccionServiceImpl implements DireccionService {

  private static final Logger log =
LoggerFactory.getLogger(DireccionServiceImpl.class);

  @Autowired
  private DireccionRepository direccionRepository;

  @Autowired
  private ClienteRepository clienteRepository;

  @Override
  public List<DireccionDTO> findAllByCliente(ClienteDTO clienteDTO) {

    log.info("DireccionServiceImpl - findAllByCliente: Lista de todas las direcciones
del cliente: "
        + clienteDTO.getId());

    // Obtenemos la lista de direcciones del cliente
    List<Direccion> lista = (List<Direccion>)
direccionRepository.findAllByCliente(clienteDTO.getId());
    // Creamos una lista de DireccionDTO que será la que devolvamos al controlador
```

```
    List<DireccionDTO> listaDireccionesDTO = new ArrayList<DireccionDTO>();
    // Recorremos la lista de direcciones y las mapeamos a DTO
    for (int i = 0; i < lista.size(); ++i) {
      listaDireccionesDTO.add(DireccionDTO.convertToDTO(lista.get(i), clienteDTO));
    }
    // Devolvemos la lista de DTO's
    return listaDireccionesDTO;
  }

  @Override
  public void save(DireccionDTO direccionDTO) {

    log.info("DireccionServiceImpl - save: Salva la direccion del cliente: "
        + direccionDTO.getListaClientesDTO().get(0).getId());

    // Solo hay un clienteDTO en la direccionDTO
    /*
    Optional<Cliente> cliente = clienteRepository.findById(direccionDTO.getListaClientesDTO().get(0).getId());
    if (cliente.isPresent()) {
      // Creamos las relaciones en las entidades
      Direccion direccion = DireccionDTO.convertToEntity(direccionDTO, cliente.get());
      cliente.get().getListaDirecciones().add(direccion);
      // Almacenamos la direccion, y por la relacion que tiene, almacenará la relacion
      // N a N
      direccionRepository.save(direccion);
    }
    */

    // Solo hay un clienteDTO en la direccionDTO. En este caso tendremos que creare la
    // entidad ClienteDireccion y asignarle las relaciones.
    Optional<Cliente> cliente = clienteRepository.findById(direccionDTO.getListaClientesDTO().get(0).getId());
    if (cliente.isPresent()) {
      // Creamos las relaciones en las entidades
      Direccion direccion = DireccionDTO.convertToEntity(direccionDTO, cliente.get());
      ClienteDireccion cd = new ClienteDireccion();
      cd.setDireccion(direccion);
      cd.setCliente(cliente.get());
      // Almacenamos la direccion, y por la relacion que tiene, almacenará la relacion
      // N a N
      direccionRepository.save(direccion);
    }
  }
}
```

Tras esto si probamos el programa sigue funcionando correctamente, solo que hemos modificado como está implementada la relación N a N.

Como el caso que vamos a explicar va a hacer uso de un atributo en medio de la relación lo dejaremos para apartados más adelante, puesto que es necesario explicar otros apartados antes para proceder a implementar esta solución.

# El lenguaje de consultas JPQL

`Java Persistence Query Language (JPQL)` es un lenguaje de consulta orientado a objetos independiente de la plataforma definido como parte de la especificación Java Persistence API (JPA).

JPQL es usado para hacer consultas contra las entidades almacenadas en una base de datos relacional. Está inspirado en gran medida por SQL, y sus consultas se asemejan a las consultas SQL en la sintaxis, pero opera con objetos entidad de JPA en lugar de hacerlo directamente con las tablas de la base de datos.

Además de recuperar objetos (consultas SELECT), JPQL soporta consultas de actualización (UPDATE) y borrado (DELETE).

Al contrario, JPQL nos permite realizar consultas a través de multitud de criterios posibles (dentro de JPQL los criterios se entiende como filtros) y obtener más de un objeto por consulta.

Si recordáis, a la hora de definir el método `findAllByCliente` de la clase `CuentaRepository.java` definimos el método poniendo el siguiente código:

```
package com.example.demo.repository.dao;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import com.example.demo.repository.entity.Cuenta;

import jakarta.transaction.Transactional;

@Repository
@Transactional
public interface CuentaRepository extends JpaRepository<Cuenta, Long>{

  /*
  @Query(value = "SELECT * FROM cuentas WHERE idcliente = ?1", nativeQuery = true)
  public List<Cuenta> findAllByCliente(Long idCliente);
  */

  @Query(value = "SELECT * FROM cuentas WHERE idcliente = :idc", nativeQuery = true)
```

```
24    public List<Cuenta> findAllByCliente(@Param("idc") Long idCliente);
25
26  }
```

Pasando la consulta de código nativo SQL a código JPQL (que es muy similar al SQL pero orientado a las entidades) quedaría de la siguiente forma:

```
1   package com.example.demo.repository.dao;
2
3   import java.util.List;
4
5   import org.springframework.data.jpa.repository.JpaRepository;
6   import org.springframework.data.jpa.repository.Query;
7   import org.springframework.data.repository.query.Param;
8   import org.springframework.stereotype.Repository;
9
10  import com.example.demo.repository.entity.Cuenta;
11
12  import jakarta.transaction.Transactional;
13
14  @Repository
15  @Transactional
16  public interface CuentaRepository extends JpaRepository<Cuenta, Long>{
17
18    /*
19    @Query(value = "SELECT * FROM cuentas WHERE idcliente = ?1", nativeQuery = true)
20    public List<Cuenta> findAllByCliente(Long idCliente);
21    */
22
23    /*
24    @Query(value = "SELECT * FROM cuentas WHERE idcliente = :idc", nativeQuery = true)
25    public List<Cuenta> findAllByCliente(@Param("idc") Long idCliente);
26    */
27
28    @Query(value = "SELECT c FROM Cuenta WHERE c.cliente.id = :idc")
29    public List<Cuenta> findAllByCliente(@Param("idc") Long idCliente);
30  }
```

Donde `Cuenta` hace referencia a la entidad `Cuenta.java`, y `c.cliente.id` hace referencia al `id` del objeto de tipo `Cliente` que hay declarado dentro de la clase `Cuenta.java`. Como podéis observar es muy similar al SQL pero orientado a entidades.

No vamos a ahondar muchos más en este tema, ya que es muy sencillo de trabajar.

# Extendiendo los repositorios

Spring Data facilita mucho el proceso de trabajar con entidades simplemente definiendo interfaces de repositorio, como ya hemos hecho en esta unidad.

Los repositorios vienen con un conjunto de métodos predefinidos y permiten la posibilidad de agregar métodos personalizados en cada interfaz.

Sin embargo, si queremos agregar un método personalizado más complejo, que no es de los predeterminados, y que mediante una anotación `@Query` no puede ser resuelto, tenemos que implementar y programar este método que declararemos en la interfaz.

Por ejemplo, supongamos que necesitamos implementar el método que permite añadir cuentas a un cliente. Esta operación está ya implementada en el repositorio `CuentaRepository.java` por defecto, pero, aún así, vamos a suponer que no disponemos de ella y que debemos implementarla.

Para disponer de dicha operación en el repositorio lo primero que tenemos que hacer es:

1. Crear una nueva interfaz que contendrá las declaraciones del método nuevo. Crearemos la interfaz `CustomCuentaRepository.java` y definiremos el nuevo método:

```java
package com.example.demo.repository.dao;

import com.example.demo.repository.entity.Cuenta;

public interface CustomCuentaRepository {

  public void saveCustom(Cuenta cuenta);
}
```

2. Ahora hacemos que la interfaz que tenemos para definir las operaciones de las cuentas, que es `CuentaRepository.java`, extienda también la nueva interfaz. Quedará de la siguiente forma:

```java
package com.example.demo.repository.dao;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import com.example.demo.repository.entity.Cuenta;

import jakarta.transaction.Transactional;

@Repository
@Transactional
public interface CuentaRepository extends JpaRepository<Cuenta, Long>, CustomCuentaRepository{

  /*
  @Query(value = "SELECT * FROM cuentas WHERE idcliente = ?1", nativeQuery = true)
  public List<Cuenta> findAllByCliente(Long idCliente);
```

```
  */

  /*
  @Query(value = "SELECT * FROM cuentas WHERE idcliente = :idc", nativeQuery = true)
  public List<Cuenta> findAllByCliente(@Param("idc") Long idCliente);
  */

  @Query(value = "SELECT c FROM Cuenta c WHERE c.cliente.id = :idc")
  public List<Cuenta> findAllByCliente(@Param("idc") Long idCliente);
}
```

3. Observamos que `CuentaRepository` extiende de `JpaRepository` y de `CustomCuentaRepository`. Ahora podemos llamar a nuestro nuevo método, pero ¿qué pasa con su código?
4. Ahora es solo cuestión de implementar el código que necesitamos. Crearemos una clase nueva denominada `CustomCuentaRepositoryImpl.java` que implementa nuestra interfaz `CustomCuentaRepository` recién creada. La clase va a quedar de la siguiente forma:

```
package com.example.demo.repository.dao;

import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

import com.example.demo.repository.entity.Cliente;
import com.example.demo.repository.entity.Cuenta;

import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import jakarta.transaction.Transactional;

@Repository
@Transactional
public class CustomCuentaRepositoryImpl implements CustomCuentaRepository{

  @PersistenceContext
  private EntityManager entityManager;

  @Autowired
  private ClienteRepository clienteRepository;

  @Override
  public void saveCustom(Cuenta cuenta) {
    Optional<Cliente> cliente = clienteRepository.findById(cuenta.getCliente().getId());
    cuenta.setCliente(cliente.get());
    entityManager.persist(cuenta);
  }
```

```
}
```

5. Lo primero es poner las anotaciones `@Repository` y `@Transactional`. Ya las hemos explicado con anterioridad.
6. Podemos ver en dicha clase que tenemos un objeto denominador `EntityManager`, que es el gestor de las entidades que tenemos mapeadas en la aplicación. Este objeto, que es inyectado en la clase `CustomCuentaRepositoryImpl.java` mediante la anotación `@PersistenceContext`, que permite inyectar una unidad de persistencia con la que trabaja JPA, permitiendo así realizar operaciones con las entidades (salvar, borrar, obtener...).
7. Las operaciones del `EntityManager` que podemos realizar (las más comunes) son:
    1. `persist`: permite almacenar un objeto en la base de datos.
    2. `contains`: comprueba si la entidad está persistida en la base de datos o no. Devuelve true o false.
    3. `find`: localiza una entidad en la base de datos a través de su clave primaria.
    4. `remove`: elimina una entidad de la base de datos.
    5. `flush`: se encarga de sincronizar con la base de datos.
8. En cuanto a los DTO's, el DTO de `Cuenta`, que será `CuentaDTO.java`, es necesario destacarlo porque hemos comentado dentro, a la hora de convertir a entidad, el mapeo de cliente para evitar el error de referencias circulares. Por eso en `CustomCuentaRepositoryImpl.java` buscamos el cliente y lo pasamos directamente a la entidad `Cuenta`. Por ello hay que tener en cuenta que en el `convertToEntity` debe estar comentado este paso. El código será:

```java
package com.example.demo.model.dto;

import java.io.Serializable;

import com.example.demo.repository.entity.Cliente;
import com.example.demo.repository.entity.Cuenta;

import lombok.Data;
import lombok.ToString;

@Data
public class CuentaDTO implements Serializable{

  private static final long serialVersionUID = 1L;
  private Long id;
  private String banco;
  private String sucursal;
  private String dc;
  private String numeroCuenta;
  private float saldoActual;
  @ToString.Exclude
  private ClienteDTO clienteDTO;

```

```
24    public static CuentaDTO convertToDTO(Cuenta cuenta, ClienteDTO cliente) {
25
26      CuentaDTO cuentaDTO = new CuentaDTO();
27      cuentaDTO.setId(cuenta.getId());
28      cuentaDTO.setBanco(cuenta.getBanco());
29      cuentaDTO.setSucursal(cuenta.getSucursal());
30      cuentaDTO.setDc(cuenta.getDc());
31      cuentaDTO.setNumeroCuenta(cuenta.getNumeroCuenta());
32      cuentaDTO.setSaldoActual(cuenta.getSaldoActual());
33      cuentaDTO.setClienteDTO(cliente);
34
35      return cuentaDTO;
36    }
37
38    public static Cuenta convertToEntity(CuentaDTO cuentaDTO) {
39
40      Cuenta cuenta = new Cuenta();
41      cuenta.setId(cuentaDTO.getId());
42      cuenta.setBanco(cuentaDTO.getBanco());
43      cuenta.setSucursal(cuentaDTO.getSucursal());
44      cuenta.setDc(cuentaDTO.getDc());
45      cuenta.setNumeroCuenta(cuentaDTO.getNumeroCuenta());
46      cuenta.setSaldoActual(cuentaDTO.getSaldoActual());
47
48      // Esta linea produce un error de referencia circular, por lo que no
49      // mapearemos el cliente a entidad hasta que haga falta
50      /*
51      Cliente cliente = ClienteDTO.convertToEntity(cuentaDTO.getClienteDTO());
52      cuenta.setCliente(cliente);
53      */
54
55      return cuenta;
56    }
57  }
```

9. Inyectaremos un objeto `ClienteRepository` para buscar la entidad cliente.
10. Ahora vamos a modificar el controlador `CuentaController.java`, ya que hemos de modificar la operación `save` para adaptarla a este cambio. Hemos seguido los mismos pasos para dar de alta un cliente (operación `add`).El código será el siguiente (despues procedemos a comentarlo):

```
1  package com.example.demo.web.controller;
2
3  import java.util.List;
4
5  import org.slf4j.Logger;
6  import org.slf4j.LoggerFactory;
7  import org.springframework.beans.factory.annotation.Autowired;
8  import org.springframework.stereotype.Controller;
9  import org.springframework.web.bind.annotation.GetMapping;
```

```java
10 import org.springframework.web.bind.annotation.ModelAttribute;
11 import org.springframework.web.bind.annotation.PathVariable;
12 import org.springframework.web.bind.annotation.PostMapping;
13 import org.springframework.web.servlet.ModelAndView;
14
15 import com.example.demo.model.dto.ClienteDTO;
16 import com.example.demo.model.dto.CuentaDTO;
17 import com.example.demo.service.ClienteService;
18 import com.example.demo.service.CuentaService;
19
20 @Controller
21 public class CuentaController {
22
23   private static final Logger log =
LoggerFactory.getLogger(CuentaController.class);
24
25   @Autowired
26   private CuentaService cuentaService;
27
28   @Autowired
29   private ClienteService clienteService;
30
31   @GetMapping("/clientes/{idCliente}/cuentas")
32   public ModelAndView findAllByCliente(@PathVariable Long idCliente) {
33
34     log.info("CuentaController - findAllByCliente: Mostramos todas las cuentas
del cliente: " + idCliente);
35
36     // Obtenemos el cliente
37     ClienteDTO clienteDTO = new ClienteDTO();
38     clienteDTO.setId(idCliente);
39     clienteDTO = clienteService.findById(clienteDTO);
40
41     // Obtenemos la lista de cuentas
42     List<CuentaDTO> listaCuentasDTO =
cuentaService.findAllByCliente(clienteDTO);
43
44     ModelAndView mav = new ModelAndView("cuentas");
45     mav.addObject("clienteDTO", clienteDTO);
46     mav.addObject("listaCuentasDTO", listaCuentasDTO);
47
48     return mav;
49   }
50
51   // Borrar una cuenta
52   @GetMapping("/clientes/{idCliente}/cuentas/delete/{idCuenta}")
53   public ModelAndView delete(@PathVariable("idCliente") Long idCliente,
54       @PathVariable("idCuenta") Long idCuenta) {
55
56     //log.info("CuentaController - delete: Usamos el metodo 1 para borrar la
cuenta:" + idCuenta);
```

```
    log.info("CuentaController - delete: Usamos el metodo 2 para borrar la cuenta:" + idCuenta);

    // Creamos el clienteDTO
    ClienteDTO clienteDTO = new ClienteDTO();
    clienteDTO.setId(idCliente);

    // Creamos la cuentaDTO
    CuentaDTO cuentaDTO = new CuentaDTO();
    cuentaDTO.setId(idCuenta);

    // Invocamos al metodo de borrar
    // cuentaService.delete(cuentaDTO);
    cuentaService.delete(cuentaDTO, clienteDTO);

    // Redireccionamos para volver a invocar al metodo que escucha /clientes
    ModelAndView mav = new ModelAndView("redirect:/clientes/{idCliente}/cuentas");

    return mav;
  }

  @GetMapping("/clientes/{idCliente}/cuentas/add")
  public ModelAndView add(@PathVariable Long idCliente) {

    log.info("CuentaController - add: Alta de cuenta del cliente: " + idCliente);

    // Obtenemos el cliente para luego poner sus datos en la pantalla
    ClienteDTO clienteDTO = new ClienteDTO();
    clienteDTO.setId(idCliente);
    clienteDTO = clienteService.findById(clienteDTO);
    // pasamos el cliente y la nueva cuenta a la vista
    ModelAndView mav = new ModelAndView("cuentaform");
    mav.addObject("clienteDTO", clienteDTO);
    mav.addObject("cuentaDTO", new CuentaDTO());
    mav.addObject("add", true);
    return mav;
  }

  @PostMapping("/clientes/{idCliente}/cuentas/save")
  public ModelAndView save(@PathVariable Long idCliente, @ModelAttribute("cuentaDTO") CuentaDTO cuentaDTO) {

    log.info("CuentaController - save: Salvando la cuenta del cliente: " + idCliente);

    // Obtenemos el cliente para luego poner sus datos en la pantalla
    ClienteDTO clienteDTO = new ClienteDTO();
    clienteDTO.setId(idCliente);
    // Asignamos a la cuenta el cliente (no hace falta buscarlo ya que al salvarlo lo buscaremos)
```

```
103         cuentaDTO.setClienteDTO(clienteDTO);
104 
105         // invocamos la operacion save a la capa de servicio de cuenta
106         cuentaService.save(cuentaDTO);
107         // Retornamos a la lista de cuentas del cliente
108         ModelAndView mav = new
    ModelAndView("redirect:/clientes/{idCliente}/cuentas");
109         return mav;
110       }
111     }
```

11. Creamos el método `save` en la clase `CuentaService.java`. En dicho método, que será implementado en `CuentaServiceImpl.java`, recibiremos el DTO y lo convertiremos a una entidad, puesto que los DTO's no deben llegar a la capa de repositorio. Quedará de la siguiente forma:

```
1  package com.example.demo.service;
2
3  import java.util.List;
4
5  import com.example.demo.model.dto.ClienteDTO;
6  import com.example.demo.model.dto.CuentaDTO;
7
8  public interface CuentaService {
9
10   public List<CuentaDTO> findAllByCliente(ClienteDTO clienteDTO);
11   public void delete(CuentaDTO cuentaDTO);
12   public void delete(CuentaDTO cuentaDTO, ClienteDTO clienteDTO);
13   public void save(CuentaDTO cuentaDTO);
14 }
```

```
1  package com.example.demo.service;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.Optional;
6
7  import org.slf4j.Logger;
8  import org.slf4j.LoggerFactory;
9  import org.springframework.beans.factory.annotation.Autowired;
10 import org.springframework.stereotype.Service;
11
12 import com.example.demo.model.dto.ClienteDTO;
13 import com.example.demo.model.dto.CuentaDTO;
14 import com.example.demo.repository.dao.ClienteRepository;
15 import com.example.demo.repository.dao.CuentaRepository;
16 import com.example.demo.repository.entity.Cliente;
17 import com.example.demo.repository.entity.Cuenta;
18
19 @Service
```

```java
public class CuentaServiceImpl implements CuentaService {

  private static final Logger log = LoggerFactory.getLogger(CuentaServiceImpl.class);

  @Autowired
  private CuentaRepository cuentaRepository;

  @Autowired
  private ClienteRepository clienteRepository;

  @Override
  public List<CuentaDTO> findAllByCliente(ClienteDTO clienteDTO) {
    log.info("CuentaServiceImpl - finfindAllByClientedAll: Lista de todas las cuentas del cliente: "
        + clienteDTO.getId());

    // Obtenemos la lista de cuentas del cliente
    List<Cuenta> lista = (List<Cuenta>) cuentaRepository.findAllByCliente(clienteDTO.getId());
    // Creamos una lista de CuentaDTO que será la que devolvamos al controlador
    List<CuentaDTO> listaResultadoDTO = new ArrayList<CuentaDTO>();
    // Recorremos la lista de cuentas y las mapeamos a DTO
    for (int i = 0; i < lista.size(); ++i) {
      listaResultadoDTO.add(CuentaDTO.convertToDTO(lista.get(i), clienteDTO));
    }
    // Devolvemos la lista de DTO's
    return listaResultadoDTO;
  }

  /*
   * Metodo 1 de borrado de un hijo en una relacion 1 a N:
   * ejecutamos directamente el borrado por medio del id
   */
  @Override
  public void delete(CuentaDTO cuentaDTO) {
    log.info("CuentaServiceImpl - delete: Metodo 1: borramos la cuenta: " + cuentaDTO.toString());

    cuentaRepository.deleteById(cuentaDTO.getId());
  }

  /*
   * Metodo 2 de borrado de un hijo en una relacion 1 a N:
   * ejecutamos directamente el borrado por medio del id
   */
  @Override
  public void delete(CuentaDTO cuentaDTO, ClienteDTO clienteDTO) {
    log.info("CuentaServiceImpl - delete: Metodo 1: borramos la cuenta: " + cuentaDTO.toString());
    // Localizamos ambos objetos
    Optional<Cliente> cliente = clienteRepository.findById(clienteDTO.getId());
```

```
    Optional<Cuenta> cuenta = cuentaRepository.findById(cuentaDTO.getId());

    // Eliminamos la cuenta de la lista de cuentas del cliente.
    // Con esto eliminamos la relacion y podemos borrar la cuenta
    cliente.get().getListaCuentas().remove(cuenta.get());

    // Eliminamos la cuenta
    cuentaRepository.deleteById(cuenta.get().getId());

    // Salvamos el cliente con la nueva situacioni
    clienteRepository.save(cliente.get());
  }

  @Override
  public void save(CuentaDTO cuentaDTO) {
    log.info("CuentaServiceImpl - save: salvamos la cuenta : " +
cuentaDTO.toString());

    Cuenta cuenta = CuentaDTO.convertToEntity(cuentaDTO);
    // Seguimos sin tener la necesidad de buscarlo
    Cliente cliente = new Cliente();
    cliente.setId(cuentaDTO.getClienteDTO().getId());
    cuenta.setCliente(cliente);

    //cuentaRepository.save(cuenta);
    cuentaRepository.saveCustom(cuenta);
  }
}
```

12. La vista `cuentas.html` tiene todo lo necesario, por lo que no tenemos que tocarla.
13. Creamos la vista `cuentaform.html` para poder dar de alta las cuentas. El código será el siguiente:

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">

<head>
    <meta charset="UTF-8" />
    <title th:utext="${add} ? 'Crear una cuenta' : 'Editar una cuenta'" />
 </head>

<body>
    <h1 th:text="${add} ? 'Crear una cuenta' : 'Editar una cuenta'" />
    <br/>
    <br/>
    <form th:action="@{/clientes/{idCliente}/cuentas/save
(idCliente=${clienteDTO.id})}" th:object="${cuentaDTO}" method="POST">
        <table border="0">
            <tr th:if="${cuentaDTO.id}">
                <td>ID</td>
```

```
                <td>:</td>
                <td>
                  <input type="text" th:field="*{id}" readonly />
                </td>
            </tr>
            <tr>
                <td>Banco</td>
                <td>:</td>
                <td>
                    <input type="text" th:field="*{banco}" />
                </td>
            </tr>
            <tr>
                <td>Sucursal</td>
                <td>:</td>
                <td>
                    <input type="text" th:field="*{sucursal}" />
                </td>
            </tr>
            <tr>
                <td>Dígito de control</td>
                <td>:</td>
                <td>
                    <input type="text" th:field="*{dc}" />
                </td>
            </tr>
            <tr>
                <td>Número de cuenta</td>
                <td>:</td>
                <td>
                    <input type="text" th:field="*{numeroCuenta}" />
                </td>
            </tr>
            <tr th:if="${cuentaDTO.id}">
                <td>Saldo actual</td>
                <td>:</td>
                <td>
                  <input type="text" th:field="*{saldoActual}" readonly />
                </td>
            </tr>
        </table>
        <input type="submit" th:value="${add} ? 'Crear' : 'Actualizar'" />
</form>
    <br/>
    <br/>
    <a th:href="@{/clientes/{idCliente}/cuentas
(idCliente=${clienteDTO.id})}">Volver a la lista de cuentas</a>
</html>
```

14. Si probamos la aplicación podemos comprobar que funciona correctamente el poder dar de alta cuentas y persistirlas en la base de datos.

# Mapeando relaciones (Parte 2 de 2)

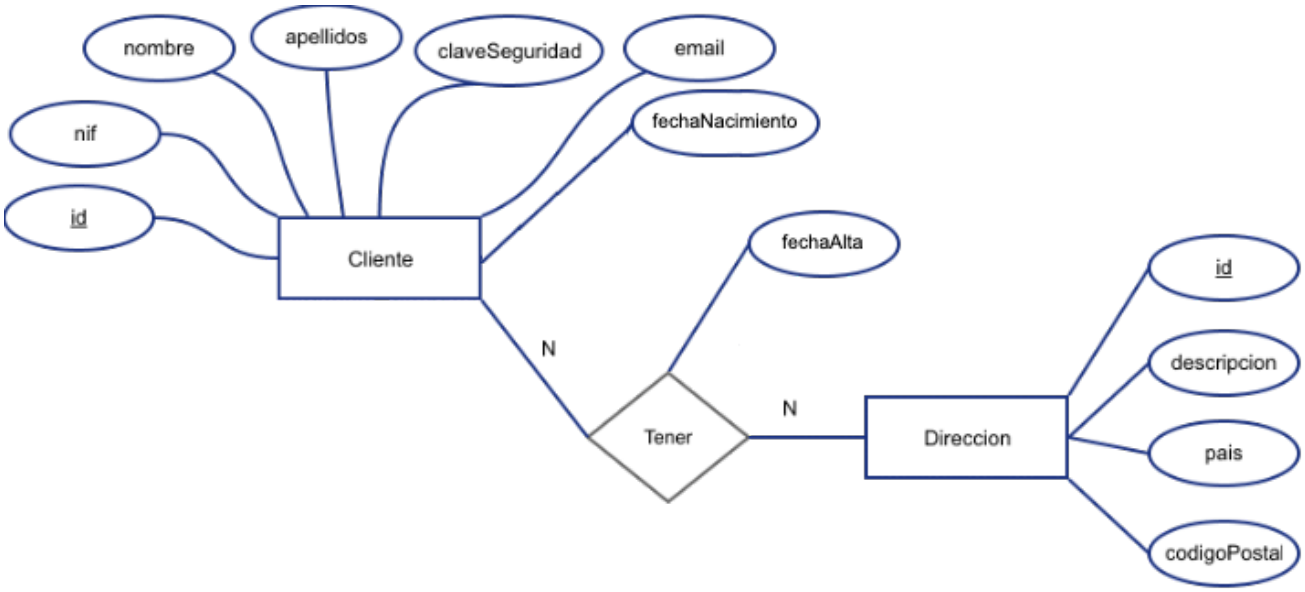
## Relaciones N a N con atributo en medio de la relación y mapeando la entidad del medio. Trabajando con fechas.

Vamos a implementar este tipo de relaciones y para ello, el atributo que existe en medio de la relación, va a ser de tipo fecha. Para trabajar con ellas, destacamos que existen dos tipos de datos enb Javaque hacen referencias a las fechas (aunque hay más):

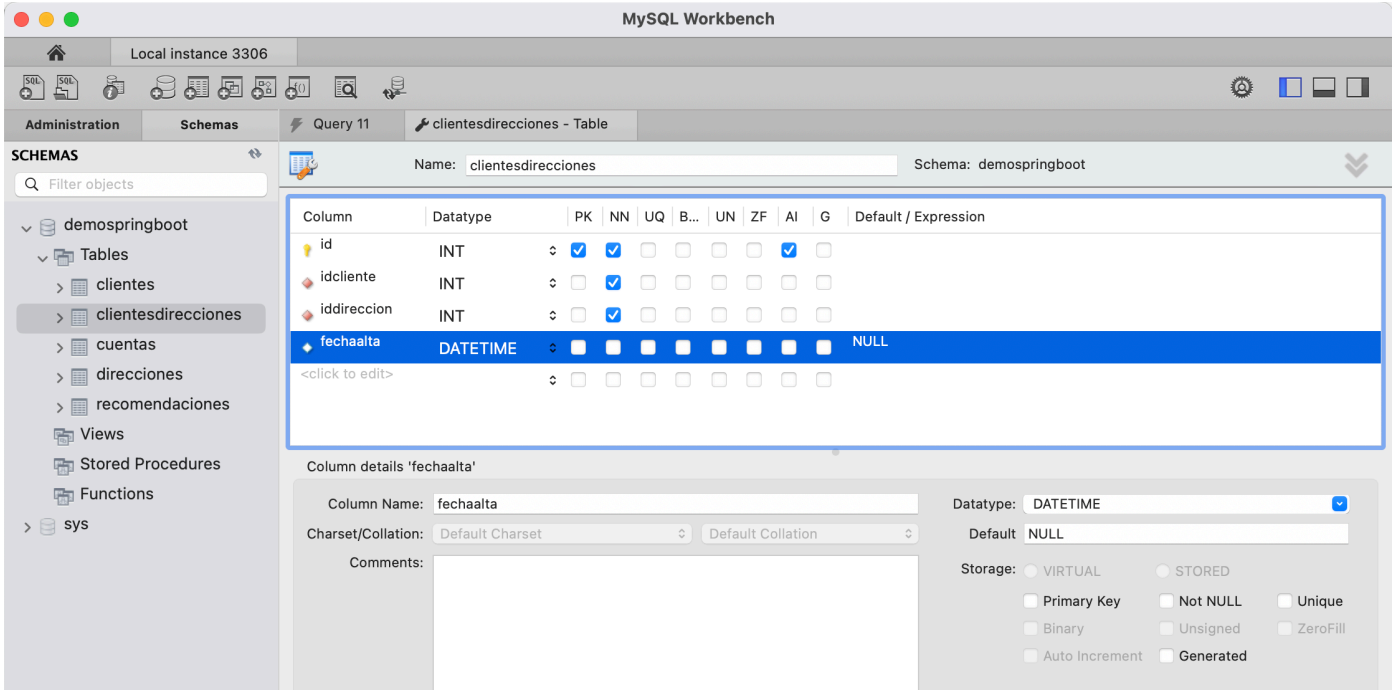
- `DateTime`
- `Date`

Hay que tener en cuenta que estos tipos de datos en Java representan los datos que son almacenados en la base de datos, por lo que tenemos que tener un correcto diseño e implementación de la base de datos.

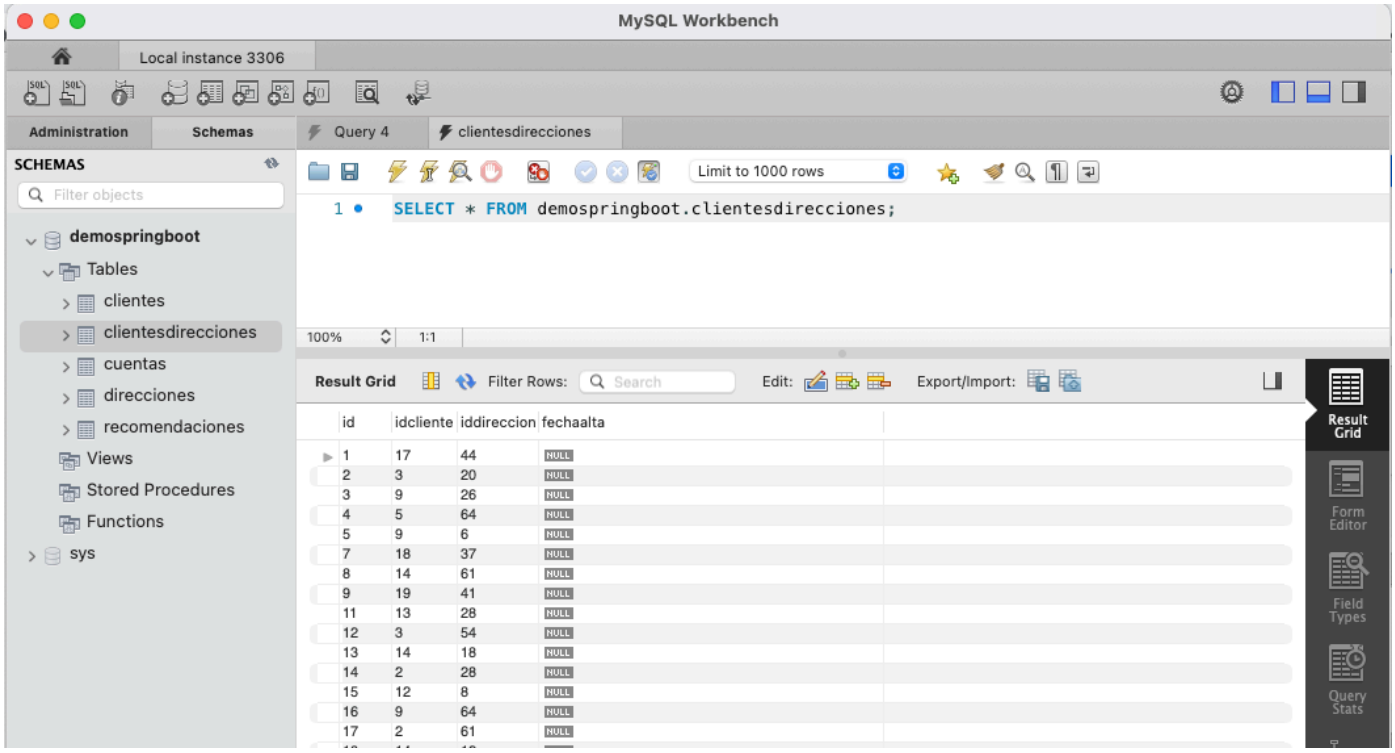
Vamos a comenzar realizando una pequeña modificación en el diagrama entidad relación, de forma que vamos a añadir un campo de fecha a la relación N a N. Además de esto vamos a aprovechar para almacenar una fecha de nacimiento en el cliente.




Esto va a hacer que tengamos que modificar la base de datos, añadiendo dicho campo a la tabla `ClientesDirecciones`.El resultado será el siguiente:



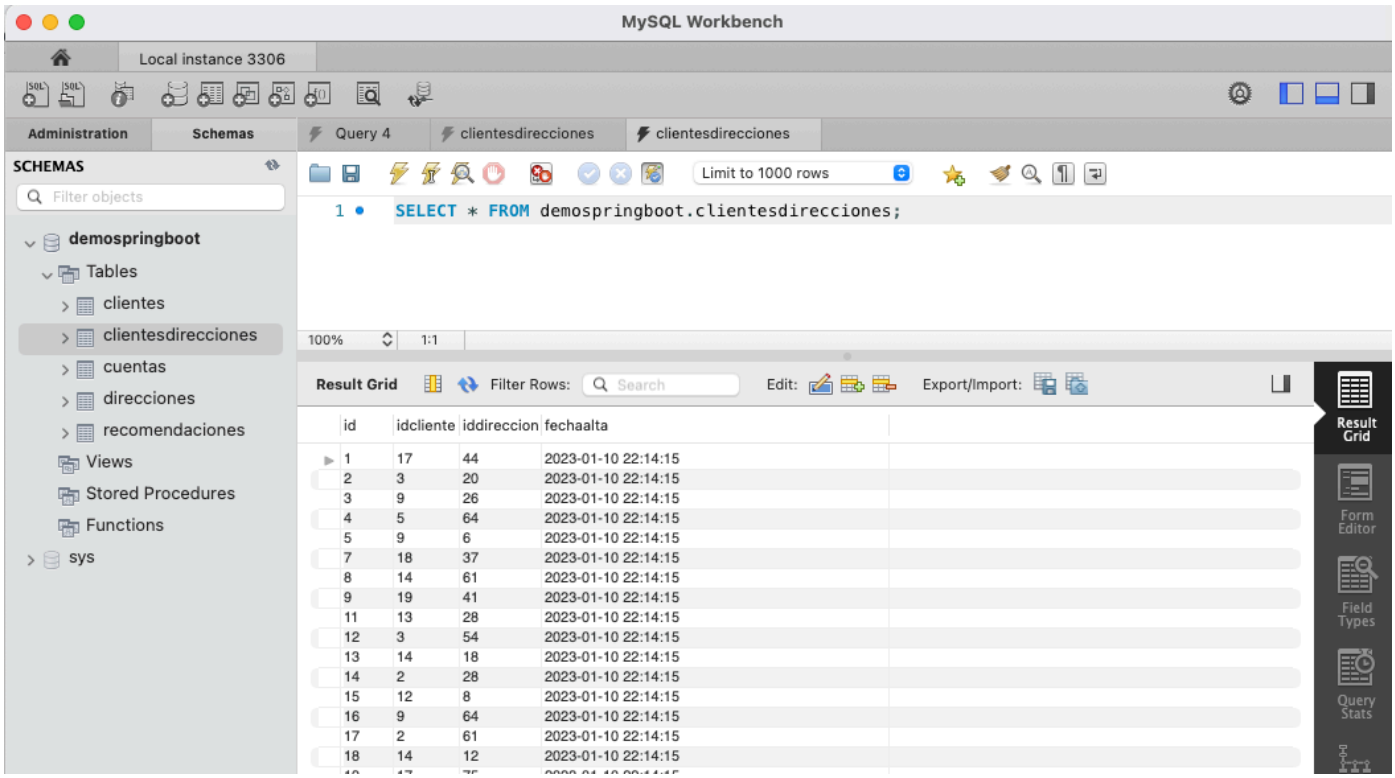

Con esto, si visualizamos los datos de la tabla tenemos lo siguiente:




De esta forma podemos comprobar que ya tenemos la columna creada pero está vacia. Lanzamos la siguiente consulta para asignarle datos y volvemos a comprobar:

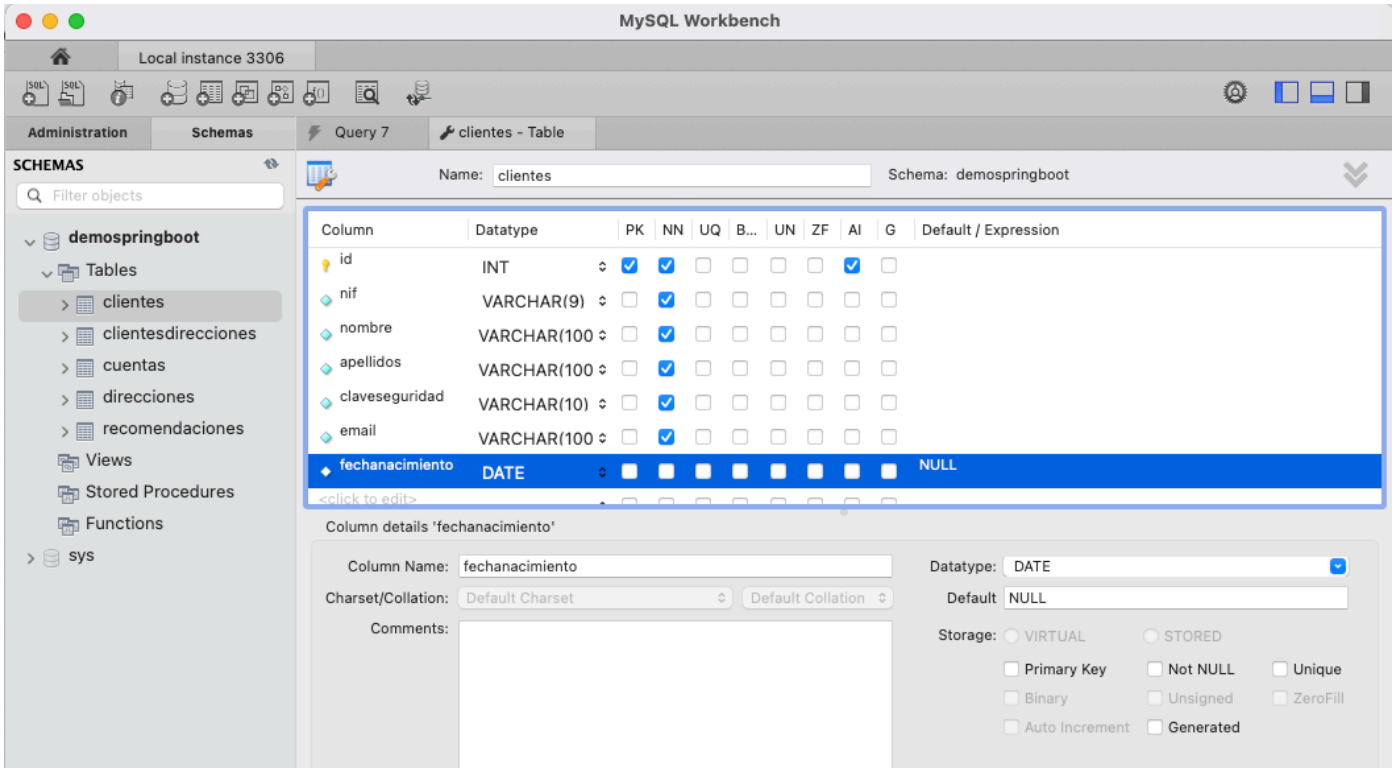
```
UPDATE clientesdirecciones SET fechaalta=now();
```

Con esto conseguimos que todos los registros tengan una fecha de alta por defecto.




Más tarde, cuando demos de alta un registro asignaremos a mano un valor (fijaros que almacenamos fecha y hora). El tipo de datos en Java que vamos a usar para este campo será `DateTime`.

En cuanto a la tabla `clientes` la modificaremos y añadiremos el campo nuevo para contener la fecha de nacimiento:. Quedará de la siguiente forma:




El tipo de datos en Java que vamos a usar para este campo va a ser `Date`.

Tras tener la estructura de la BD ya preparada vamos a pasar a realizar los cambios comenzando por el mapeo de las entidades. Comenzaremos por `Cliente.java` (omitiremos código repetido con anterioridad y nos centraremos solo en lo nuevo que vamos a añadir). Es importante que la clase que importéis para la fecha sea `java.util.Date`:

```
package com.example.demo.repository.entity;

import java.util.Date;
import jakarta.persistence.Temporal;
import jakarta.persistence.TemporalType;

...

@Data
@Entity
@Table(name = "clientes")
public class Cliente {

...

  // Mapeo con la entidad ClienteDireccion con Set
  // IMPORTATE: Poner a LAZY la relacion en este caso, ya que es la relacion N a N
  @OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL, mappedBy = "cliente")
  @ToString.Exclude
  private Set<ClienteDireccion> listaClientesDirecciones;

  // Atributo que almacena la fecha de nacimiento
  @Temporal(TemporalType.DATE)
  @Column(name = "fechanacimiento")
  private Date fechaNacimiento;

...
}
```

Vamos a comentar la anotación que hemos añadido:

- Mediante la anotación `@Temporal` es posible mapear las fechas con la base de datos de una forma simple. Es posible determinar si nuestro atributo almacena Hora, Fecha u Hora y fecha. Se pueden establecer tres posibles valores para la anotación:
  - DATE: Acotara el campo solo a la Fecha, descartando la hora.

    `@Temporal(TemporalType.DATE)`
  - TIME: Acotara el campo solo a la Hora, descartando a la fecha.

    `@Temporal(TemporalType.TIME)`
  - TIMESTAMP: Toma la fecha y hora.

    `@Temporal(TemporalType.TIMESTAMP)`

En cuanto a la clase `ClienteDireccion.java` realizaremos lo mismo:

```java
package com.example.demo.repository.entity;

import java.util.Date;
import jakarta.persistence.Temporal;
import jakarta.persistence.TemporalType;

...

@Data
@Entity
@Table(name = "clientesdirecciones")
public class ClienteDireccion {

  ...

  @ManyToOne
  @JoinColumn(name="iddireccion")
  @ToString.Exclude
  private Direccion direccion;

  @Column(name = "fechaalta")
  @Temporal(TemporalType.TIMESTAMP)
  private Date fechaAlta;

  ...
}
```

La clase `Direccion.java` queda exáctamente igual.

Crearemos la clase `ClienteDireccionDTO.java` que quedará de la siguiente forma:

```java
package com.example.demo.model.dto;

import java.io.Serializable;
import java.util.Date;

import org.springframework.format.annotation.DateTimeFormat;

import com.example.demo.repository.entity.ClienteDireccion;
import lombok.Data;

@Data
public class ClienteDireccionDTO implements Serializable{


  private static final long serialVersionUID = 1L;

  private Long id = 0L;

  private ClienteDTO clienteDTO;
```

```
20
21    private DireccionDTO direccionDTO;
22
23    @DateTimeFormat(pattern = "dd-MM-yyyy HH:mm:ss")
24    private Date fechaAlta;
25
26    public static ClienteDireccionDTO convertToDTO(ClienteDireccion clienteDireccion) {
27      ClienteDireccionDTO cdDTO = new ClienteDireccionDTO();
28      ClienteDTO clienteDTO = ClienteDTO.convertToDTO(clienteDireccion.getCliente());
29      cdDTO.setClienteDTO(clienteDTO);
30
cdDTO.setDireccionDTO(DireccionDTO.convertToDTO(clienteDireccion.getDireccion(),clien
teDTO));
31      cdDTO.setFechaAlta(clienteDireccion.getFechaAlta());
32      return cdDTO;
33    }
34  }
```

Y modificaremos el DTO de cliente, que es `ClienteDTO.java`, de forma que quedará de la siguiente forma:

```
1   package com.example.demo.model.dto;
2
3   import java.io.Serializable;
4   import java.util.ArrayList;
5   import java.util.Date;
6   import java.util.List;
7   import java.util.Optional;
8
9   import org.springframework.format.annotation.DateTimeFormat;
10  import org.springframework.format.annotation.DateTimeFormat.ISO;
11
12  import com.example.demo.repository.entity.Cliente;
13  import com.example.demo.repository.entity.ClienteDireccion;
14  import com.example.demo.repository.entity.Cuenta;
15  import com.example.demo.repository.entity.Direccion;
16  import com.example.demo.repository.entity.Recomendacion;
17
18  import jakarta.persistence.Temporal;
19  import jakarta.persistence.TemporalType;
20  import lombok.Data;
21  import lombok.ToString;
22
23  @Data
24  public class ClienteDTO implements Serializable{
25
26    private static final long serialVersionUID = 1L;
27    private Long id;
28    private String nif;
29    private String nombre;
30    private String apellidos;
```

```java
  private String claveSeguridad;
  private String email;
  @ToString.Exclude
  private RecomendacionDTO recomendacionDTO;
  @ToString.Exclude
  private List<CuentaDTO> listaCuentasDTO;

  //@ToString.Exclude
  //private List<DireccionDTO> listaDireccionesDTO;

  @ToString.Exclude
  private List<ClienteDireccionDTO> listaClientesDireccionesDTO;

  @DateTimeFormat(iso=ISO.DATE)
  private Date fechaNacimiento;

  // Convertir una entidad a un DTO
  public static ClienteDTO convertToDTO(Cliente cliente) {

    ClienteDTO clienteDTO = new ClienteDTO();
    clienteDTO.setId(cliente.getId());
    clienteDTO.setNif(cliente.getNif());
    clienteDTO.setNombre(cliente.getNombre());
    clienteDTO.setApellidos(cliente.getApellidos());
    clienteDTO.setClaveSeguridad(cliente.getClaveSeguridad());
    clienteDTO.setEmail(cliente.getEmail());
    clienteDTO.setFechaNacimiento(cliente.getFechaNacimiento());

    RecomendacionDTO rec = RecomendacionDTO.converToDTO(cliente.getRecomendacion(), clienteDTO);
    clienteDTO.setRecomendacionDTO(rec);

    // Cargamos la lista de cuentas, que como es un HashSet hemos de convertirla a ArrayList
    List<Cuenta> listaCuentas = new ArrayList<Cuenta>(cliente.getListaCuentas());
    for(int i=0;i<listaCuentas.size();i++) {
      CuentaDTO cuentadto = CuentaDTO.convertToDTO(listaCuentas.get(i), clienteDTO);
      clienteDTO.getListaCuentasDTO().add(cuentadto);
    }

    /*
    for(int i=0;i<cliente.getListaDirecciones().size();i++) {
      DireccionDTO direccionDTO = DireccionDTO.convertToDTO(cliente.getListaDirecciones().get(i), clienteDTO);
      clienteDTO.getListaDireccionesDTO().add(direccionDTO);
    }
    */

    // Cargamos la lista de clientes direcciones, que como es un HashSet hemos de convertirla a ArrayList
    List<ClienteDireccion> listaClientesDirecciones = new ArrayList<ClienteDireccion>(cliente.getListaClientesDirecciones());
```

```
    for(int i=0;i<listaClientesDirecciones.size();i++) {
      ClienteDireccionDTO clienteDireccionDTO = new ClienteDireccionDTO();
      clienteDireccionDTO.setClienteDTO(clienteDTO);
clienteDireccionDTO.setDireccionDTO(DireccionDTO.convertToDTO(listaClientesDirecciones.get(i).getDireccion(), clienteDTO));
clienteDireccionDTO.setFechaAlta(listaClientesDirecciones.get(i).getFechaAlta());
      clienteDTO.getListaClientesDireccionesDTO().add(clienteDireccionDTO);
    }

    return clienteDTO;
  }

  // Convertir un DTO a una entidad
  public static Cliente convertToEntity(ClienteDTO clienteDTO) {

    Cliente cliente = new Cliente();
    cliente.setId(clienteDTO.getId());
    cliente.setNif(clienteDTO.getNif());
    cliente.setNombre(clienteDTO.getNombre());
    cliente.setApellidos(clienteDTO.getApellidos());
    cliente.setClaveSeguridad(clienteDTO.getClaveSeguridad());
    cliente.setEmail(clienteDTO.getEmail());
    cliente.setFechaNacimiento(clienteDTO.getFechaNacimiento());

    Recomendacion rec = RecomendacionDTO.convertToEntity(clienteDTO.getRecomendacionDTO(), cliente);
    cliente.setRecomendacion(rec);

    // Cargamos la lista de cuentas
    for(int i=0;i<clienteDTO.getListaCuentasDTO().size();i++) {
      Cuenta cuenta = CuentaDTO.convertToEntity(clienteDTO.getListaCuentasDTO().get(i));
      cliente.getListaCuentas().add(cuenta);
    }
    // Cargamos la lista de ClientesDireccionesDTO
    for(int i=0;i<clienteDTO.getListaClientesDireccionesDTO().size();i++) {
      ClienteDireccion cd = new ClienteDireccion();
      cd.setId(clienteDTO.getListaClientesDireccionesDTO().get(i).getId());
      cd.setCliente(cliente);
cd.setDireccion(DireccionDTO.convertToEntity(clienteDTO.getListaClientesDireccionesDTO().get(i).getDireccionDTO(), cliente));
cd.setFechaAlta(clienteDTO.getListaClientesDireccionesDTO().get(i).getFechaAlta());
      cliente.getListaClientesDirecciones().add(cd);
    }

    return cliente;
  }

```

```
  // Constructor vacio
  public ClienteDTO() {
    super();
    this.recomendacionDTO = new RecomendacionDTO();
    this.listaCuentasDTO = new ArrayList<CuentaDTO>();
    this.listaClientesDireccionesDTO = new ArrayList<ClienteDireccionDTO>();
  }
}
```

La anotación `@DateTimeFormat` nos permite definir el formato de visualización tanto de las fechas como de las horas. Tiene varias opciones, como podemos ver en los DTOs, por lo que destacamos las siguientes opciones:

- `iso`: formateo basado en el estandar ISO. Nos encontramos lo siguiente:
  - `DATE`: El formato ISO más común es yyyy-MM-dd — por ejemplo, "2000-10-31".
  - `DATE_TIME`: El formato ISO más común es yyyy-MM-dd'T'HH:mm:ss.SSSXXX — por ejemplo, "2000-10-31T01:30:00.000-05:00".
  - `NONE`: Indica que no será aplicado ningún patrón basado en ISO.
  - `TIME`: El formato ISO más común es HH:mm:ss.SSSXXX — por ejemplo, "01:30:00.000-05:00"
- `Pattern`: Permite customizar el patron, por ejemplo dd-MM-yyyy HH:mm:ss.

También vamos a destacar que hemos adaptado tanto `convertToDTO()` como `convertToEntity()`.

En cuanto al DTO de dirección, que es `DireccionDTO.java`, quedaría de la siguiente forma:

```
package com.example.demo.model.dto;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import com.example.demo.repository.entity.Cliente;
import com.example.demo.repository.entity.ClienteDireccion;
import com.example.demo.repository.entity.Direccion;

import lombok.Data;
import lombok.ToString;

@Data
public class DireccionDTO implements Serializable {

  private static final long serialVersionUID = 1L;
  private Long id;
  private String descripcion;
  private String pais;
  private String cp;
  @ToString.Exclude
  private List<ClienteDTO> listaClientesDTO;
```

```java

  public static DireccionDTO convertToDTO(Direccion direccion, ClienteDTO clienteDTO) {
    DireccionDTO direccionDTO = new DireccionDTO();
    direccionDTO.setId(direccion.getId());
    direccionDTO.setDescripcion(direccion.getDescripcion());
    direccionDTO.setPais(direccion.getPais());
    direccionDTO.setCp(direccion.getCp());

    // No tiene sentido mapear todos los clientes que tiene la direccion, puesto que
    // al mapear cada cliente volveriamos a mapear sus direcciones, y así
    // sucesivamente.
    direccionDTO.getListaClientesDTO().add(clienteDTO);

    return direccionDTO;
  }

  public static Direccion convertToEntity(DireccionDTO direccionDTO, Cliente cliente) {
    Direccion direccion = new Direccion();
    direccion.setId(direccionDTO.getId());
    direccion.setDescripcion(direccionDTO.getDescripcion());
    direccion.setPais(direccionDTO.getPais());
    direccion.setCp(direccionDTO.getCp());

    // No tiene sentido mapear todos los clientes que tiene la direccion, puesto que
    // al mapear cada cliente volveriamos a mapear sus direcciones, y así
    // sucesivamente.
    //direccion.getListaClientes().add(cliente);

    // Creamos una entidad de ClienteDireccion y asignamos la relacion de ambos.
    ClienteDireccion cd = new ClienteDireccion();
    cd.setCliente(cliente);
    cd.setDireccion(direccion);
    direccion.getListaClientesDirecciones().add(cd);
    cliente.getListaClientesDirecciones().add(cd);

    return direccion;

  }

  // Constructor vacio
  public DireccionDTO() {
    super();
    this.listaClientesDTO = new ArrayList<ClienteDTO>();
  }
}
```

Volvemos a destacar que hemos adaptado tanto `convertToDTO()` como `convertToEntity()` a los cambios requeridos.

Como ya tenemos las entidades y los DTOs podemos pasar a modificar los controladores, de forma que el controlador de clientes, que es `ClienteController.java`, quedará igual que tenemos ahora mismo desarrollado.

En cuanto al controlador de direcciones, que es `DireccionController.java`, quedará de la siguiente forma:

```
package com.example.demo.web.controller;

import java.sql.Timestamp;
import java.time.Instant;
import java.util.Date;
import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.servlet.ModelAndView;

import com.example.demo.model.dto.ClienteDTO;
import com.example.demo.model.dto.ClienteDireccionDTO;
import com.example.demo.model.dto.DireccionDTO;
import com.example.demo.service.ClienteService;
import com.example.demo.service.DireccionService;

@Controller
public class DireccionController {

  private static final Logger log =
LoggerFactory.getLogger(DireccionController.class);

  @Autowired
  private DireccionService direccionService;

  @Autowired
  private ClienteService clienteService;

  @GetMapping("/clientes/{idCliente}/direcciones")
  public ModelAndView findAllByCliente(@PathVariable Long idCliente) {

    log.info("DireccionController - findAllByCliente: Listamos las direcciones del
cliente: " + idCliente);

    // Obtenemos el cliente para luego poner sus datos en la pantalla
    ClienteDTO clienteDTO = new ClienteDTO();
    clienteDTO.setId(idCliente);
    clienteDTO = clienteService.findById(clienteDTO);
```

```
    // Obtenemos la lista de direcciones del cliente
    List<ClienteDireccionDTO> listaClientesDireccionesDTO = direccionService.findAllByCliente(clienteDTO);
    // Pasamos los datos al modelo
    ModelAndView mav = new ModelAndView("direcciones");
    mav.addObject("listaClientesDireccionesDTO", listaClientesDireccionesDTO);
    mav.addObject("clienteDTO", clienteDTO);
    // retornamos el ModelAndView
    return mav;
  }

  @GetMapping("/clientes/{idCliente}/direcciones/add")
  public ModelAndView add(@PathVariable Long idCliente) {

    log.info("DireccionController - add: Anyadimos una nueva direccion al cliente: " + idCliente);

    ClienteDTO clienteDTO = new ClienteDTO();
    clienteDTO.setId(idCliente);
    clienteDTO = clienteService.findById(clienteDTO);
    // Pasamos los datos al modelo
    ModelAndView mav = new ModelAndView("direccionform");
    mav.addObject("direccionDTO", new DireccionDTO());
    mav.addObject("clienteDTO", clienteDTO);
    // retornamos el ModelAndView
    return mav;
  }

  @PostMapping("/clientes/{idCliente}/direcciones/save")
  public ModelAndView save(@PathVariable Long idCliente, @ModelAttribute("direccionDTO") DireccionDTO direccionDTO) {

    log.info("DireccionController - save: Almacenamos la direccion " + direccionDTO.toString());

    ClienteDTO clienteDTO = new ClienteDTO();
    clienteDTO.setId(idCliente);
    clienteDTO = clienteService.findById(clienteDTO);

    // Anyadimos a la direccion el cliente. En este momento solo habra un clienteDTO
    direccionDTO.getListaClientesDTO().add(clienteDTO);
    direccionService.save(direccionDTO);
    // Redireccionamos para volver a invocar el metodo que escucha /clientes
    ModelAndView mav = new ModelAndView("redirect:/clientes/{idCliente}/direcciones");
    return mav;
  }
}
```

En cuanto a la capa de servicios, nos encontramos que hemos modificado la interfaz `DireccionService.java`, quedando de la siguiente forma:

```
package com.example.demo.service;

import java.util.List;

import com.example.demo.model.dto.ClienteDTO;
import com.example.demo.model.dto.ClienteDireccionDTO;
import com.example.demo.model.dto.DireccionDTO;

public interface DireccionService {

  //public List<DireccionDTO> findAllByCliente(ClienteDTO clienteDTO);
  public List<ClienteDireccionDTO> findAllByCliente(ClienteDTO clienteDTO);
  public void save(DireccionDTO direccionDTO) ;

}
```

En cuanto a su implementación, que es `DireccionServiceImpl.java` queda de la siguiente forma:

```
package com.example.demo.service;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.Optional;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.example.demo.model.dto.ClienteDTO;
import com.example.demo.model.dto.ClienteDireccionDTO;
import com.example.demo.model.dto.DireccionDTO;
import com.example.demo.repository.dao.ClienteDireccionRepository;
import com.example.demo.repository.dao.ClienteRepository;
import com.example.demo.repository.dao.DireccionRepository;
import com.example.demo.repository.entity.Cliente;
import com.example.demo.repository.entity.ClienteDireccion;
import com.example.demo.repository.entity.Direccion;

@Service
public class DireccionServiceImpl implements DireccionService {

  private static final Logger log =
LoggerFactory.getLogger(DireccionServiceImpl.class);

```

```
  @Autowired
  private DireccionRepository direccionRepository;

  @Autowired
  private ClienteRepository clienteRepository;

  @Autowired
  private ClienteDireccionRepository clienteDireccionRepository;

  /*
  @Override
  public List<DireccionDTO> findAllByCliente(ClienteDTO clienteDTO) {

    log.info("DireccionServiceImpl - findAllByCliente: Lista de todas las direcciones del cliente: "
        + clienteDTO.getId());

    // Obtenemos la lista de direcciones del cliente
    List<Direccion> lista = (List<Direccion>) direccionRepository.findAllByCliente(clienteDTO.getId());
    // Creamos una lista de DireccionDTO que será la que devolvamos al controlador
    List<DireccionDTO> listaDireccionesDTO = new ArrayList<DireccionDTO>();
    // Recorremos la lista de direcciones y las mapeamos a DTO
    for (int i = 0; i < lista.size(); ++i) {
      listaDireccionesDTO.add(DireccionDTO.convertToDTO(lista.get(i), clienteDTO));
    }
    // Devolvemos la lista de DTO's
    return listaDireccionesDTO;


    /*
    // Obtenemos la lista de direcciones del cliente
    List<ClienteDireccion> lista = clienteDireccionRepository.findAllByCliente(clienteDTO.getId());
    // Creamos una lista de DireccionDTO que será la que devolvamos al controlador
    List<ClienteDireccionDTO> listaClientesDireccionesDTO = new ArrayList<ClienteDireccionDTO>();
    // Recorremos la lista de direcciones y las mapeamos a DTO
    for (int i = 0; i < lista.size(); ++i) {
      listaClientesDireccionesDTO.add(ClienteDireccionDTO.convertToDTO(lista.get(i)));
    }
    // Devolvemos la lista de DTO's
    return listaClientesDireccionesDTO;

    */
/*
  }
*/

  @Override
  public List<ClienteDireccionDTO> findAllByCliente(ClienteDTO clienteDTO) {
```

```
75
76      log.info("DireccionServiceImpl - findAllByCliente: Lista de todas las
direcciones del cliente: "
77          + clienteDTO.getId());
78
79      // Obtenemos la lista de direcciones del cliente
80      List<ClienteDireccion> lista =
clienteDireccionRepository.findAllByCliente(clienteDTO.getId());
81      // Creamos una lista de DireccionDTO que será la que devolvamos al controlador
82      List<ClienteDireccionDTO> listaClientesDireccionesDTO = new
ArrayList<ClienteDireccionDTO>();
83      // Recorremos la lista de direcciones y las mapeamos a DTO
84      for (int i = 0; i < lista.size(); ++i) {
85
listaClientesDireccionesDTO.add(ClienteDireccionDTO.convertToDTO(lista.get(i)));
86      }
87      // Devolvemos la lista de DTO's
88      return listaClientesDireccionesDTO;
89
90    }
91
92    @Override
93    public void save(DireccionDTO direccionDTO) {
94
95      log.info("DireccionServiceImpl - save: Salva la direccion del cliente: "
96          + direccionDTO.getListaClientesDTO().get(0).getId());
97
98      /*
99      // Solo hay un clienteDTO en la direccionDTO
100     Optional<Cliente> cliente =
clienteRepository.findById(direccionDTO.getListaClientesDTO().get(0).getId());
101     if (cliente.isPresent()) {
102       // Creamos las relaciones en las entidades
103       Direccion direccion = DireccionDTO.convertToEntity(direccionDTO,
cliente.get());
104       cliente.get().getListaDirecciones().add(direccion);
105       // Almacenamos la direccion, y por la relacion que tiene, almacenará la
relacion
106       // N a N
107       direccionRepository.save(direccion);
108     }
109      */
110
111     /*
112     // Solo hay un clienteDTO en la direccionDTO. En este caso tendremos que creare
la
113     // entidad ClienteDireccion y asignarle las relaciones.
114     Optional<Cliente> cliente =
clienteRepository.findById(direccionDTO.getListaClientesDTO().get(0).getId());
115     if (cliente.isPresent()) {
116       // Creamos las relaciones en las entidades
```

```
      Direccion direccion = DireccionDTO.convertToEntity(direccionDTO, cliente.get());
      ClienteDireccion cd = new ClienteDireccion();
      cd.setDireccion(direccion);
      cd.setCliente(cliente.get());
      // Almacenamos la direccion, y por la relacion que tiene, almacenará la relacion
      // N a N
      direccionRepository.save(direccion);
    }
    */

    // Obtenemos la entidad cliente
    Optional<Cliente> cliente = clienteRepository.findById(direccionDTO.getListaClientesDTO().get(0).getId());

    // Persistimos la entidad direccion
    Direccion direccion = DireccionDTO.convertToEntity(direccionDTO, cliente.get());
    direccionRepository.save(direccion);

    // Hay que almacenar la relacion intermedia, que es ClienteDireccion, ya sea nueva o una actualizacion
    ClienteDireccion cd = null;
    Optional<ClienteDireccion> cdo = clienteDireccionRepository.findAllByClienteDireccion(cliente.get().getId(), direccion.getId());
    if(cdo.isPresent()) {
      cd = cdo.get();
      cd.setFechaAlta(new Date());
    }else {
      cd = new ClienteDireccion();
      cd.setCliente(cliente.get());
      cd.setDireccion(direccion);
      cd.setFechaAlta(new Date());
    }
    clienteDireccionRepository.save(cd);
  }
}
```

Ya solo nos falta crear en los repositorios el repositorio `ClienteDireccionRepository.java`, donde se creará 2 métodos: uno para obtener una lista de `ClienteDireccion` a partir del cliente y uno para obtener un objeto `ClienteDireccion` a partir del cliente y de la direccion. Quedará de la siguiente forma:

```
package com.example.demo.repository.dao;

import java.util.List;
import java.util.Optional;

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

```
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import com.example.demo.repository.entity.ClienteDireccion;

import jakarta.transaction.Transactional;

@Repository
@Transactional
public interface ClienteDireccionRepository extends JpaRepository<ClienteDireccion, Long>{

  @Query(value = "SELECT c FROM ClienteDireccion c WHERE c.cliente.id = :idc AND c.direccion.id = :idd")
  public Optional<ClienteDireccion> findAllByClienteDireccion(@Param("idc") Long idCliente, @Param("idd") Long idDireccion);

  @Query(value = "SELECT c FROM ClienteDireccion c WHERE c.cliente.id = :idcliente")
  public List<ClienteDireccion> findAllByCliente(@Param("idcliente") Long idCliente);
}
```

Con esto ya tendremos la parte de Java terminada. Solo nos queda la parte de la visualización. Por ello vamos a comenzar modificando la vista `clientes.html` de forma que vamos a añadir la fecha de nacimiento. Quedará de la siguiente forma:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <title>Mantenimiento de clientes</title>
</head>
<body>
  <div>
    <br>
    <h2 th:if="${listaClientesDTO.isEmpty()}">No hay clientes</h2>
    <div th:if="${!listaClientesDTO.isEmpty()}">
      <h2>Mantenimiento de clientes</h2>
      <table border="1">
        <thead>
          <tr>
            <td>id</td>
            <td>Nombre</td>
            <td>Apellidos</td>
            <td>NIF</td>
            <td>Correo electrónico</td>
            <td>Fecha nacimiento</td>
            <td>Editar</td>
            <td>Eliminar</td>
```

```
24              <th>Cuentas</th>
25              <th>Direcciones</th>
26            </tr>
27          </thead>
28          <tbody>
29            <tr th:each="clienteDTO : ${listaClientesDTO}">
30              <td><a
31                th:href="@{/clientes/{idCliente}(idCliente=${clienteDTO.id})}"
32                th:text="${clienteDTO.id}"></a></td>
33              <td th:text="${clienteDTO.nombre}" />
34              <td th:text="${clienteDTO.apellidos}" />
35              <td th:text="${clienteDTO.nif}" />
36              <td th:text="${clienteDTO.email}" />
37              <td th:text="${#dates.format(clienteDTO.fechaNacimiento, 'dd/MM/yyyy')}"
/>
38              <td><a
39                th:href="@{/clientes/update/{idCliente}(idCliente=${clienteDTO.id})}"
/>Editar</a></td>
40              <td><a
41                th:href="@{/clientes/delete/{idCliente}(idCliente=${clienteDTO.id})}"
/>Eliminar</a></td>
42              <td><a
43                th:href="@{/clientes/{idCliente}/cuentas
(idCliente=${clienteDTO.id})}">Cuentas</a></td>
44              <td><a
45                th:href="@{/clientes/{idCliente}/direcciones
(idCliente=${clienteDTO.id})}">Direcciones</a></td>
46            </tr>
47          </tbody>
48        </table>
49      </div>
50    </div>
51    <br>
52    <div>
53      <a th:href="@{/clientes/add}">Nuevo cliente</a>
54      <br>
55      <a th:href="@{/}">Volver al inicio</a>
56    </div>
57  </body>
58  </html>
```

Resaltamos que es posible modificar el formato de visualización de las fechas mediante `#dates.format`.

Modificaremos también la vista `clienteform.html`, de forma que quedará de la siguiente forma:

```
1  <!DOCTYPE html>
2  <html xmlns:th="http://www.thymeleaf.org">
3  <head>
4  <meta charset="UTF-8">
5  <title th:text="${add} ? 'Nuevo cliente' : 'Actualizar cliente'"></title>
```

```
6  </head>
7  <body>
8    <h1 th:text="${add} ? 'Nuevo cliente' : 'Actualizar cliente'" />
9    <br>
10   <br>
11   <form th:action="@{/clientes/save}" th:object="${clienteDTO}"
12     method="POST">
13     <table border="0">
14       <tr th:if="${clienteDTO.id}">
15         <td>id</td>
16         <td>:</td>
17         <td> <input type="text" th:field="*{id}" readonly/> </td>
18       </tr>
19       <tr>
20         <td>Nombre</td>
21         <td>:</td>
22         <td> <input type="text" th:field="*{nombre}" /> </td>
23       </tr>
24       <tr>
25         <td>Apellidos</td>
26         <td>:</td>
27         <td> <input type="text" th:field="*{apellidos}" /> </td>
28       </tr>
29       <tr>
30         <td>NIF</td>
31         <td>:</td>
32         <td> <input type="text" th:field="*{nif}" /> </td>
33       </tr>
34       <tr>
35         <td>Correo electrónico</td>
36         <td>:</td>
37         <td> <input type="text" th:field="*{email}" /> </td>
38       </tr>
39       <tr>
40         <td>Clave seguridad</td>
41         <td>:</td>
42         <td> <input type="text" th:field="*{claveSeguridad}" /> </td>
43       </tr>
44       <tr>
45         <td>Fecha Nacimiento</td>
46         <td>:</td>
47         <td> <input type="date" th:field="*{fechaNacimiento}" /> </td>
48       </tr>
49       <tr>
50         <td>Recomendación</td>
51         <td>:</td>
52         <td> <textarea th:field="*{recomendacionDTO.observaciones}" rows="4"
cols="50" /> </td>
53       </tr>
54
55       <!-- Ocultamos el id de la recomendacion del cliente -->
56       <input type="hidden" th:field="*{recomendacionDTO.id}">
```

```
57
58      </table>
59      <input type="submit" th:value="${add} ? 'Nuevo' : 'Actualizar'" />
60    </form>
61    <br>
62    <a th:href="@{/clientes}">Volver a la lista de clientes</a>
63  </body>
64  </html>
```

Y la vista `clienteshow.html` que quedará de la siguiente forma:

```
1   <!DOCTYPE html>
2   <html xmlns:th="http://www.thymeleaf.org">
3   <head>
4   <meta charset="UTF-8">
5   <title>Visualizar cliente</title>
6   </head>
7   <body>
8     <h1>Visualizar cliente</h1>
9     <br>
10    <table border="0">
11      <tr>
12        <td>id</td>
13        <td>:</td>
14        <td th:text="${clienteDTO.id}"></td>
15      </tr>
16      <tr>
17        <td>Nombre</td>
18        <td>:</td>
19        <td th:text="${clienteDTO.nombre}"></td>
20      </tr>
21      <tr>
22        <td>Apellidos</td>
23        <td>:</td>
24        <td th:text="${clienteDTO.apellidos}"></td>
25      </tr>
26      <tr>
27        <td>NIF</td>
28        <td>:</td>
29        <td th:text="${clienteDTO.nif}"></td>
30      </tr>
31      <tr>
32        <td>Correo electrónico</td>
33        <td>:</td>
34        <td th:text="${clienteDTO.email}"></td>
35      </tr>
36      <tr>
37        <td>Clave seguridad</td>
38        <td>:</td>
39        <td th:text="${clienteDTO.claveSeguridad}"></td>
```

```
    </tr>
    <tr>
      <td>Fecha Nacimiento</td>
      <td>:</td>
      <td th:text="${#dates.format(clienteDTO.fechaNacimiento, 'dd/MM/yyyy')}"></td>
    </tr>
    <tr>
      <td>Recomendación</td>
      <td>:</td>
      <td th:text="${clienteDTO.recomendacionDTO.observaciones}"></td>
    </tr>
  </table>
  <br>
  <br>
  <a th:href="@{/clientes}">Volver a la lista de clientes</a>
</body>
</html>
```

Modificamos la vista `direcciones.html` para adaptarla a la clase `ClienteDireccion.html` de forma que quedará así:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <title>Mantenimiento de direcciones</title>
</head>
<body>
  <div>
    <br>
    <h2 th:if="${listaClientesDireccionesDTO.isEmpty()}">No hay direcciones para el cliente
      <td th:utext="${clienteDTO.nombre}"></td> <td th:utext="${clienteDTO.apellidos}"></td>
    </h2>
    <div th:if="${!listaClientesDireccionesDTO.isEmpty()}">
      <h2>Mantenimiento de direcciones del cliente
        <td th:utext="${clienteDTO.nombre}"></td> <td th:utext="${clienteDTO.apellidos}"></td>
      </h2>
      <br>
      <table border="1">
        <thead>
          <tr>
            <th>id</th>
            <th>Direccion</th>
            <th>País</th>
            <th>CP</th>
            <th>Fecha Alta</th>
          </tr>
```

```
27          </thead>
28          <tbody>
29            <tr th:each="clienteDireccionDTO : ${listaClientesDireccionesDTO}">
30              <td th:utext="${clienteDireccionDTO.direccionDTO.id}">...</td>
31              <td th:utext="${clienteDireccionDTO.direccionDTO.descripcion}">...</td>
32              <td th:utext="${clienteDireccionDTO.direccionDTO.pais}">...</td>
33              <td th:utext="${clienteDireccionDTO.direccionDTO.cp}">...</td>
34              <!--  formateamos la fecha para verla en formato correcto y quitamos los
segundos y milisegundos -->
35                <td th:text="${#dates.format(clienteDireccionDTO.fechaAlta, 'dd-MM-yyyy
HH:mm')}">...</td>
36            </tr>
37          </tbody>
38        </table>
39      </div>
40      <br>
41      <div>
42        <a th:href="@{/clientes/{idCliente}/direcciones/add
(idCliente=${clienteDTO.id})}" >Nueva direccion</a>
43        </br>
44        <a th:href="@{/clientes/{idCliente}/direcciones/addselect
(idCliente=${clienteDTO.id})}" >Añadir dirección ya existente</a>
45        </br>
46        <a th:href="@{/clientes}">Volver a la lista de clientes</a>
47      </div>
48    </div>
49  </body>
50  </html>
```

Tras todo esto, si probamos la aplicación nos encontramos que ya está la relación N a N funcionando correctamente y con las fechas implementadas:

Mantenimiento de clientes

localhost:8888/clientes

# Mantenimiento de clientes

| id | Nombre | Apellidos | NIF | Correo electrónico | Fecha nacimiento | Editar | Eliminar | **Cuentas** | **Direcciones** |
|---|---|---|---|---|---|---|---|---|---|
| 1 | Keiko | Meadows | 44999594K | meadowskeiko6933@icloud.net | 03/01/2023 | Editar | Eliminar | Cuentas | Direcciones |
| 2 | Whoopi | Phelps | 28553744Y | w-phelps@aol.ca | 12/01/2023 | Editar | Eliminar | Cuentas | Direcciones |
| 3 | Hilary | Mcmahon | 25861374E | hmcmahon6589@google.org | 30/12/2022 | Editar | Eliminar | Cuentas | Direcciones |
| 4 | Hunter | Frederick | 12242423G | h_frederick@icloud.edu | | Editar | Eliminar | Cuentas | Direcciones |
| 5 | Diana | Dickerson | 51724739D | d-dickerson5965@aol.net | 05/01/2023 | Editar | Eliminar | Cuentas | Direcciones |
| 6 | Amber | Ramirez | 13176696D | amber.ramirez6505@protonmail.org | | Editar | Eliminar | Cuentas | Direcciones |
| 7 | Chava | Floyd | 82568848H | cfloyd@hotmail.couk | 10/01/2023 | Editar | Eliminar | Cuentas | Direcciones |
| 8 | Tatum | Walsh | 03768520L | walsh_tatum2491@outlook.couk | | Editar | Eliminar | Cuentas | Direcciones |
| 9 | Ezekiel | Woodard | 51976869H | e-woodard@icloud.couk | | Editar | Eliminar | Cuentas | Direcciones |
| 10 | Coby | Lester | 55172493N | c-lester7681@hotmail.couk | | Editar | Eliminar | Cuentas | Direcciones |
| 11 | Brielle | Howard | 48037186R | b_howard@aol.com | | Editar | Eliminar | Cuentas | Direcciones |
| 12 | Candace | Bates | 71227610F | c-bates@google.org | | Editar | Eliminar | Cuentas | Direcciones |
| 13 | Eagan | Carpenter | 41124626J | e_carpenter9768@outlook.org | | Editar | Eliminar | Cuentas | Direcciones |
| 14 | Yvette | Slater | 86926685Y | yslater@google.edu | | Editar | Eliminar | Cuentas | Direcciones |
| 15 | Heather | Abbott | 18829162J | abbottheather@hotmail.org | | Editar | Eliminar | Cuentas | Direcciones |
| 16 | Kessie | Rich | 10159358C | k_rich804@google.net | | Editar | Eliminar | Cuentas | Direcciones |
| 17 | Keelie | Forbes | 61816073X | k-forbes1767@hotmail.com | | Editar | Eliminar | Cuentas | Direcciones |
| 18 | Dale | Glass | 27682739N | gdale@google.ca | | Editar | Eliminar | Cuentas | Direcciones |
| 19 | Farrah | Farmer | 91124511M | farrah-farmer6033@hotmail.com | | Editar | Eliminar | Cuentas | Direcciones |
| 20 | Graham | Casey | 21865746E | g.casey3482@google.com | 31/08/2022 | Editar | Eliminar | Cuentas | Direcciones |

Nuevo cliente
Volver al inicio

Actualizar cliente

localhost:8888/clientes/update/4

# Actualizar cliente

id : 4

Nombre : Hunter

Apellidos : Frederick

NIF : 12242423G

Correo electrónico : h_frederick@icloud.edu

Clave seguridad : 1332

Fecha Nacimiento : dd/mm/aaaa

Recomendación : tus urna convallis

Actualizar

Volver a la lista de cl

enero de 2023

| L | M | X | J | V | S | D |
|---|---|---|---|---|---|---|
| 26 | 27 | 28 | 29 | 30 | 31 | 1 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 |
| 23 | 24 | 25 | 26 | 27 | 28 | 29 |
| 30 | 31 | 1 | 2 | 3 | 4 | 5 |

Borrar Hoy

Mantenimiento de direcciones

localhost:8888/clientes/5/direcciones

# Mantenimiento de direcciones del cliente Diana Dickerson

| id | Direccion | País | CP | Fecha Alta |
|---|---|---|---|---|
| 3 | Ap #282-2375 Donec Street | Austria | 35795 | 10-01-2023 23:14 |
| 6 | 204-4479 Nullam Rd. | Germany | 84777 | 10-01-2023 23:14 |
| 13 | 511-9481 Interdum. St. | Ireland | 58771 | 10-01-2023 23:14 |
| 21 | 906-9945 Laoreet Avenue | Sweden | 28139 | 10-01-2023 23:14 |
| 37 | Ap #176-9738 Phasellus Rd. | Poland | 22001 | 10-01-2023 23:14 |
| 41 | 238-733 Aenean Road | Turkey | 38880 | 10-01-2023 23:14 |
| 43 | Ap #270-8384 Mollis St. | India | 17133 | 10-01-2023 23:14 |
| 46 | Ap #127-8719 Quis Avenue | Turkey | 87532 | 10-01-2023 23:14 |
| 48 | P.O. Box 548, 7423 Vel, Av. | Philippines | 09759 | 10-01-2023 23:14 |
| 64 | 418-9258 Nec St. | South Africa | 34213 | 10-01-2023 23:14 |
| 76 | P.O. Box 489, 4632 Nulla. Avenue | Ukraine | 38571 | 10-01-2023 23:14 |
| 77 | 530-9117 Metus Avenue | Peru | 74509 | 10-01-2023 23:14 |

Nueva direccion
Añadir dirección ya existente
Volver a la lista de clientes