

SPRING: CRUD MODELO-VISTA-CONTROLADOR

Por Mikel Jorge y Eneko Martinez

Antes de empezar...

Realiza las actividades guiadas [UT4_A2 CRUD Básico](#) y [UT4_A3 Formulario](#) para tener un conocimiento básico sobre cómo conectar nuestra aplicación a una base de datos y realizar operaciones sobre ella mediante vistas y controladores.

Crea con workbench una base de datos **vacía** llamada **bdgestionventas** que usaremos para almacenar la tabla que creemos.

¿Qué queremos que haga nuestro programa?

Vamos a crear una aplicación que las dé acceso a una tabla de una base de datos y nos permita hacer todas las funciones CRUD básicas.

Creación del proyecto y puesta en marcha

Vamos a crear el proyecto usando la herramienta web que proporciona el propio framework y posteriormente la importaremos en IntelliJ para trabajar con ella.

Primero nos conectaremos a start.spring.io y crearemos nuestro proyecto. Marcaremos el tipo de proyecto como Maven y el lenguaje de programación Java. La versión de Spring Boot 3.4.2 (última estable) y Java 23 y ponemos un nombre acorde a nuestro proyecto.

Por último, debemos añadir las dependencias de nuestro proyecto a la derecha. En esta ocasión añadiremos **Spring Web**, **Spring Boot Dev Tools**, **Spring Data JPA**, **MySQL Driver** y **Thymeleaf**.

The screenshot shows the Spring Boot Start web interface. On the left, under 'Project', 'Gradle - Kotlin' is selected. Under 'Spring Boot', '3.4.2' is selected. Under 'Project Metadata', the 'Artifact' and 'Name' fields are both 'UT4_A4_CRUD', and the 'Package name' is '.UT4_A4_CRUD'. Under 'Packaging', 'Jar' is selected, and under 'Java', '23' is selected. On the right, under 'Dependencies', a list of dependencies is shown: 'Spring Web' (WEB), 'Spring Data JPA' (SQL), 'MySQL Driver' (SQL), 'Thymeleaf' (TEMPLATE ENGINES), and 'Spring Boot DevTools' (DEVELOPER TOOLS). Each dependency has a brief description. A button 'ADD DEPENDENCIES... CTRL + B' is at the top right of the dependencies section.

Una vez esté todo a nuestro gusto generamos el proyecto en la parte de abajo y nos descargará un archivo .zip con nuestro proyecto.

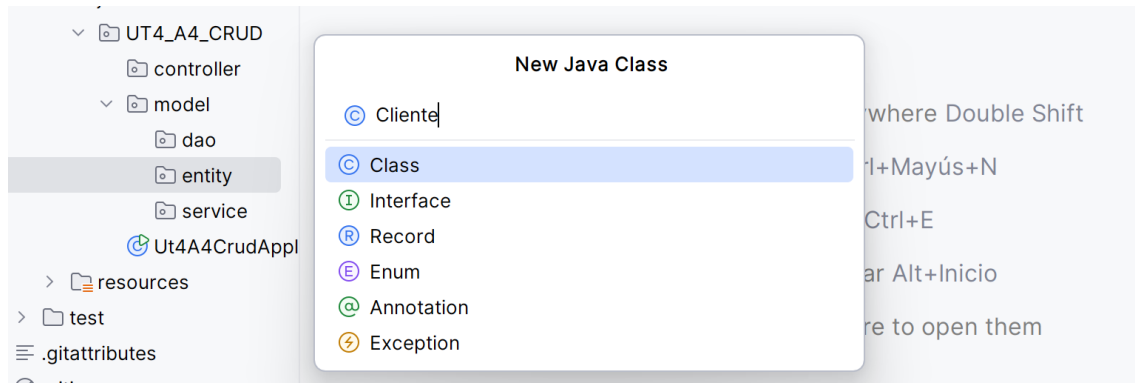
Guardamos y descomprimos el archivo en el lugar donde deseemos que esté nuestro proyecto. Seguidamente tendremos que importar el proyecto generado a IntelliJ.

Abrimos IntelliJ y creamos un nuevo proyecto a partir de uno ya existente como hemos hecho anteriormente.

Creación del modelo

Primero vamos a crear la clase encargada de modelar la tabla de la base de datos, según el esquema que se muestra a continuación.

Como vimos en anteriores prácticas, creamos diferentes paquetes para el modelo y sus clases y para el controlador. Dentro de *entity* crearemos la clase *Cliente*.

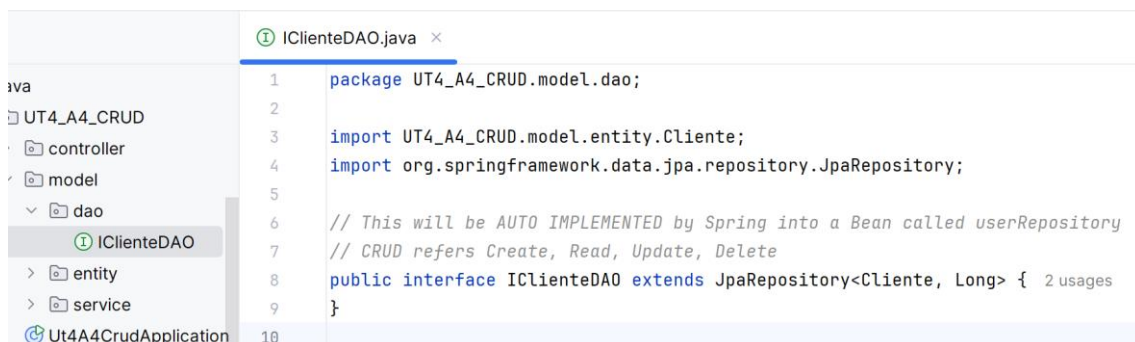


Añadimos los atributos con el nombre y propiedades que muestra el esquema usando etiquetas.

1. **Añade las etiquetas y atributos necesarios a la clase para que coincidan con los requeridos en el esquema de la tabla**

Creación del dao

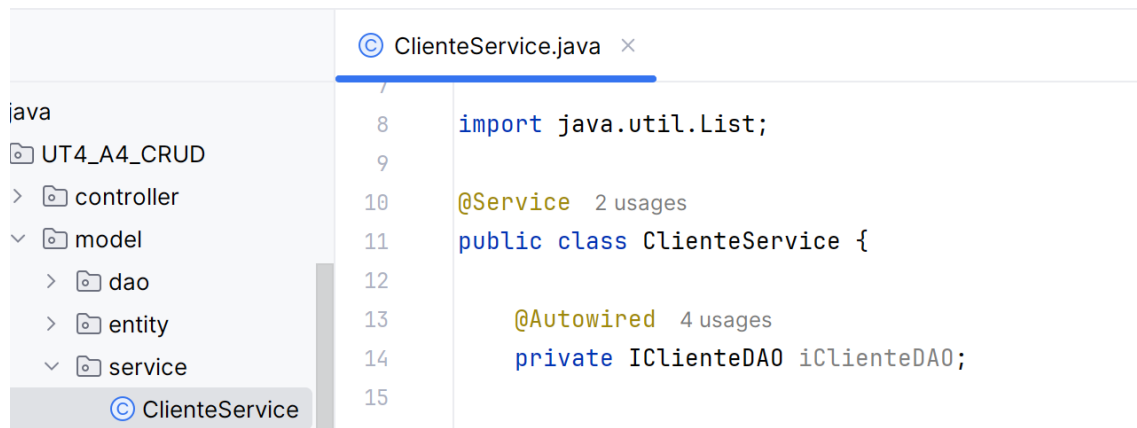
Creamos una interface llamada *IClienteDAO* que heredará de *JpaRepository*.



Creación de los servicios

Seguidamente crearemos la clase de servicios que contendrá la lógica de negocio. La llamaremos *ClienteService*. Indicaremos que es un servicio con la etiqueta *@Service*.

También añadiremos un atributo privado de la clase *IClienteDAO*. Tendremos que añadir la etiqueta *@Autowired* para que Spring haga su magia y podamos acceder a los métodos CRUD.



```

7
8 import java.util.List;
9
10 @Service 2 usages
11 public class ClienteService {
12
13     @Autowired 4 usages
14     private IClienteDAO iClienteDAO;
15
16

```

Seguidamente añadiremos los métodos para acceder a la base de datos. En nuestro caso tenemos

```

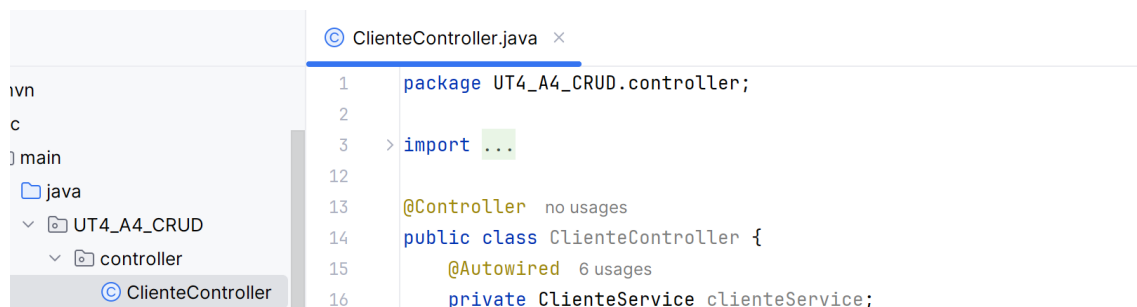
18 public Cliente getById(Long id) { 1 usage
19     // Lógica de negocio
20     if (id == null)
21         return null;
22
23     return iClienteDAO.getReferenceById(id);
24 }
25
26 public List<Cliente> findAll() { 1 usage
27     return iClienteDAO.findAll();
28 }
29
30 public Integer save(Cliente cliente) { 3 usages
31     iClienteDAO.save(cliente);
32     return 0;
33 }
34
35 public void deleteById(Long id) { 1 usage
36     iClienteDAO.deleteById(id);
37 }

```

El método `save` lo usaremos tanto para crear un nuevo registro como para editar uno ya existente.

Creación del controlador

Creamos un controlador llamado `ClienteController` indicando que es un controlador con la etiqueta `@Controller` y le añadimos un atributo de la clase de servicios `ClienteService`, con la etiqueta `@Autowired`



```

1 package UT4_A4_CRUD.controller;
2
3 import ...
4
12
13 @Controller no usages
14 public class ClienteController {
15     @Autowired 6 usages
16     private ClienteService clienteService;

```

Seguidamente tendremos que crear los métodos, tanto GET como POST, para poder listar los registros, añadir nuevos y editar o borrar los ya existentes.

En nuestra aplicación podremos acceder a los clientes a través de la URL `/clientes`. En ella crearemos un cliente nuevo, con los valores por defecto si es que no se han proporcionado los parámetros necesarios. En el caso de este controlador pasamos a la vista los atributos de clase a través de variables independientes para cada uno. Comprobamos aquellos valores que no admiten `null` en la tabla y les proporcionamos un valor por defecto si es necesario. Por último, se cargará la plantilla `clientes.html` que tendremos que crear en `templates`.

```

18     @GetMapping("/clientes") no usages
19     public String clientes(
20         @RequestParam(name = "nombre", required = false) String nombre,
21         @RequestParam(name = "apellido1", required = false) String apellido1,
22         @RequestParam(name = "apellido2", required = false) String apellido2,
23         @RequestParam(name = "ciudad", required = false) String ciudad,
24         @RequestParam(name = "categoria", required = false) Long categoria,
25         Model model) {
26
27         if (nombre == null)
28             nombre = "Anonimo";
29         if (apellido1 == null)
30             apellido1 = "";
31
32         Cliente cliente = new Cliente();
33         cliente.setNombre(nombre);
34         cliente.setApellido1(apellido1);
35         cliente.setApellido2(apellido2);
36         cliente.setCiudad(ciudad);
37         cliente.setCategoria(categoria);
38         clienteService.save(cliente);
39
40         model.addAttribute( attributeName: "nombre", nombre);
41         model.addAttribute( attributeName: "apellido1", apellido1);
42         model.addAttribute( attributeName: "apellido2", apellido2);
43         model.addAttribute( attributeName: "ciudad", ciudad);
44         model.addAttribute( attributeName: "categoria", categoria);
45         return "clientes";
46     }

```

Seguidamente creamos el método que nos listará todos los registros, accesible en `/clientes/all`. Añadirá al modelo que pasaremos a la vista un atributo llamado **clientes** que contendrá una lista con todos los clientes, que nos los dará `clienteService.findAll()`. Seguidamente se cargará `mostrar_clientes.html` que tendremos que crear en `templates`.

```

48     @GetMapping("/clientes/all") 3 usages
49     @ public String all(Model model) {
50         model.addAttribute( attributeName: "clientes", clienteService.findAll());
51         return "mostrar_clientes";
52     }

```

Seguidamente añadiremos los métodos que controlen la creación de nuevos registros, accesibles en `/clientes/add`. La introducción de los datos la haremos a través de un formulario, y necesitaremos métodos diferenciados para la petición GET, que mostrará el formulario, y para la POST, que realizaremos al remitir el formulario con los datos cumplimentados.

En el primer caso, crearemos un nuevo usuario en blanco que será el que nos muestre la vista en el formulario. Una vez editemos y remitamos el formulario, el método POST recibirá como parámetro el cliente con los datos introducidos en el formulario y los persistirá en la base de datos. No olvidarse por tanto de la etiqueta `@ModelAttribute` para que el método sepa que es un parámetro del modelo y no uno introducido en la URL.

```

54      @GetMapping("/clientes/add") no usages
55 @      public String add(Model model) {
56          model.addAttribute(attributeName: "cliente", new Cliente());
57          return "cliente_add";
58      }
59
60      @PostMapping("/clientes/add") no usages
61      public String add(@ModelAttribute Cliente cliente, Model model) {
62          clienteService.save(cliente);
63          return all(model);
64      }

```

El siguiente método servirá para borrar registros, accesible desde `/clientes/delete`. Necesitaremos que le suministremos el ID del registro a borrar.

```

66      @GetMapping("/clientes/delete") no usages
67      public String delete(@RequestParam(name = "id") Long id, Model model) {
68          clienteService.deleteById(id);
69          return all(model);
70      }

```

Finalmente tendremos que crear los métodos que controlen la edición de registros ya creados, que será accesible en `/clientes/edit`. El método POST, tras persistir los datos, deberá recargar el listado de registros.

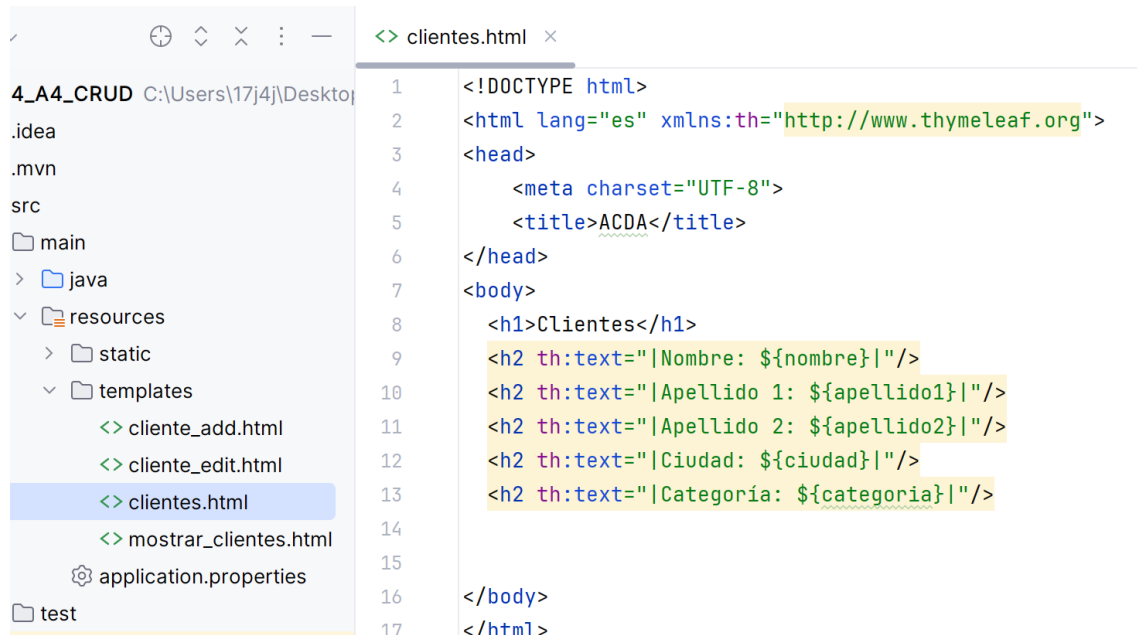
2. Crea los dos métodos del controlador necesarios para la edición de los registros de la tabla

Pista: serán similares a los de crear uno nuevo pero en este caso el formulario tendrá que cargar el registro cuyo ID le suministremos, no uno en blanco.

Creación de las vistas

Tenemos que crear las plantillas que luego *Thymeleaf* se encargará de renderizar y mandar al cliente. Estarán en `resources/templates`.

Cuando accedemos a `/clientes` el método del controlador encargado nos crea un nuevo cliente y carga `clientes.html`. En este archivo mostraremos los datos del nuevo cliente que se ha creado al acceder. Como hemos comentado arriba, se han pasado los atributos de la clase por separado en vez de como un objeto, por tanto accederemos directamente a ellos por el nombre que se les puso en el controlador.



```

1 <!DOCTYPE html>
2 <html lang="es" xmlns:th="http://www.thymeleaf.org">
3 <head>
4     <meta charset="UTF-8">
5     <title>ACDA</title>
6 </head>
7 <body>
8     <h1>Clientes</h1>
9     <h2 th:text="|Nombre: ${nombre}|" />
10    <h2 th:text="|Apellido 1: ${apellido1}|" />
11    <h2 th:text="|Apellido 2: ${apellido2}|" />
12    <h2 th:text="|Ciudad: ${ciudad}|" />
13    <h2 th:text="|Categoría: ${categoria}|" />
14
15
16 </body>
17 </html>

```

Para listar todos los registros debíamos acceder a `/clientes/all` y el método cargaba `mostrar_clientes.html`

Para esta vista usaremos una *stylesheet* que nos ponga la tabla más aparente.

```

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
rel="stylesheet" integrity="sha384-
QWTKZyjpPEjISv5WaRU90FeRpok6YctnYmDr5pNlyT2bRjXh0JMhY6hW+ALEwIH"
crossorigin="anonymous">

<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"
integrity="sha384-YvpcrYf0tY3lHB60NNkmXc5s9fDVZLESaAA55NDzOxhy9GkcIdslK1eN7N6jIeHz"
crossorigin="anonymous"></script>

```

Aquí puedes copiar y pegar las dos etiquetas para incluirlas en tu código, ya que en las imágenes no se aprecian enteros.

Mostraremos los registros en una tabla. Primero pondremos el encabezado con los nombres de las columnas. Como se puede apreciar hay una columna en la tabla para cada atributo, y además al final hay dos columnas más en las que podremos borrar o editar cada registro a través de un enlace.

<> mostrar_clientes.html x

```

1  <!DOCTYPE html>
2  <html lang="es" xmlns:th="http://www.thymeleaf.org">
3  <head>
4      <meta charset="UTF-8">
5      <title>ACDA</title>
6      <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet">
7  </head>
8  <body>
9  <h1>Todos los clientes</h1>
10 <p th:text="|Clientes: ${clientes}|" />
11
12 <table class="table">
13     <thead>
14         <tr>
15             <th scope="col">ID</th>
16             <th scope="col">Nombre</th>
17             <th scope="col">Apellido 1</th>
18             <th scope="col">Apellido 2</th>
19             <th scope="col">Ciudad</th>
20             <th scope="col">Categoría</th>
21             <th>Eliminar</th>
22             <th>Editar</th>
23         </tr>
24     </thead>

```

Seguidamente añadiremos una fila por cada uno de los registros de la tabla. El controlador ha añadido al modelo un atributo llamado clientes que contiene una lista con todos los registros de la tabla. Recorreremos la lista extrayendo cada cliente de uno en uno (a la variable cliente) y generando una nueva fila en la que pondremos los datos, a los que accedemos mediante cliente.nombre_atributo.

Además, en las dos últimas columnas generaremos un enlace a las rutas que permiten borrar y editar registros, pasándoles como parámetro el ID del registro que muestra esa fila.

```

25 <tbody>
26 <tr th:each="cliente: ${clientes}">
27     <th scope="row" th:text="${cliente.id}" />
28     <td th:text="${cliente.nombre}" />
29     <td th:text="${cliente.apellido1}" />
30     <td th:text="${cliente.apellido2}" />
31     <td th:text="${cliente.ciudad}" />
32     <td th:text="${cliente.categoria}" />
33     <td><a th:href="|/clientes/delete?id=${cliente.id}|">Eliminar</a></td>
34     <td><a th:href="|/clientes/edit?id=${cliente.id}|">Editar</a></td>
35 </tr>
36 </tbody>
37 </table>
38
39 <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js" rel="stylesheet">
40 </body>
41 </html>

```

El siguiente archivo es el que nos mostrará el formulario para crear un nuevo registro. Como se aprecia, es similar al que usamos en la actividad guiada para remitir datos con formularios. El método POST, tras persistir los datos, recarga el listado de registros, así que no tiene una plantilla asignada.

<> cliente_add.html x

```

1  <!DOCTYPE html>
2  <html lang="es" xmlns:th="http://www.thymeleaf.org">
3  <head>
4      <meta charset="UTF-8">
5      <title>ACDA</title>
6  </head>
7  <body>
8      <h1>Formulario para añadir cliente</h1>
9
10     <form action="#" th:action="@{/clientes/add}" th:object="${cliente}" method="post">
11         <p>Nombre: <input type="text" th:field="*{nombre}" /></p>
12         <p>Apellido 1: <input type="text" th:field="*{apellido1}" /></p>
13         <p>Apellido 2: <input type="text" th:field="*{apellido2}" /></p>
14         <p>Ciudad: <input type="text" th:field="*{ciudad}" /></p>
15         <p>Categoría: <input type="number" th:field="*{categoria}" /></p>
16         <p><input type="submit" value="Submit" /> <input type="reset" value="Reset" /></p>
17     </form>
18 </body>
19 </html>

```

El borrado de datos tampoco tenía una plantilla asignada. El método borra el registro de la base de datos y recarga el listado de registros.

Nos queda la plantilla de edición de registros. El controlador de las peticiones POST se ha dicho que recargará el listado de registros, así que no tendrá plantilla asignada.

3. Crea la plantilla que muestre el formulario de edición de un registro.

Pista: El ID también debe remitirse ya que creará un nuevo registro en vez de editarlo si no se lo proporcionamos. Por lo tanto, debe estar en el formulario, sin embargo, no debe ser editable. Explora las posibilidades que tienes para ello.

Conexión con la base de datos y puerto de escucha

De manera similar a la actividad guiada que ya hicimos. Tendrás que indicar las propiedades de la conexión en el archivo de propiedades de la aplicación. Si no quieres tener que poner en la URL el puerto de la conexión, puedes establecerlo en este archivo para que escuche en el puerto 80 que es el de por defecto para aplicaciones web. Sin embargo, asegúrate de no tener en marcha ningún otro servidor en el mismo puerto.

Clase de Aplicación

La clase para ejecutar la aplicación la ha creado automáticamente Spring.

Ejecución de la aplicación

Como ya hemos visto, al ejecutar la aplicación nos va a poner en marcha un servidor en el puerto 8080. Si lo hemos establecido en el puerto 80 no necesitaremos indicarlo en la URL.

← → 🛡️ 📄 localhost/clientes

Al acceder a `/clientes` nos creará un cliente nuevo con datos por defecto.

Cientes

Nombre: Anonimo

Apellido 1:

Apellido 2: null

Ciudad: null

Categoría: null

Accediendo a `/clientes/all` veremos el listado de clientes.

Todos los clientes

Clientes: [UT4_A4_CRUD.model.entity.Cliente@126912ae]

ID	Nombre	Apellido 1	Apellido 2	Ciudad	Categoría	Eliminar	Editar
1	Anonimo					Eliminar	Editar

Al haberle aplicado una hoja de estilos, tenemos una vista diferente a la de por defecto. Se aprecia a la derecha del registro cómo hay dos enlaces que nos permiten borrar o editar (si lo hemos hecho bien) el registro.

Vamos a añadir un nuevo registro. Para ello, accedemos a `/clientes/add`

Formulario para añadir cliente

Nombre:

Apellido 1:

Apellido 2:

Ciudad:

Categoría:

Formulario para añadir cliente

Nombre:

Apellido 1:

Apellido 2:

Ciudad:

Categoría:

Tras hacer el submit nos carga el listado.

Todos los clientes

Clientes: [UT4_A4_CRUD.model.entity.Cliente@417645ac, UT4_A4_CRUD.model.entity.Cliente@7dbde3f2]

ID	Nombre	Apellido 1	Apellido 2	Ciudad	Categoría	Eliminar	Editar
1	Anonimo					Eliminar	Editar
2	Eneko	eres	mal cliente	Iruñea	1	Eliminar	Editar

Llegó el momento de editar el primer registro a través del enlace.

Formulario para editar un cliente

Id: 1

Nombre:

Apellido 1:

Apellido 2:

Ciudad:

Categoría:

Formulario para editar un cliente

Id: 1

Nombre:

Apellido 1:

Apellido 2:

Ciudad:

Categoría:

Todos los clientes

Clientes: [UT4_A4_CRUD.model.entity.Cliente@1ba7e0b6, UT4_A4_CRUD.model.entity.Cliente@304f148b]

ID	Nombre	Apellido 1	Apellido 2	Ciudad	Categoría	Eliminar	Editar
1	Emilio	es	mucho mejor	Granada	10	Eliminar	Editar
2	Eneko	eres	mal cliente	Iruñea	1	Eliminar	Editar

Por último, borramos registros directamente desde el enlace para eliminar.

Todos los clientes

Clientes: [UT4_A4_CRUD.model.entity.Cliente@101dfbe9]

ID	Nombre	Apellido 1	Apellido 2	Ciudad	Categoría	Eliminar	Editar
1	Emilio	es	mucho mejor	Granada	10	Eliminar	Editar

Mini reto: modifica el listado de registros para que también tenga un enlace para añadir nuevos registros.