



Unión Europea

Fondo Social Europeo

El FSE invierte en tu futuro

Fecha	Versión	Descripción
17/10/2022	1.0.0	Versión inicial.
07/11/2022	1.0.1	Revisión y corrección de errores.
12/11/2022	1.0.2	Corrección de errores. Aclaración de tipos de expresiones. Cambio de orden a la hora de crear los métodos del controlador. Anexos. TextArea.
15/11/2022	1.0.3	Error al crear un nuevo cliente con la lista vacía. Solucionado al asignar id=1
09/10/2024	2.0.0	Actualización de los apuntes

Unidad 4 - Estructura de una aplicación empresarial.

Unidad 4 - Estructura de una aplicación empresarial.

Estructura de aplicaciones Spring

Estructura de capas en una aplicación Spring.

Clase Principal

Capa Web (Web Layer). Capa Controlador

Capa de Servicios

Capa repositorio

Capa modelo: entidades y DTO's

La estructura del proyecto en Spring.

Patrones de diseño.

Patrón MVC.

Patrón DTO.

Patrón DAO.

Patrón FACADE.

Biblioteca Lombok

DISEÑANDO LA APLICACIÓN SPRING APLICANDO LOS PATRONES VISTOS.

Definir e implementar el modelo

Definir e implementar el DTO

Definir e implementar el controlador

Mostrar página inicial de la aplicación

Operación "Listar todos los clientes"

Operación "Obtener Cliente por id"

Diferencia entre @PathVariable y @RequestParam

- Operación Add
- Operación Update
- Operación Delete
- Anotación @Autowired
- Definir e implementar el Servicio
- Definir e implementar el Repositorio
- Definir e implementar las Vistas

ANEXO I: ACLARACIONES ACERCA DE NUESTRA ARQUITECTURA

ANEXO II: TABLA DE CODIFICACIONES ENTRE ANSI Y UTF-8

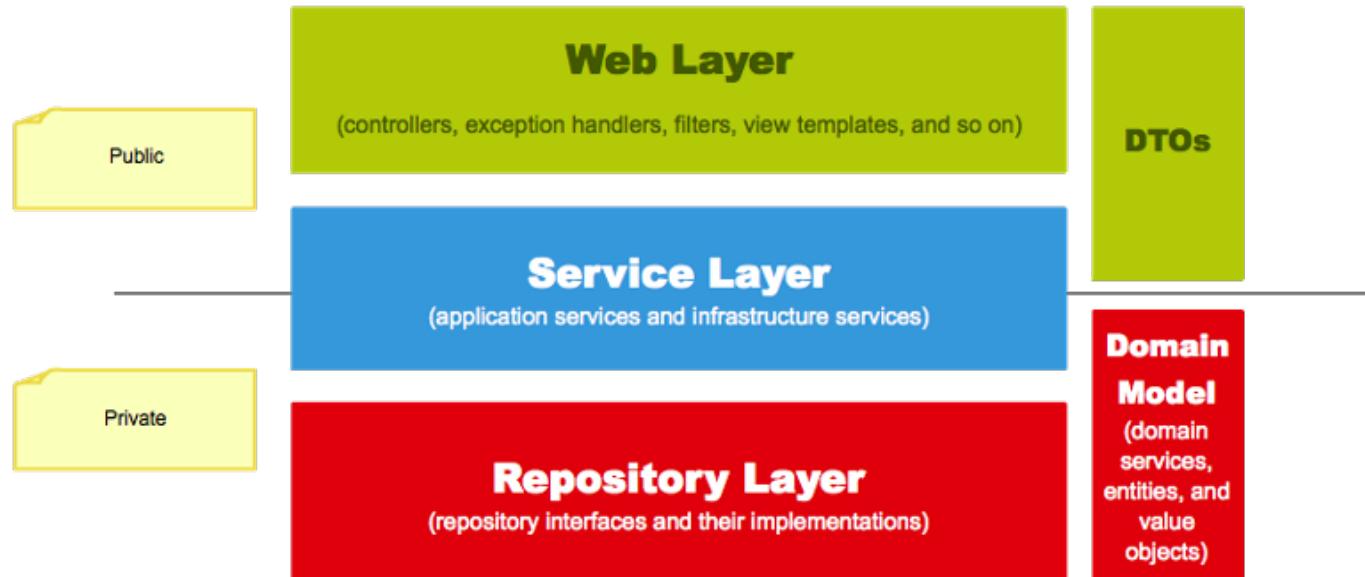
Estructura de aplicaciones Spring

Vamos a comenzar conociendo algunos patrones de diseño que vamos a usar en nuestra aplicación, y que son necesarios para definir una correcta arquitectura en una aplicación Spring. Por ello, lo primero que haremos será definir las capas de la estructura de nuestra aplicación, y una vez que la definamos definiremos nuevos patrones de diseño que integraremos en una aplicación de ejemplo antes de conectarnos a la base de datos y comenzar a utilizar Hibernate.

Estructura de capas en una aplicación Spring.

La idea es crear una estructura de paquetes que agrupe las clases en 4 paquetes principales: paquete que contiene la clase principal que hace funcionar la aplicación, capa web que contiene los controladores, capa de acceso a datos que contiene el repositorio, capa de servicio que contiene la lógica de negocio y, dentro de ella, paquete de entidades y paquete de dto. Todas estas se engloban, básicamente en 3: web, servicio y repositorio.

El objetivo es que con una arquitectura bien definida y acoplada es posible usar la Inyección de Dependencias, ya que nos facilitará la comunicación y el acceso entre las diferentes capas.



Clase Principal

Toda aplicación en java debe contener una clase principal con un método `main`. Dicho método, en caso de implementar una aplicación con Spring, deberá llamar al método `run` de la clase `SpringApplication`.

Esta clase la dejaremos en la raíz por defecto, de forma que siempre la tendremos en el mismo sitio.

Capa Web (Web Layer). Capa Controlador

Definamos ahora el comportamiento de la aplicación implementando el resto de clases. Vamos a comenzar por la capa de más alto nivel, la de los controladores, donde expondremos los servicios de la aplicación. El paquete principal se llamará `web`.

Esta capa trabajara básicamente con lo siguiente:

- Servicios Web consumidos por aplicaciones (servicio REST o SOAP).
- Controladores y/o vistas implementadas con JSP, JSF, Thymeleaf, etc. Tener en cuenta que algunos frameworks de desarrollo de la capa de vista ubican los ficheros de la vista en `resources`.
- Vistas con frameworks tipo Vaadin, Wicket, ZK, etc.

Dentro del paquete `web` crearemos un nuevo paquete denominado `controller` donde ubicaremos los controladores que trabajan directamente con la capa de vista (no los controladores que trabajan con servicios web). En caso de ser controladores de servicios web se encontrarán en el paquete `webservices`.

Capa de Servicios

Capa que se encarga de implementar la lógica empresarial, o sea, todas las tareas que es capaz de realizar nuestro sistema.

Esta es una de las capas más importantes ya que aquí se realizarán todas las operaciones de validación de datos que hacen referencia a la lógica del negocio (por ejemplo, controlar que una cuenta corriente dispone de saldo a la hora de realizar un pago) y seguridad. Se denominará `service`.

Estos servicios son clases que se encargan de la capa de negocio de la aplicación. Normalmente, acceden a los datos almacenados en la base de datos de la aplicación a través de los repositorios, hacen una serie de operaciones, y envían los datos al controlador. Podemos encontrarnos con los siguientes tipos de servicio:

- Servicios de Integridad del Repositorio: se encargan de consumir información del repositorio. Son servicios fáciles de implementar (por ejemplo, pedir una lista de clientes).
- Servicios de Operabilidad del negocio: realizan operaciones específicas para el flujo de negocio (realizan operaciones complejas para completar una transacción, como puede ser una venta, almacenar un pedido, etc).
- Servicios de Seguridad: dedicados a realizar operaciones de seguridad
- Servicios de Gestión: dedicados a generar informes y/o estadísticas.

Capa repositorio

Los repositorios son las clases encargadas de gestionar el acceso a los datos. Por lo general, contiene clases que realizan operaciones CRUD utilizando solo una clase de entidad de un modelo de dominio. Se denominará el paquete `repository`.

Capa modelo: entidades y DTO's

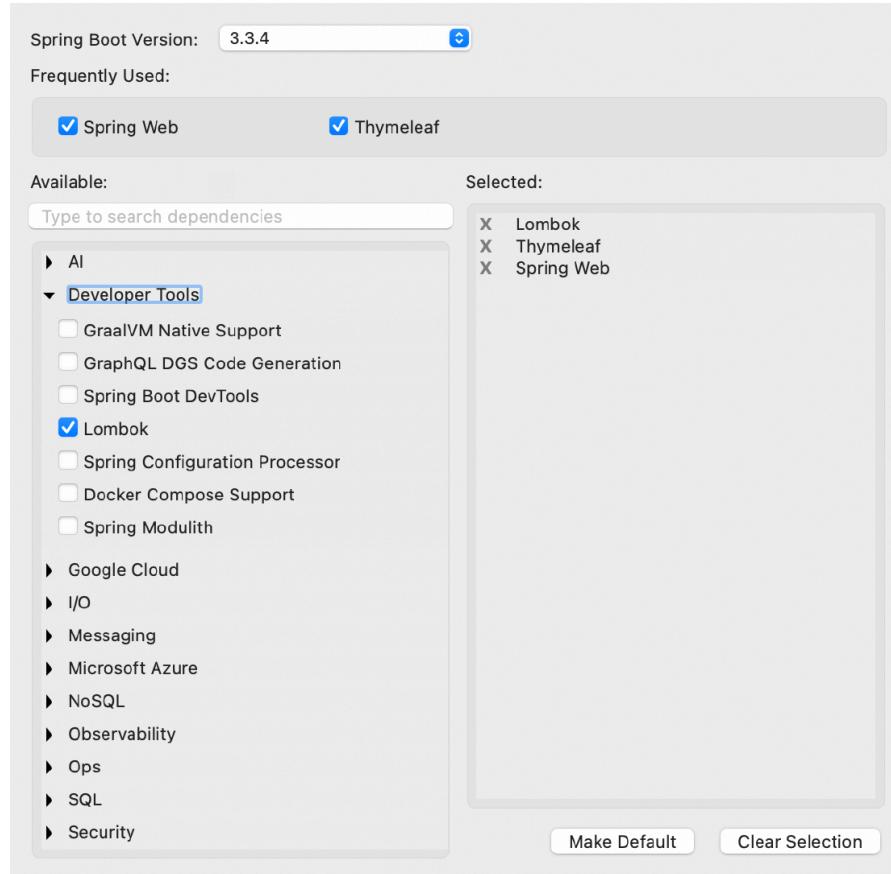
Contendrá los mapeos de las tablas de base de datos en clases que representan entidades (anteriormente serán nuestros POJOS, pero ahora se convierten en mapeos de entidades), así como los dto's.

Como los POJO's ahora son mapeos de entidades se encontrarán dentro del paquete `repository` en un paquete denominado `entity`.

Los controladores manipulan DTO (Data Transfer Object o Objeto de Transferencia de Datos) en lugar de POJO's, debido a la estructura de la API o la representación en las vistas. Por lo tanto, tendremos que implementar una conversión bidireccional entre un POJO y un modelo DTO. Más adelante definiremos, en los patrones de diseño, que es un DTO y veremos la conversión bidireccional, denominada mapeo. Además, estos DTO's estarán en un paquete aparte denominado `dto` y se encontrará dentro del paquete `model`.

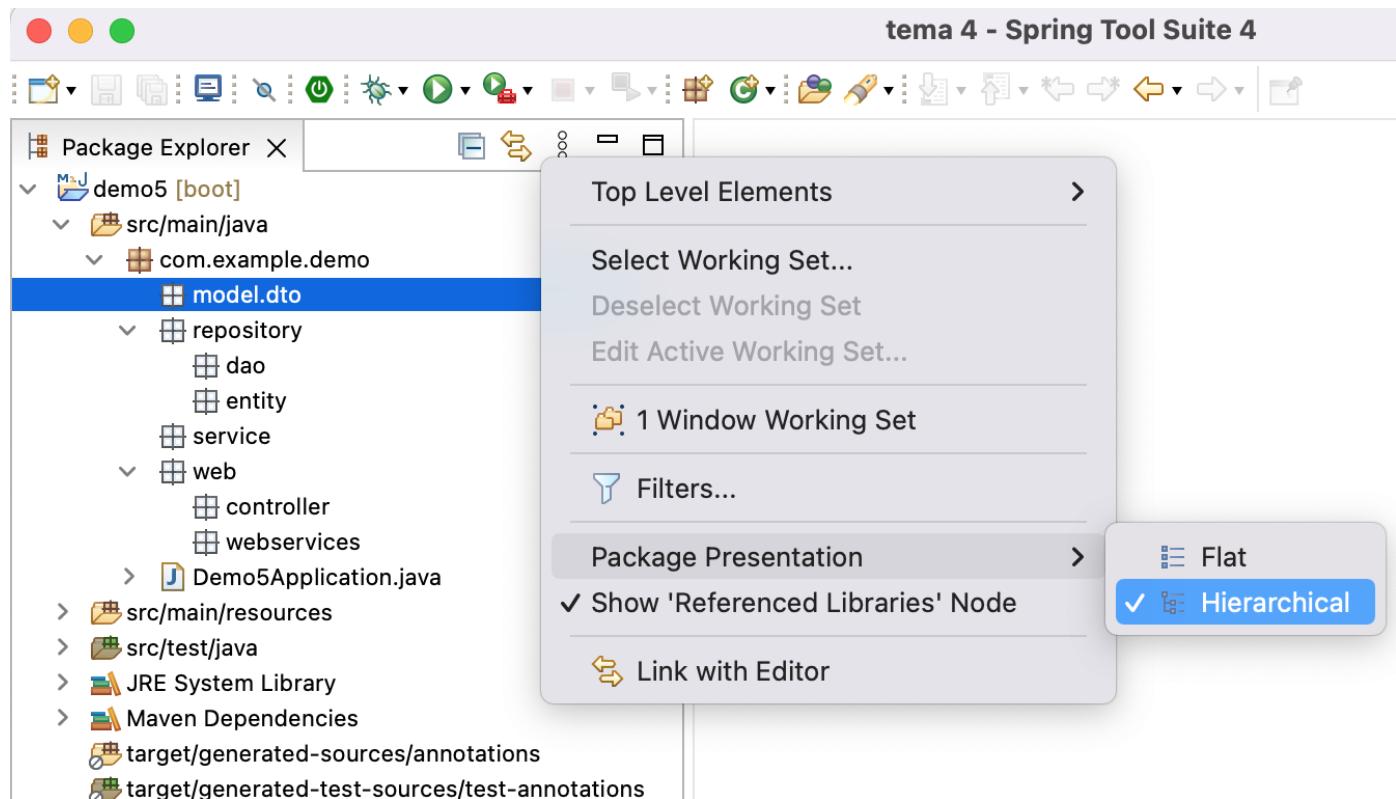
La estructura del proyecto en Spring.

Creamos un proyecto en Spring Boot denominado demo5, de forma que añadiremos las siguientes dependencias al paquete:



Fijaros que hemos añadido también la dependencia `Lombok`, que es una librería para Java que a través de anotaciones nos reduce el código que codificamos, es decir, nos ahorra tiempo y mejora la legibilidad del mismo. Las transformaciones de código que realiza se hacen en tiempo de compilación.

Ahora procedemos a crear la estructura de paquetes, tal y como hemos definido con anterioridad. La estructura del proyecto quedará de la siguiente forma (cambiamos la visualización de los paquetes para poder ver la estructura jerárquica que se crea):



Patrones de diseño.

Un patrón de diseño es una solución probada que resuelve un tipo específico de problema en el desarrollo de software referente al diseño.

Existen una infinidad de patrones de diseño los mismos que se dividen en categorías, por ejemplo: de creación, estructurales, de comportamiento, interacción etc.

¿Por qué utilizar patrones de diseño?: permiten tener el código bien organizado, legible y mantenable, además te permite reutilizar código y aumenta la escalabilidad en tu proyecto.

En sí proporcionan una terminología estándar y un conjunto de buenas prácticas en cuanto a la solución en problemas de desarrollo de software.

Vamos a explicar varios de ellos para comenzar a entender que son los patrones de diseño.

Patrón MVC.

Este patrón ya lo hemos explicado en el tema anterior, pero vamos a darle un pequeño repaso.

Permite separar una aplicación en 3 capas, una forma de organizar y de hacer escalable un proyecto. Las capas que podemos encontrar son:

- **Modelo:** Esta capa representa todo lo que tiene que ver con el acceso a datos: guardar, actualizar, obtener datos, además todo el código de la lógica del negocio, básicamente son las clases Java y parte de la lógica de negocio. En esta capa se encuentran los paquetes `model`, `repository` y `service`.
- **Vista:** La vista tiene que ver con la presentación de datos del modelo y lo que ve el usuario, por lo general una vista es la representación visual de un modelo (POJO o clase java). Por ejemplo, el modelo usuario que es una clase en Java y que tiene como propiedades, nombre y apellido debe pertenecer a una vista en la que el usuario vea esas propiedades. En esta capa se encuentran todos los ficheros `html` que vayamos creando con Thymeleaf que se encuentran en `resources`.
- **Controlador:** El controlador es el encargado de conectar el modelo con las vistas, funciona como un puente entre la vista y el modelo, el controlador recibe eventos generados por el usuario desde las vistas y se encargar de dirigir al modelo la petición respectiva. Por ejemplo, el usuario quiere ver los clientes con apellido Álvarez, la petición va al controlador y el se encarga de utilizar el modelo adecuado y devolver ese modelo a la vista. Esta capa, si son controladores sin servicios web se encontrarán en el paquete `controller`, mientras que si son controladores con servicios web se encontrarán en el paquete `webservice`, ambos dentro del paquete `web`.

En ningún momento interactúan directamente la vista con el modelo, esto también mantiene la seguridad en una aplicación.

Lo importante de este patrón es que permite dividir en partes, que de alguna manera son independientes, con lo que si por ejemplo se hace algún cambio en el modelo no afectaría a la vista o si hay algún cambio sería mínimo.

Patrón DTO.

Con este patrón se diseña una de las capas transversales de la arquitectura. Resuelve el problema de cómo permitir a un cliente intercambiar datos con el servidor sin hacer múltiples llamadas preguntando por cada dato. Por ejemplo, si tenemos una entidad denominada Persona y una entidad denominada Direcciones, al preguntar por las personas y sus direcciones tenemos que realizar múltiples llamadas hacia el servidor para preguntar las personas y las direcciones de cada persona, construyendo con esta información la vista.

DTO lo resuelve transfiriendo un objeto ligero al cliente con todos los datos necesarios, de forma conjunta. Después, el cliente puede hacer peticiones locales al objeto que ha recibido.

Para ello, se crean clases Java que encapsulan los datos en un paquete transportable por la red (pueden implementar `java.io.Serializable`, aunque no es obligatorio), es decir, con el ejemplo anterior, crearíamos una clase Java que llevaría la persona y sus direcciones, todo junto en el mismo objeto.

Estos objetos se usan en todas las capas de la aplicación, de forma que la información es transportada por todas las capas de la aplicación.

Es recomendable llenar siempre todos los campos del DTO para evitar errores `NullPointerException` (puede ser mejor una cadena vacía), hacer que los DTOs sean autodescriptivos, usar arrays o colecciones de DTOs cuando sea necesario, y considerar métodos que redefinan `equals()`.

Hay dos variantes de DTO's:

- DTOs personalizados que representan parte de un POJO o agrupan varios POJO's. Nosotros trabajaremos con ellos en el paquete `dto`.
- DTOs de dominio denominados `entities`. Una clase de dominio no es accesible directamente por el cliente, ya que, por la separación del patrón MVC desde la vista no se puede acceder a las entidades que mapean la BD (que son las `entities`). Por esa razón, se hacen copias DTO de los objetos de dominio a los DTOs personalizados, así los clientes pueden operar sobre copias locales mejorando el rendimiento de las lecturas y actualizaciones. Estas `entities` son las que se encuentran en el paquete `entity`.

Con todo esto, podemos resumir que DTO es un patrón muy efectivo para transmitir información entre un cliente y un servidor, pues permite crear estructuras de datos independientes de nuestro modelo de datos (entidades), lo que nos permite crear cuantas "vistas" sean necesarias de un conjunto de tablas u orígenes de datos. Además, nos permite controlar el formato, nombre y tipos de datos con los que transmitimos los datos para ajustarnos a un determinado requerimiento. Finalmente, si por alguna razón, el modelo de datos cambia (y con ello las entidades) el cliente no se afectará, pues seguirá recibiendo el mismo DTO.

En el ejemplo que vamos a diseñar a continuación veremos como implementar el patrón DTO.

Patrón DAO.

El patrón Data Access Object (DAO), el cual permite separar la lógica de acceso a datos de los Objetos de negocios (Business Objects), de tal forma que el DAO encapsula toda la lógica de acceso de datos al resto de la aplicación.

Esta propuesta propone separar por completo la lógica de negocio de la lógica para acceder a los datos, de esta forma, el DAO proporcionará los métodos necesarios para insertar, actualizar, borrar y consultar la información; por otra parte, la capa de negocio solo se preocupa por lógica de negocio y utiliza el DAO para interactuar con la fuente de datos.

En el ejemplo que vamos a diseñar a continuación veremos como implementar el patrón DAO pero de una forma distinta a la habitual, puesto que en este ejemplo todavía no vamos a acceder a la base de datos.

Patrón FACADE.

El patrón de diseño `Facade` simplifica la complejidad de un sistema mediante una interfaz más sencilla. Mejora el acceso a nuestro sistema logrando que otros sistemas o subsistemas usen un punto de acceso en común que reduce la complejidad, minimizando las interacciones y dependencias. Es decir, crearemos una interfaz Java que tendrá las cabeceras de los métodos como punto de acceso común, mientras que habrá clases Java que implementarán dicha interfaz.

A lo largo de este ejemplo haremos uso de dicho patrón.

Biblioteca LomBok

Project Lombok es una biblioteca para Java que nos ofrece bastantes funcionalidades. Para ello tiene que estar conectado a nuestro compilador, ya que su objetivo es facilitarnos el desarrollo de nuestro código evitándonos tener que escribir ciertos métodos, que van a ser repetitivos y que realmente tampoco aportan lógica al negocio.

Es necesario que Lombok esté integrado con el propio IDE, con nuestro entorno de desarrollo y compilación, porque va a generar una serie de métodos, y con ese plugin podremos acceder desde una clase al método generado por Lombok.

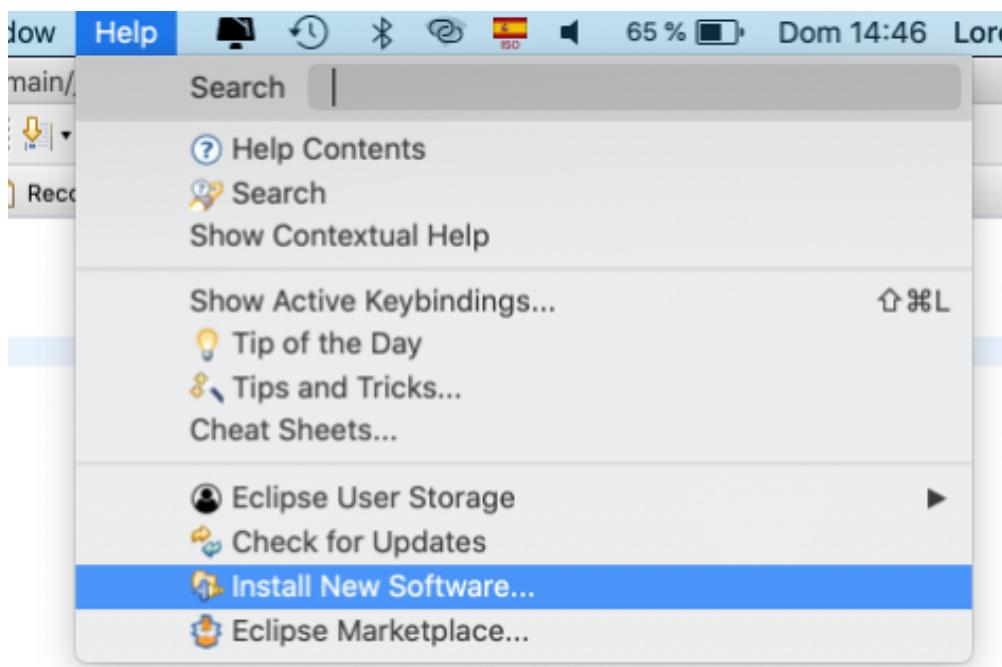
El código que puede generar Lombok y así nos evita a nosotros desarrollar, son métodos como getters/setters, equals, constructores y nos facilita la creación de variables y constantes.

Lombok ofrece una serie de ventajas, de las que podemos destacar:

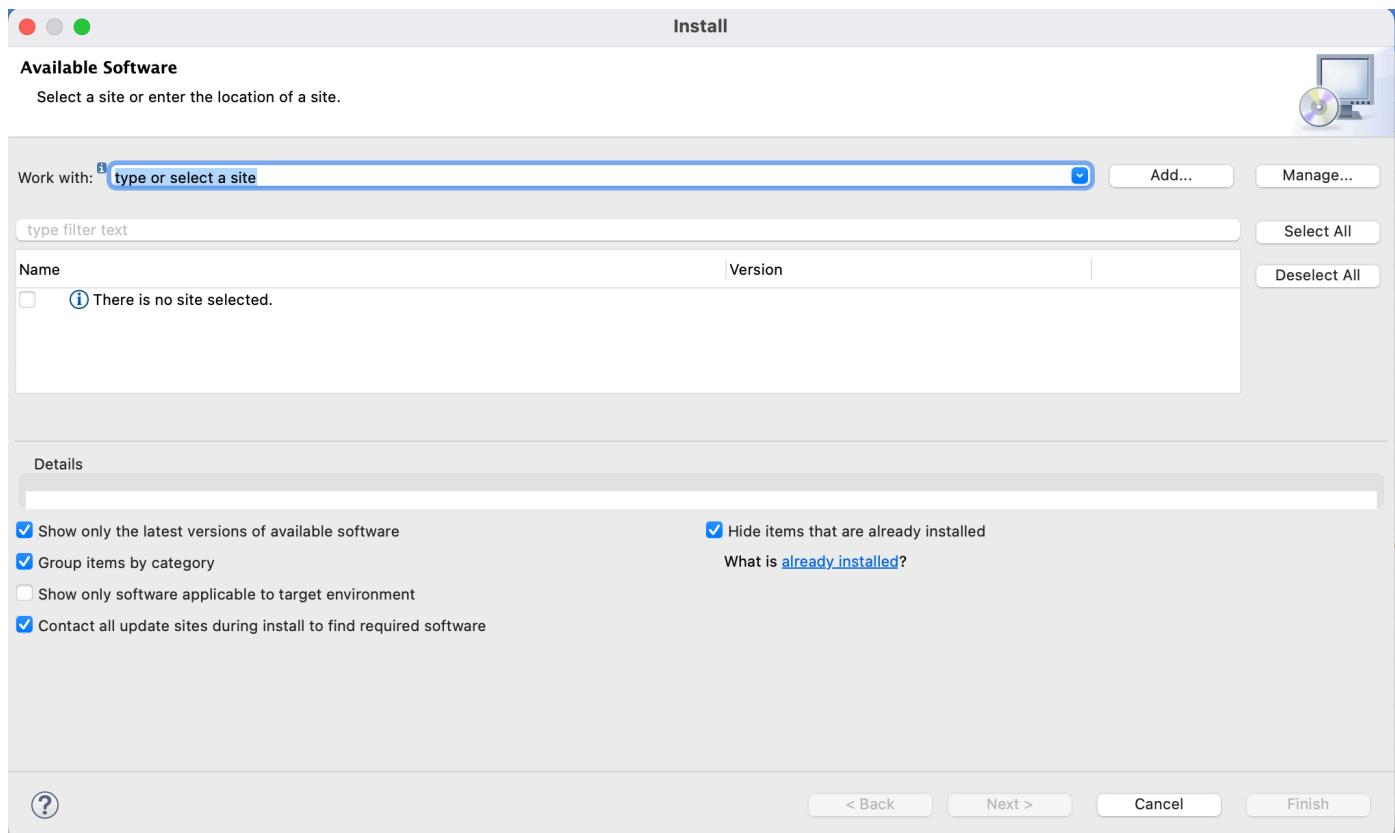
- Elimina código repetitivo y nos facilita generar código. Por lo tanto, no tenemos que desarrollar nosotros ese código, y si lo hacemos, simplemente eliminándolo, nos quedará un código mucho más limpio, lo que nos permitirá focalizarnos en el negocio, que al final es lo que importa.
- Inyecta código sobre el código que hemos desarrollado. Nos permite tener getters, setters, constructores, un constructor para un parámetro, otro para los parámetros obligatorios, etc. Lombok se encarga de todo esto tras injectar su código sobre el nuestro, además es totalmente instantáneo para el desarrollador.

A la hora de trabajar con STS Lombok tiene un pequeño bug que va a hacer que no nos vaya mostrando lo que va a ir generando en tiempo de edición. Por ello vamos a instalar Lombok en nuestro STS previamente, de forma que funcionará de manera correcta cuando trabajemos con la vista *outline*:

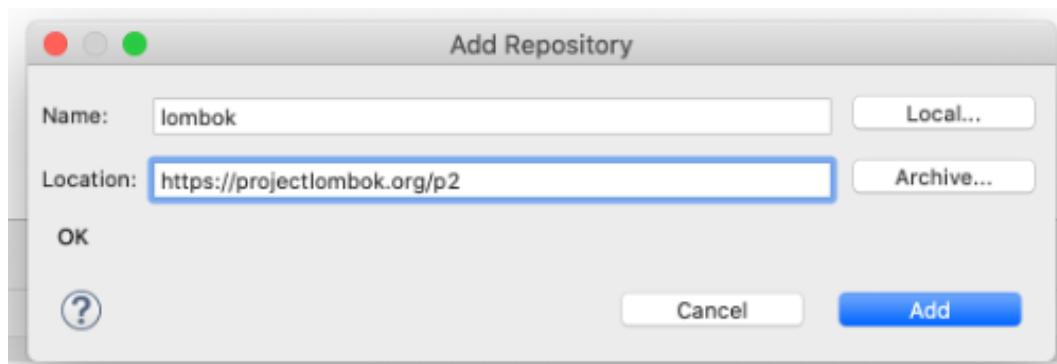
1. Iremos a instalar un nuevo software en el STS desde la opción *Help > Install New Software...*



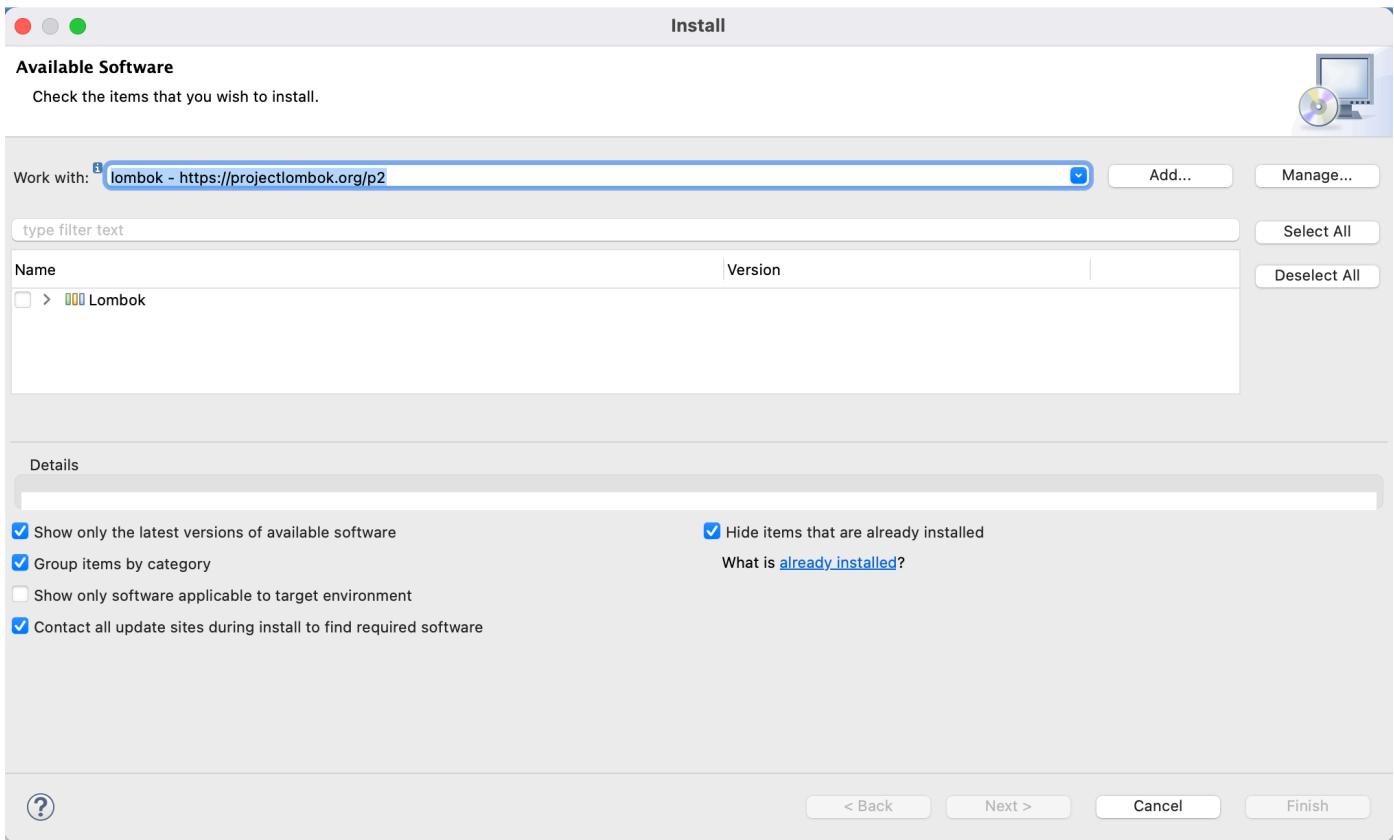
2. Nos aparecerá la siguiente pantalla para instalar nuevo software..



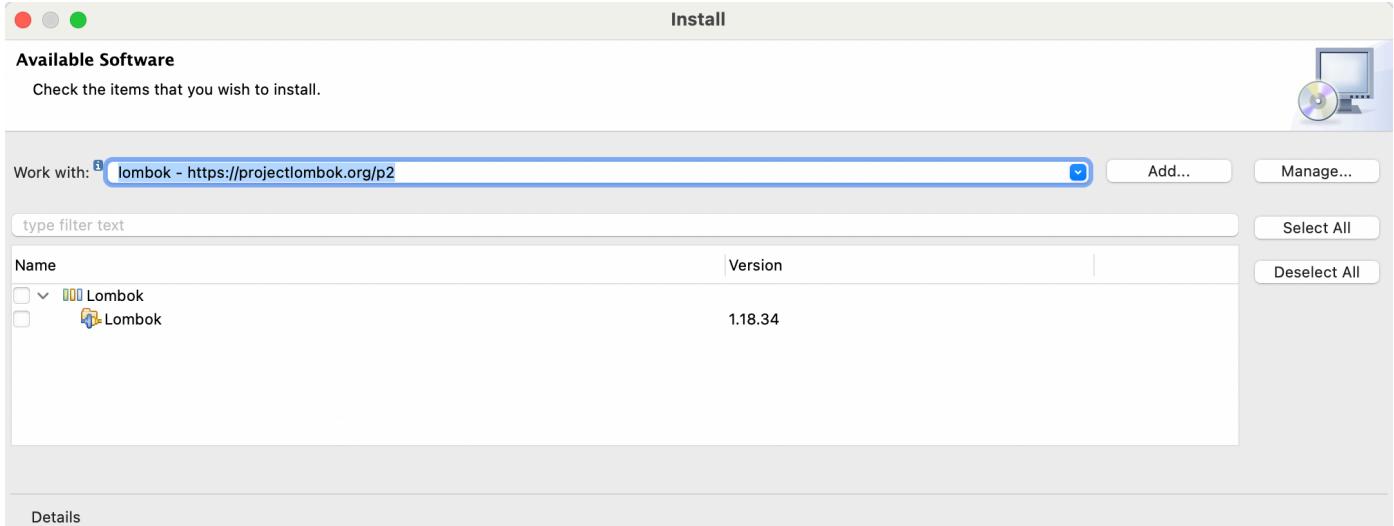
3. Pulsamos el botón *Add..* para añadir un nuevo repositorio de software y ponemos los siguientes datos:



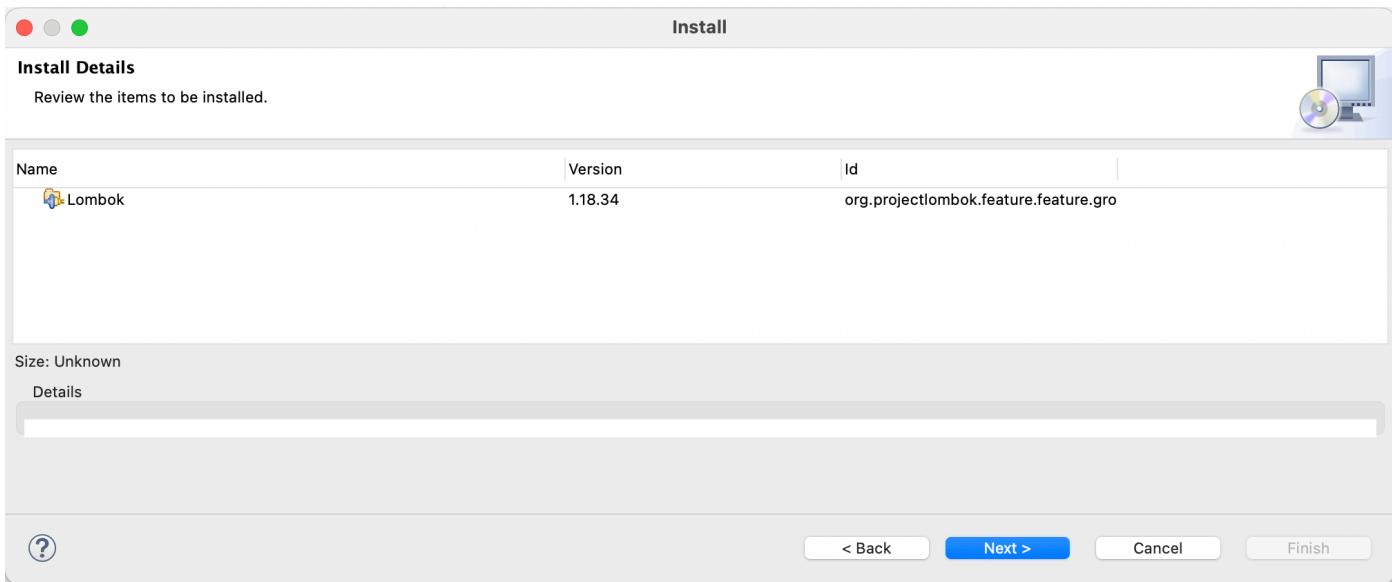
4. Al darle a *Add* obtendremos el software que nos proporciona el origen de datos:



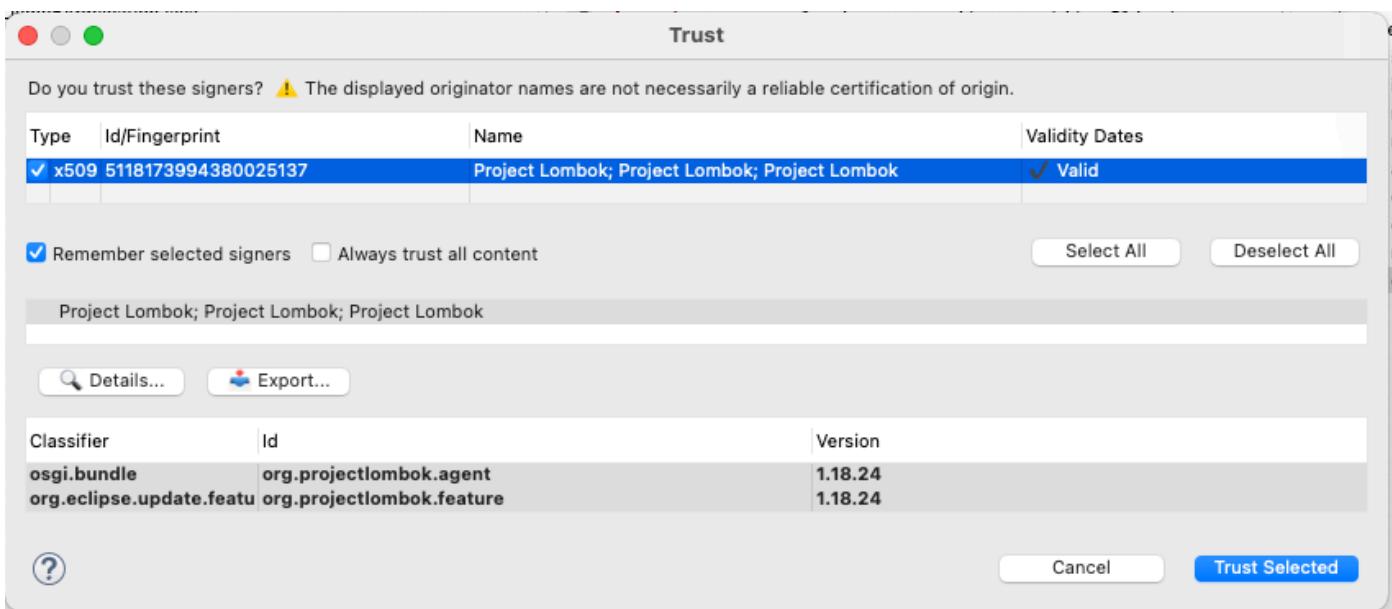
5. Seleccionamos *Lombok* y pulsamos el botón *Next*:



6. Nos lleva a una nueva pantalla que nos muestra detalles de la instalación. Pulsamos el botón *Next*:



7. A continuación nos sale que aceptemos la licencia. Aceptamos y pulsamos *Finish*.
8. A continuación nos sale un mensaje indicando que hay que aceptar el certificado con el que vamos a trabajar. Seleccionamos todo pulsando *Select All* y a continuación pulsamos *Trust Selected*.

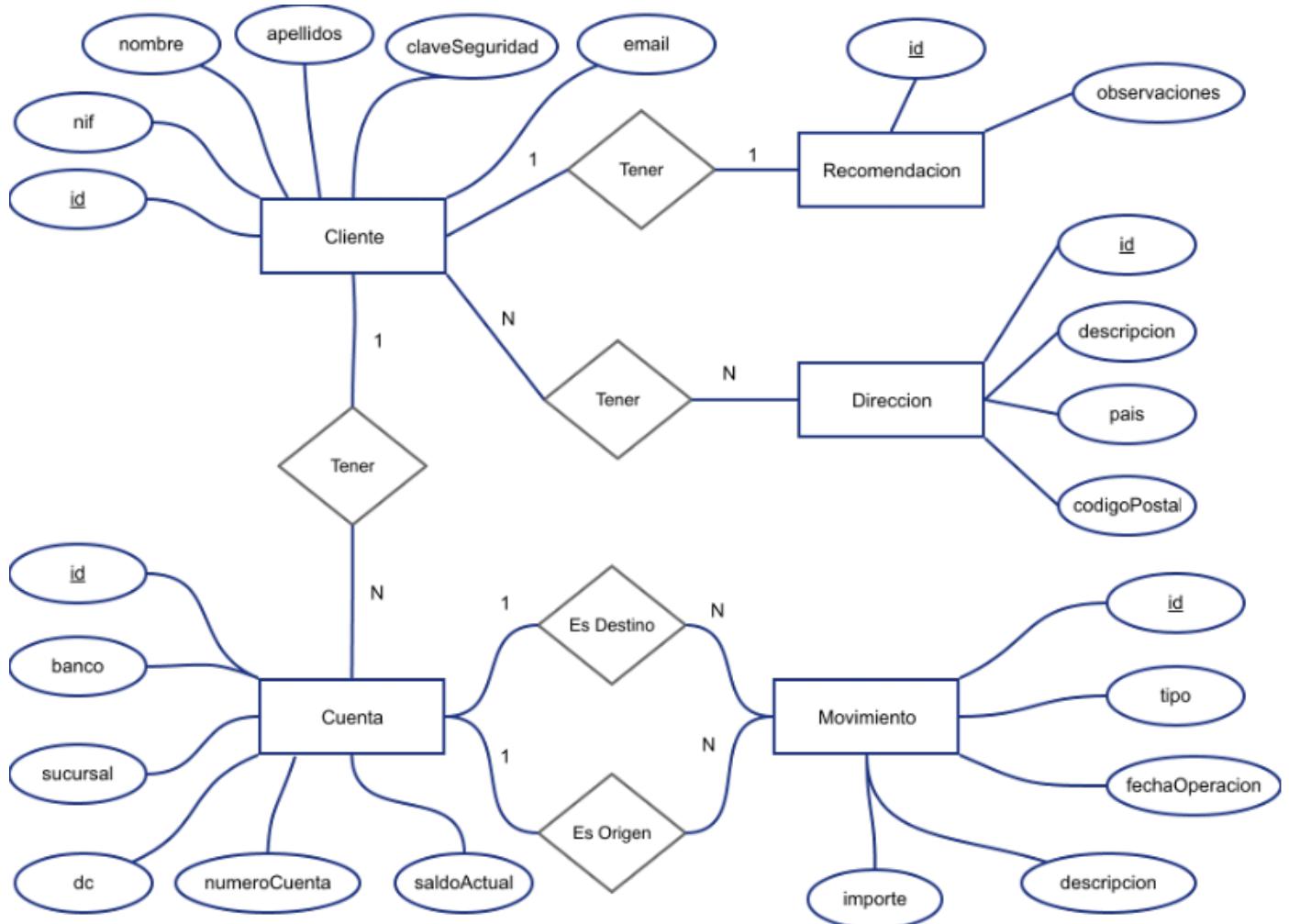


8. Tras ello el software queda instalado. Cerramos el STS y lo volvemos a iniciar para que el plugin se cargue correctamente.

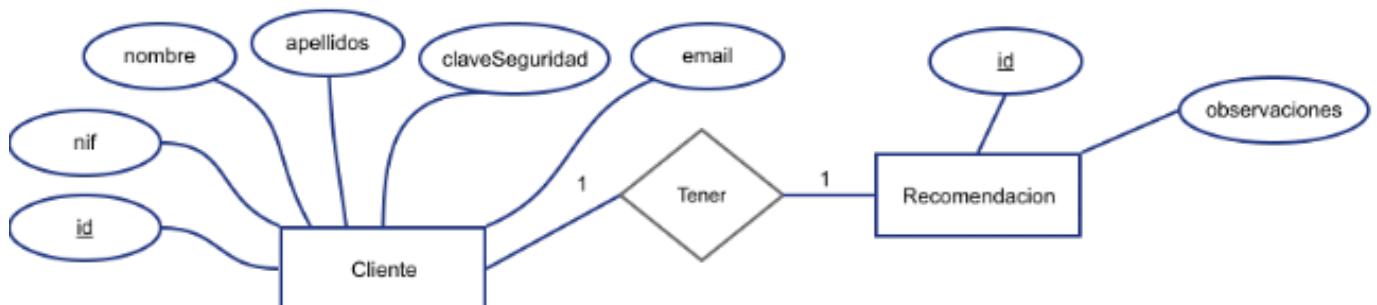
DISEÑANDO LA APLICACIÓN SPRING APLICANDO LOS PATRONES VISTOS.

A partir de los patrones explicados vamos a trabajar sobre un ejemplo, de forma que implementaremos una pequeña parte de dicho ejemplo, aunque la idea es usarlo para, en el siguiente tema, implementar JPA con Hibernate y mapear todas las tablas.

El ejemplo concreto a usar será el siguiente:



En nuestro caso vamos a usar la siguiente pequeña porción del ejemplo:



Los pasos a seguir para desarrollar el proyecto serán:

1. Definir e implementar las entidades
2. Definir e implementar los DTOs
3. Definir e implementar los controladores
4. Definir e implementar los servicios

5. Definir e implementar los repositorios
6. Definir e implementar las vistas

Definir e implementar el modelo

El primer paso es realizar la implementación del modelo por la parte de las entidades, es decir, mapear las dos tablas (tener en cuenta que no estamos trabajando todavía con bases de datos) que tenemos en clases Java con sus atributos. A estos mapeos hay que añadirles los métodos `getter/setter`, un constructor por defecto (o sea el vacío) y el método `toString()`. Quedaría de la siguiente forma:

```
1 package com.example.demo.repository.entity;
2
3 public class Cliente {
4
5     private Long id;
6     private String nif;
7     private String nombre;
8     private String apellidos;
9     private String claveSeguridad;
10    private String email;
11
12    // Contructor por defecto
13    public Cliente() {
14        super();
15    }
16
17    // Getter/Setter
18
19    public Long getId() {
20        return id;
21    }
22
23    public void setId(Long id) {
24        this.id = id;
25    }
26
27    public String getNif() {
28        return nif;
29    }
30
31    public void setNif(String nif) {
32        this.nif = nif;
33    }
34
35    public String getNombre() {
36        return nombre;
37    }
38
39    public void setNombre(String nombre) {
```

```

40     this.nombre = nombre;
41 }
42
43 public String getApellidos() {
44     return apellidos;
45 }
46
47 public void setApellidos(String apellidos) {
48     this.apellidos = apellidos;
49 }
50
51 public String getClaveSeguridad() {
52     return claveSeguridad;
53 }
54
55 public void setClaveSeguridad(String claveSeguridad) {
56     this.claveSeguridad = claveSeguridad;
57 }
58
59 public String getEmail() {
60     return email;
61 }
62
63 public void setEmail(String email) {
64     this.email = email;
65 }
66
67 // Método toString
68 @Override
69 public String toString() {
70     return "Cliente [id=" + id + ", nif=" + nif + ", nombre=" + nombre + ",
71     apellidos=" + apellidos
72         + ", claveSeguridad=" + claveSeguridad + ", email=" + email + "]";
73 }

```

Como podemos comprobar, la implementación de los modelos se puede hacer demasiado extensa. Por ello, vamos a explotar el uso de las anotaciones que nos permite Lombok, disminuyendo considerablemente el código.

Borramos de la entidad `cliente.java` todos los `getter/setter` y sobre la clase ponemos las anotaciones `@Getter` y `@Setter`. La clase quedaría de la siguiente forma (reduciendo considerablemente el tamaño de la misma):

```

1 package com.example.demo.repository.entity;
2
3 import lombok.Getter;
4 import lombok.Setter;
5
6 @Getter
7 @Setter

```

```

8 public class Cliente {
9
10    private Long id;
11    private String nif;
12    private String nombre;
13    private String apellidos;
14    private String claveSeguridad;
15    private String email;
16
17    // Contructor por defecto
18    public Cliente() {
19        super();
20    }
21
22    // Método toString
23    @Override
24    public String toString() {
25        return "Cliente [id=" + id + ", nif=" + nif + ", nombre=" + nombre + ",
26        apellidos=" + apellidos
27            + ", claveSeguridad=" + claveSeguridad + ", email=" + email + "]";
28    }

```

En este caso hemos realizado las anotaciones a nivel de clase, pero podemos anotar cualquier campo con las anotaciones `@Getter` y `@Setter` para generar automáticamente los métodos `getter` y `setter` de dicho campo.

Los métodos `get` y `set` serán públicos a no ser que se especifique pasándole un parámetro a la anotación a través de la clase `AccessLevel`. Los niveles de acceso que se permiten son `PUBLIC`, `PROTECTED`, `PACKAGE` y `PRIVATE`. También podemos desactivar la generación de los métodos `get` y `set` para un determinado campo si ponemos el nivel de acceso `NONE`. Por ejemplo:

```

1 package com.example.demo.repository.entity;
2
3 import lombok.Getter;
4 import lombok.Setter;
5
6 @Getter
7 @Setter
8 public class Cliente {
9
10    @Getter @Setter private Long id;
11    @Getter @Setter private String nif;
12    @Getter @Setter private String nombre;
13    @Getter @Setter private String apellidos;
14    @Getter @Setter(lombok.AccessLevel.NONE) private String claveSeguridad;
15    @Setter(lombok.AccessLevel.PROTECTED) private String email;
16
17    // Contructor por defecto
18    public Cliente() {
19        super();

```

```

20 }
21
22 // Método toString
23 @Override
24 public String toString() {
25     return "Cliente [id=" + id + ", nif=" + nif + ", nombre=" + nombre + ",
26 apellidos=" + apellidos
27     + ", claveSeguridad=" + claveSeguridad + ", email=" + email + "]";
28 }

```

Fijaros que hemos realizado algunos cambios en la clase, solo como ejemplo, para poder indicar como podemos realizar las anotaciones a nivel de campo. En nuestro caso lo dejaremos a nivel de clase, ya que nos interesa generarla para todos los campos. Cuando comencemos a trabajar con JPA/Hibernate volveremos a hablar de estas anotaciones.

Por otro lado, cualquier clase puede ser anotada con `@ToString` para generar una implementación del método `toString()`. De forma predeterminada, imprimirá el nombre de la clase, junto con cada campo, en orden, separados por comas.

Podemos establecer el parámetro `includeFieldNames`, lo cual nos puede dar mayor legibilidad, indicando de esta forma que incluya también los nombres de los campos.

Por defecto, se imprimirán todos los campos no estáticos. Si desea omitir algunos campos, puede anotar estos campos con `@ToString.Exclude`. De forma alternativa, se puede especificar exactamente qué campos desea utilizar con `onlyExplicitlyIncluded = true` y luego marcar cada campo que desee incluir con `@ToString.Include`. También podemos obtener el resultado del método `toString()` de la clase `Super` indicando el parámetro `callSuper=true`.

En nuestro caso, solo añadiremos que incluya el nombre de los campos, por lo que quedará de la siguiente forma:

```

1 package com.example.demo.repository.entity;
2
3 import lombok.Getter;
4 import lombok.Setter;
5 import lombok.ToString;
6
7 @Getter
8 @Setter
9 @ToString(includeFieldNames=true)
10 public class Cliente {
11
12     private Long id;
13     private String nif;
14     private String nombre;
15     private String apellidos;
16     private String claveSeguridad;
17     private String email;
18
19     // Contructor por defecto

```

```

20  public Cliente() {
21      super();
22  }
23 }
```

También disponemos de la anotación `@NotNull` que controla que el parámetro de entrada de un método o de un constructor sea chequeado/controlado que no sea nulo. Esta anotación realiza esta sentencia:

```
if (param == null) throw new NullPointerException("param");
```

Hay que tener en cuenta que para los constructores el chequeo se realizará inmediatamente después de cualquier llamada al `this()` o `super()`.

Todavía no hemos acabado de optimizar esto: disponemos también de las anotaciones `@NoArgsConstructor`, `@RequiredArgsConstructor` y `@AllArgsConstructor`. Estas anotaciones hacen referencia a constructores que no toman argumentos, constructores con un argumento por campo final/no nulo, o constructores con un argumento por cada campo.

- `@NoArgsConstructor` generará un constructor sin parámetros. Si esto no es posible (debido a los campos finales), en su lugar se producirá un error de compilación, a menos que se utilice `@NoArgsConstructor (force = true)`.
Luego todos los campos finales se inicializan con 0/falso/nulo. Para los campos con restricciones, como los campos `@NotNull` no se genera ninguna verificación, así que hay que tener en cuenta que estas restricciones generalmente no se cumplirán hasta que esos campos se inicien correctamente más tarde.
- `@RequiredArgsConstructor` genera un constructor con un parámetro para cada campo que requiera de un manejo especial (por ejemplo, que sea `final` o que lleve la anotación `@NotNull`). Todos los campos finales no inicializados obtienen un parámetro, así como también los campos que están marcados como `@NotNull` que no se inicializan donde se declaran. Este es el ideal que usaremos en nuestra clase (más adelante la añadiremos).
- `@AllArgsConstructor` genera un constructor con un parámetro para cada campo en su clase.

Cada una de estas anotaciones permite una forma alternativa, donde el constructor generado es siempre privado, y se genera un método estático adicional que envuelve al constructor privado.

Este modo se habilita suministrando el valor de `staticName` para la anotación, así:

```
@RequiredArgsConstructor (staticName = "additional").
```

Una de las últimas anotaciones que tenemos es `@EqualsAndHashCode`, que proporciona la generación de los métodos `equals()` y `hashCode()` a partir de los campos del objeto. De forma predeterminada para generar estos métodos utiliza todos los campos de la clase no transitorios y no estáticos, aunque podemos modificar los campos que se utilizan utilizando `@EqualsAndHashCode.Include` o `@EqualsAndHashCode.Exclude`.

Si utilizamos el `Include` adicionalmente tenemos que indicar lo siguiente `onlyExplicitlyIncluded = true`.

Recordamos que el método `equals()` comprueba si dos objetos son del mismo tipo y si alguno de los campos que hemos definido en la comparación coincide.

El método `hashCode()` viene a complementar al método `equals()` y sirve para comparar objetos de una forma más rápida en estructuras Hash ya que únicamente nos devuelve un número. Cuando Java compara dos objetos en estructuras de tipo hash (HashMap, HashSet etc) primero invoca al método `hashCode()` y luego el `equals()`. Si los métodos hashCode de cada objeto devuelven diferente hash no seguirá comparando y considerará a los objetos distintos. En el caso en el que ambos objetos comparten el mismo `hashCode()` Java invocará al método `equals()` y revisará a detalle si se cumple la igualdad. De esta forma las búsquedas quedan simplificadas en estructuras hash.

Tanto `equals()` como `hashCode()` son obligatorios de implementar, por lo que añadiremos las anotaciones a la clase, junto a la anotación del constructor:

```
1 package com.example.demo.repository.entity;
2
3 import lombok.EqualsAndHashCode;
4 import lombok.Getter;
5 import lombok.RequiredArgsConstructor;
6 import lombok.Setter;
7 import lombok.ToString;
8
9 @Getter
10 @Setter
11 @ToString(includeFieldNames=true)
12 @RequiredArgsConstructor
13 @EqualsAndHashCode
14 public class Cliente {
15
16     private Long id;
17     private String nif;
18     private String nombre;
19     private String apellidos;
20     private String claveSeguridad;
21     private String email;
22 }
```

Estas anotaciones serían las indispensables para nuestro POJO, permitiendo, además, simplificarlo lo máximo posible. Con todo esto, y tras haber entendido las distintas anotaciones, acabamos indicando que disponemos de la anotación `@Data` que agrupa las características de `@ToString`, `@EqualsAndHashCode`, `@Getter/@Setter` y `@RequiredArgsConstructor` juntas: en otras palabras, `@Data` genera todos los estándares que normalmente se asocian con los POJOs y los BEANs. Quedará de la siguiente forma:

```
1 package com.example.demo.repository.entity;
2
3 import lombok.Data;
4
5 @Data
6 public class Cliente {
7
8     private Long id;
9     private String nif;
10    private String nombre;
```

```

11  private String apellidos;
12  private String claveSeguridad;
13  private String email;
14 }

```

La diferencia entre los POJOs y los BEANs es que los POJOs no tienen otras restricciones mientras que los BEANs son POJOs especiales con algunas restricciones (por ejemplo, el BEAN es un POJO serializable, que además puede tener un constructor sin argumentos).

Implementamos también la clase `Recomendacion.java` siguiendo lo descrito con anterioridad, incluyendo, además, la relación que mantiene con la clase `Cliente.java`:

```

1 package com.example.demo.repository.entity;
2
3 import lombok.Data;
4
5 @Data
6 public class Recomendacion {
7
8     private Long id;
9     private String observaciones;
10    private Cliente cliente;
11 }

```

Modificamos la clase `Cliente.java` para añadir la relación con la clase `Recomendacion.java`:

```

1 package com.example.demo.repository.entity;
2
3 import lombok.Data;
4
5 @Data
6 public class Cliente {
7
8     private Long id;
9     private String nif;
10    private String nombre;
11    private String apellidos;
12    private String claveSeguridad;
13    private String email;
14    private Recomendacion recomendacion;
15 }

```

IMPORTANTE: Tras añadir la relación, si recordamos, el método `toString()` lo genera automáticamente, por lo que, para un objeto `Cliente` mostrará el contenido del objeto `Recomendacion`, que al mismo tiempo volverá a mostrar el contenido del `Cliente` que tiene el objeto `Recomendacion`...y continua así hasta que la pila se desborda.

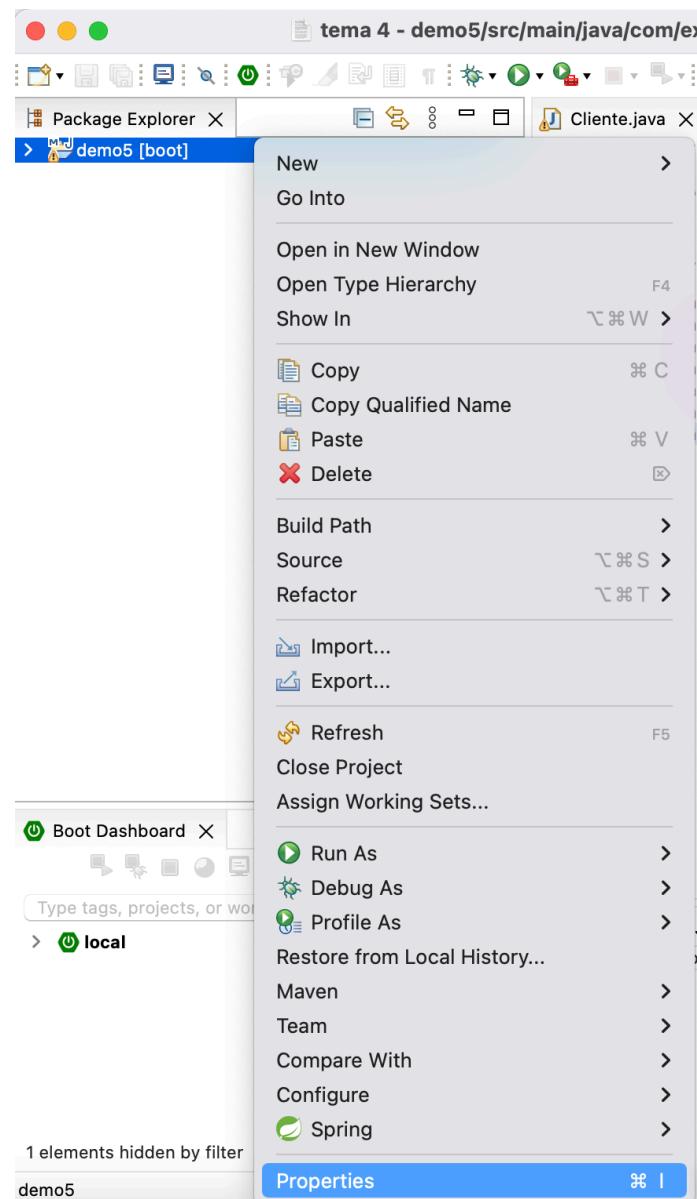
Para evitar este problema utilizaremos la anotación `@ToString.Exclude` en el objeto `cliente` que se encuentra en la clase `Recomendacion`. Quará de la siguiente forma:

```

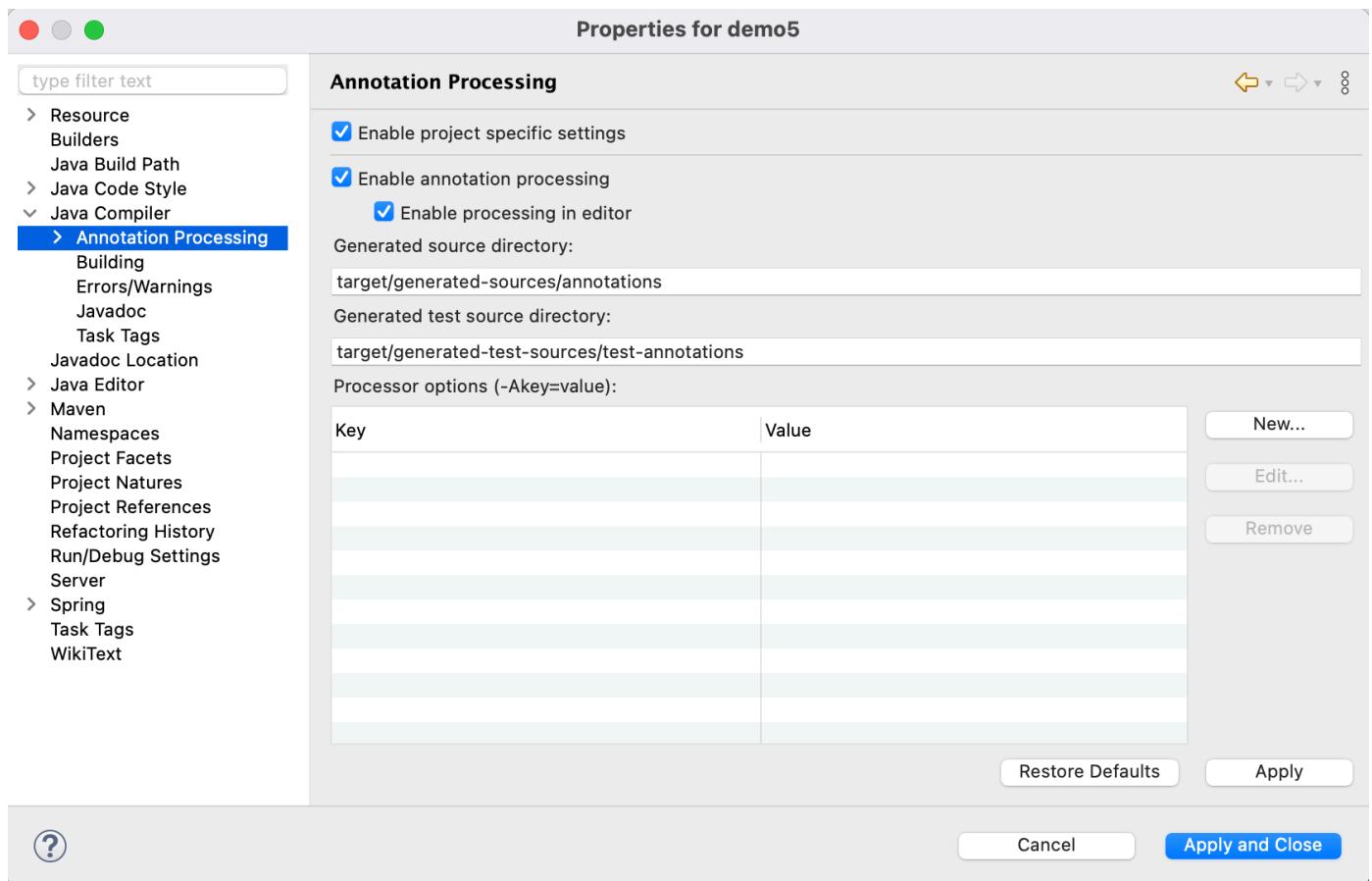
1 package com.example.demo.repository.entity;
2
3 import lombok.Data;
4 import lombok.ToString;
5
6 @Data
7 public class Recomendacion {
8
9     private Long id;
10    private String observaciones;
11    @ToString.Exclude
12    private Cliente cliente;
13 }

```

En este momento es necesario activar las anotaciones en el proyecto, tanto en tiempo de ejecución como en tiempo de edición, por lo que sobre el proyecto pulsamos con el botón derecho y seleccionamos la opción Properties.

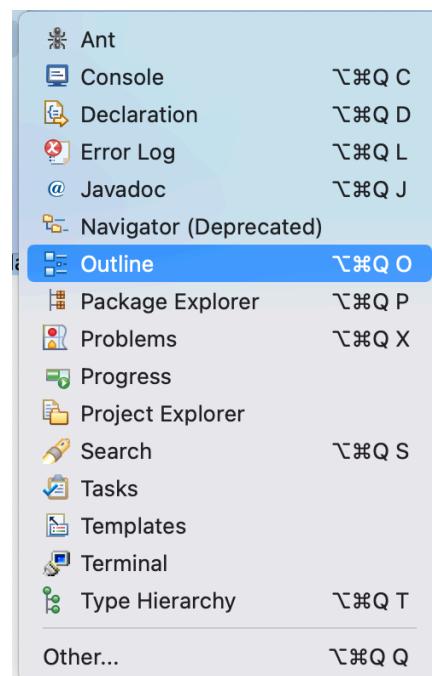


Tras esto nos vamos al apartado *Java Compiler > Annotation Processing* y activamos la opción *Enable annotation processing* y *Enable preprocessing in editor*. Pulsamos *Apply and Close* (si ya está activado lo dejamos como está).



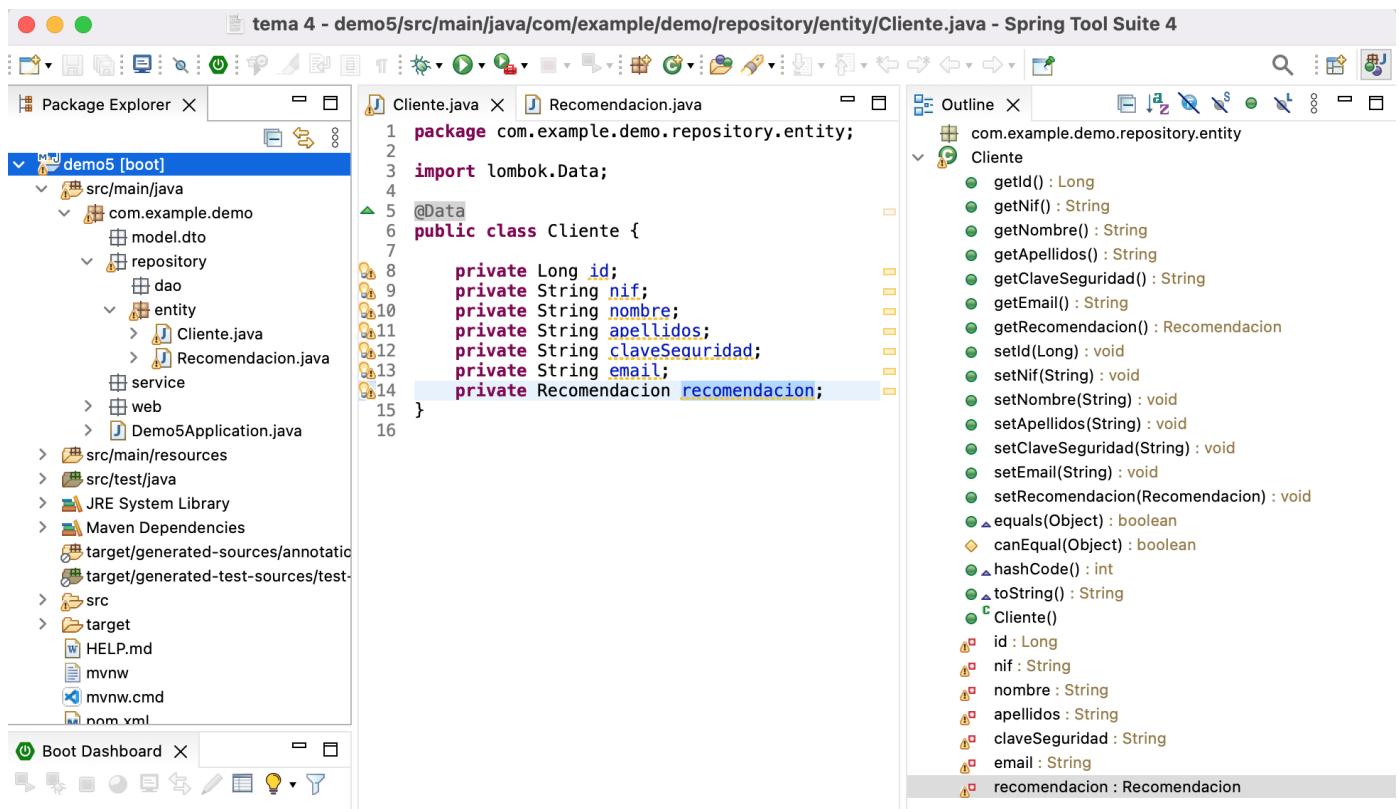
Reiniciamos de nuevo STS y el proyecto ya funcionará con el plugin de forma correcta. NOTA: Es importante reiniciar para que funcione correctamente.

Abrimos la vista *Outline* desde *Window > Show View > Outline*. También podemos encontrarlo en *Window > Show View > Other* donde seleccionamos *General > Outline*.



La vista de resumen (*outline*) es una forma rápida de ver qué métodos y atributos se encuentran definidos dentro de una clase de Java. Los iconos asociados proporcionan información adicional de acuerdo con la visibilidad del atributo o método en cuestión. Y sólo con hacer clic en cualquiera de estos iconos conducirá a la línea de código exacta en que dicho atributo o método está definido. La vista de resumen es una herramienta esencial para entender y navegar archivos Java voluminosos.

Al abrir dicha vista podemos comprobar que, aunque no hemos definido los `getter/setter`, `constructor`, `toString`, `equals`, etc... disponemos de ellos, puesto que la vista outline nos lo está proporcionando:



Definir e implementar el DTO

El patrón DTO tiene como finalidad de crear un objeto plano (POJO) con una serie de atributos que puedan ser enviados o recuperados del servidor en **una sola invocación**, de tal forma que un DTO puede contener información de **múltiples fuentes** o tablas y concentrarlas en una única clase simple.

Si bien un DTO es simplemente un objeto plano, sí que tiene que cumplir algunas reglas para poder considerar que hemos creado un DTO correctamente implementado:

- **Solo lectura:** Dado que el objetivo de un DTO es utilizarlo como un objeto de transferencia entre el cliente y el servidor, es importante evitar tener operaciones de negocio o métodos que realicen cálculos sobre los datos, es por ello que solo deberemos de tener los métodos GET y SET de los respectivos atributos del DTO (similar a un POJO, vamos).
- **Serializable:** Es claro que, si los objetos tendrán que viajar por la red, deberán de poder ser serializables, pero no hablamos solamente de la clase en sí, sino que también todos los atributos que contenga el DTO deberán ser fácilmente serializables. Un error clásico en Java es, por ejemplo, crear atributos de tipo Date o Calendar para transmitir la fecha u hora, ya que estos no tienen una forma

estándar para serializarse por ejemplo en Webservices o REST. Vamos a explicar que en Java, un objeto serializable es aquel que puede ser convertido en una secuencia de bytes, lo que permite que el estado del objeto sea almacenado o transmitido y luego restaurado a su estado original. Esto es útil, por ejemplo, para guardar objetos en archivos, transmitir objetos a través de redes o enviarlos entre procesos o sistemas distribuidos.

Un error muy frecuente entre programadores inexpertos es el hecho de utilizar las clases de Entidad para utilizarlos para la transmisión de datos entre el cliente y el servidor. Solo para entrar en contexto, las entidades son clases que representan al modelo de datos, o mapea directamente contra una tabla de la base de datos. Dicho esto, las entidades son clases que fueron diseñadas para mapear contra la base de datos, no para ser una vista para una pantalla o servicio determinado, lo que provoca que muchos de los campos no puedan ser serializables o que no contengan todos los campos necesarios para un servicio, ya sea que tengan de más o de menos.

El hecho de que las entidades no contengan todos los atributos necesarios o que no sean serializables trae otros problemas, como la necesidad de agregar más atributos a las entidades con el único objetivo de poder cubrir los requerimientos de transferencia de datos, dejando de lado el verdadero propósito de la entidad, que es únicamente mapear contra la base de datos, lo que va llevando lentamente a ir creando una mezcla entre Entidad y DTO.

Para poder diferenciar entre una entidad y un objeto DTO una buena práctica es marcar las clases con el sufijo DTO para recordar su significado, así la clase `Cliente` quedaría como `ClienteDTO`.

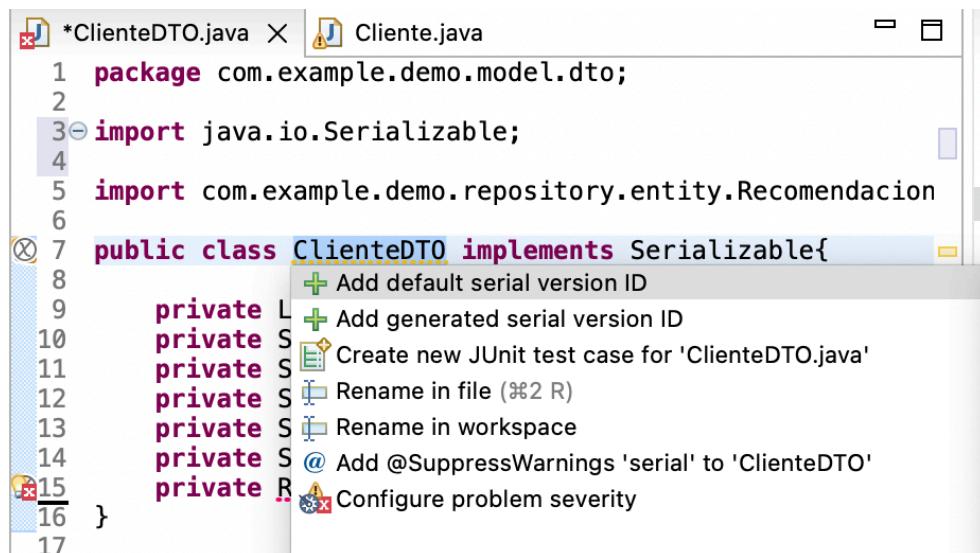
En nuestro ejemplo, el objetivo es integrar las entidades `Cliente` y `Recomendacion` en una sola clase DTO denominada `ClienteDTO`.

Por lo que creamos la nueva clase `ClienteDTO.java` en el paquete `model.dto` y la definimos 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.Recomendacion;
6
7 public class ClienteDTO implements Serializable{
8
9     private static final long serialVersionUID = 1L;
10    private Long id;
11    private String nif;
12    private String nombre;
13    private String apellidos;
14    private String claveSeguridad;
15    private String email;
16    private RecomendacionDTO recomendacionDTO;
17 }
```

Fijaros que los atributos de la entidad `ClienteDTO` pertenecen mayoritariamente a la clase `Cliente.java`. Fijaros también que hemos definido un atributo de tipo `RecomendacionDTO` que hace referencia a una nueva clase que tenemos que crear, que es `RecomendacionDTO.java`. Como no la tenemos creada todavía nos estará dando un error, pero no os preocupéis, la vamos a crear a continuación.

Una de las características de los DTO's es que han de ser objetos **Serializable** para así poder viajar por la red. Esta característica nos falta indicarla en los DTO's, por lo que vamos a añadir que implementen la interfaz `Serializable`, y con ello, la propiedad **UID** que identifica la versión de cada objeto transportado (en el código anteriormente ya lo hemos añadido). Esta propiedad la podemos añadir automáticamente tras hacer la clase `Serializable` con el botón derecho sobre el warning que nos indica en la clase.



Tras esto creamos la clase `RecomendaciónDTO.java` y la definimos de la siguiente forma:

```
1 package com.example.demo.model.dto;
2
3 import java.io.Serializable;
4
5 public class RecomendacionDTO implements Serializable{
6
7     private static final long serialVersionUID = 1L;
8     private Long id;
9     private String observaciones;
10}
```

Ahora necesitamos poder construir este tipo de objetos, poder acceder a sus campos, compararlos, etc... Por ello haremos uso, de nuevo, de la anotación `@Data` para poder asignarles todas estas características (añadiremos la anotación a las dos clases):

```
1 package com.example.demo.model.dto;
2
3 import java.io.Serializable;
4
5 import com.example.demo.repository.entity.Recomendacion;
6
7 import lombok.Data;
8
9 @Data
10 public class ClienteDTO implements Serializable{
```

```

12  private static final long serialVersionUID = 1L;
13  private Long id;
14  private String nif;
15  private String nombre;
16  private String apellidos;
17  private String claveSeguridad;
18  private String email;
19  private RecomendacionDTO recomendacionDTO;
20 }

```

Crearemos el DTO para `Recomendacion` indicando, de nuevo, la anotación `@ToString.Exclude` para el atributo de tipo `clienteDTO`. Esto lo haremos en todos los atributos que apunten a otra clase, o sea una lista de objetos de otra clase. Así nos ahorraremos un error de referencias circulares (un cliente apunta a una recomendación que apunta a un cliente que apunta a una recomendación...y así hasta desbordar la pila). Hacemos esto también con `clienteDTO.java`.

```

1 package com.example.demo.model.dto;
2
3 import java.io.Serializable;
4
5 import lombok.Data;
6 import lombok.ToString;
7
8 @Data
9 public class RecomendacionDTO implements Serializable{
10
11     private static final long serialVersionUID = 1L;
12     private Long id;
13     private String observaciones;
14     @ToString.Exclude
15     private ClienteDTO clienteDTO;
16 }

```

Si recordamos la imagen inicial de las distintas capas nos vamos a encontrar con que entre la capa web y la capa de servicio los objetos con los que se trabaja son los DTO, mientras que en la capa de repositorio y la capa de servicio los objetos que trabajan son las entidades. Esto quiere decir que la capa de servicio va a tener que poder convertir de entidades a DTOs y de DTOs a entidades. Por ello vamos a crear una serie de métodos estáticos en las clases DTO que nos van a permitir realizar este tipo de conversiones (**entity->DTO**, **DTO->entity**). Los métodos se van a llamar de la siguiente forma:

- `convertToEntity`
- `convertToDTO`

De hecho, cuando comenzemos por `clienteDTO.java` veremos que nos sale un error, y es que no hemos definido dichos métodos de mapeo en `RecomendacionDTO.java`. 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
7 import lombok.Data;
8 import lombok.ToString;
9
10 @Data
11 public class ClienteDTO implements Serializable{
12
13     private static final long serialVersionUID = 1L;
14     private Long id;
15     private String nif;
16     private String nombre;
17     private String apellidos;
18     private String claveSeguridad;
19     private String email;
20     @ToString.Exclude
21     private RecomendacionDTO recomendacionDTO;
22
23     //Convierte una entidad a un objeto DTO
24     public static ClienteDTO convertToDTO(Cliente cliente) {
25         // Creamos el clienteDTO y asignamos los valores basicos
26         ClienteDTO clienteDTO = new ClienteDTO();
27         clienteDTO.setId(cliente.getId());
28         clienteDTO.setNif(cliente.getNif());
29         clienteDTO.setNombre(cliente.getNombre());
30         clienteDTO.setApellidos(cliente.getApellidos());
31         clienteDTO.setClaveSeguridad(cliente.getClaveSeguridad());
32         clienteDTO.setEmail(cliente.getEmail());
33         // Asignamos la recomendacionDTO pasandole el clienteDTO como parametro
34
35         clienteDTO.setRecomendacionDTO(RecomendacionDTO.convertToDTO(cliente.getRecomendacion()
36             (), clienteDTO));
37
38     }
39
40     // Convierte un objeto DTO a una entidad
41     public static Cliente convertToEntity(ClienteDTO clienteDTO) {
42         // Creamos la entidad cliente y le asignamos los valores
43         Cliente cliente = new Cliente();
44         cliente.setId(clienteDTO.getId());
45         cliente.setNif(clienteDTO.getNif());
46         cliente.setNombre(clienteDTO.getNombre());
47         cliente.setApellidos(clienteDTO.getApellidos());
48         cliente.setClaveSeguridad(clienteDTO.getClaveSeguridad());
49         cliente.setEmail(clienteDTO.getEmail());
50         // Asignamos la recomendacion pasandole el cliente como parametro
51
52         cliente.setRecomendacion(RecomendacionDTO.convertToEntity(clienteDTO.getRecomendacion
53             DTO(), cliente));

```

```
53     // Retorna la entidad
54     return cliente;
55 }
56 }
```

```
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.Recomendacion;
7
8 import lombok.Data;
9 import lombok.ToString;
10
11 @Data
12 public class RecomendacionDTO implements Serializable{
13
14     private static final long serialVersionUID = 1L;
15     private Long id;
16     private String observaciones;
17     @ToString.Exclude
18     private ClienteDTO clienteDTO;
19
20     //Convierte una entidad a un objeto DTO
21     public static RecomendacionDTO convertToDTO(Recomendacion recomendacion, ClienteDTO
clienteDTO) {
22         // Creamos el objeto recomendacionDTO y asignamos los basicos
23         RecomendacionDTO recomendacionDTO = new RecomendacionDTO();
24         recomendacionDTO.setId(recomendacion.getId());
25         recomendacionDTO.setObservaciones(recomendacion.getObservaciones());
26         recomendacionDTO.setClienteDTO(clienteDTO);
27
28         // Retorna el DTO
29         return recomendacionDTO;
30     }
31
32     // Convierte un objeto DTO a una entidad
33     public static Recomendacion convertToEntity(RecomendacionDTO recomendacionDTO,
Cliente cliente) {
34         // Creamos la entidad Recomendacion y le asignamos los valores
35         Recomendacion recomendacion = new Recomendacion();
36         recomendacion.setId(recomendacionDTO.getId());
37         recomendacion.setObservaciones(recomendacionDTO.getObservaciones());
38         recomendacion.setCliente(cliente);
39
40         // Retorna el entity
41         return recomendacion;
42     }
```

Podemos observar que el método es estático, y esto nos sirve para poder usarlo en cualquier momento sin tener que haber instanciado un objeto de la clase.

Fijaros también que la vista *outline* está mostrando los distintos métodos que permite `lombok` mediante la anotación `@Data`.

Tener en cuenta tambien que hemos realizado una propuesta de métodos `convertToDTO` y `convertToEntity`, pero puede haber más propuestas recibiendo parámetros distintos.

Ya tenemos creada la clase que se encargará de transportar los datos de una capa a otra sin tener que exponer las entidades a las capas superiores.

Definir e implementar el controlador

Vamos a definir ahora la capa de más alto nivel, la capa web, donde expondremos los controladores de la aplicación.

La aplicación que vamos a crear tendrá un comportamiento completo en cuanto al mantenimiento de los clientes, así como su recomendación, por lo que podremos realizar las operaciones CRUD:

- listar los clientes
- visualizar su información
- dar de alta clientes
- actualizar datos de clientes
- borrar los clientes

Por ello comenzaremos creando dos controladores: `IndexController.java` y `ClienteController.java` en el paquete `controller`. El controlador `clienteController.java` implementará las 4 operaciones correspondientes a CRUD: `create`, `read`, `update` y `delete`. El controlador `IndexController.java` implementará una operación que mostrará la página inicial de la aplicación, con un enlace al mantenimiento de clientes.

Tras ello, pondremos la anotación `@Controller` en las dos clases controladores. Quedará todo de la siguiente forma:

```

1 package com.example.demo.web.controller;
2
3 import org.springframework.stereotype.Controller;
4
5 @Controller
6 public class ClienteController {
7
8 }
```

```

1 package com.example.demo.web.controller;
2
3 import org.springframework.stereotype.Controller;
4
5 @Controller
6 public class IndexController {
7
8 }
```

Ahora vamos a definir las operaciones en los distintos controladores.

Mostrar página inicial de la aplicación

Este método del controlador se encargará de mapear la URL inicial de la aplicación, y le pasará el título de la aplicación y el nombre de la asignatura, que los declararemos en el fichero `application.properties`, junto al puerto del servidor de aplicaciones. Ya lo hemos hecho en el tema anterior, y quedará de la siguiente forma:

```

1 package com.example.demo.web.controller;
2
3 import org.slf4j.Logger;
4 import org.slf4j.LoggerFactory;
5 import org.springframework.beans.factory.annotation.Value;
6 import org.springframework.stereotype.Controller;
7 import org.springframework.web.bind.annotation.GetMapping;
8 import org.springframework.web.servlet.ModelAndView;
9
10 @Controller
11 public class IndexController {
12
13     private static final Logger log = LoggerFactory.getLogger(IndexController.class);
14
15     @Value("${aplicacion.nombre}")
16     private String nombreAplicacion;
17
18     @Value("${asignatura}")
19     private String nombreAsignatura;
20
21     @GetMapping("/")
22     public ModelAndView index() {
23         log.info("IndexController - index: Mostramos la pagina inicial");
24
25         ModelAndView mav = new ModelAndView("index");
26         mav.addObject("titulo", nombreAplicacion);
27         mav.addObject("nombreAsignatura", nombreAsignatura);
28
29         return mav;
30     }
31 }
```

El fichero `application.properties` será el siguiente:

```
1 spring.application.name=demo5
2 server.port = 8888
3 aplicacion.nombre = Demo - SpringBoot CRUD MVC
4 asignatura = Servidor
```

Ya crearemos más adelante las distintas vistas, por ahora solo nos vamos a centrar en la funcionalidad.

Operación “Listar todos los clientes”

Un método común también es proporcionar una lista de objetos (en este caso clientes) a la vista. Normalmente, se agregará una paginación o algún tipo de filtro. Sin embargo, en este ejemplo, solo queremos mostrar un ejemplo simple de listar clientes.

```
1 package com.example.demo.web.controller;
2
3 import org.slf4j.Logger;
4 import org.slf4j.LoggerFactory;
5 import org.springframework.stereotype.Controller;
6 import org.springframework.web.bind.annotation.GetMapping;
7 import org.springframework.web.servlet.ModelAndView;
8
9 @Controller
10 public class ClienteController {
11
12     private static final Logger log = LoggerFactory.getLogger(ClienteController.class);
13
14     // Listar los clientes
15     @GetMapping("/clientes")
16     public ModelAndView findAll() {
17
18         log.info("ClienteController - findAll: Mostramos todos los clientes");
19
20         ModelAndView mav = new ModelAndView("clientes");
21         List<ClienteDTO> listaClientesDTO = clienteService.findAll();
22         mav.addObject("listaClientesDTO", listaClientesDTO);
23
24         return mav;
25     }
26 }
```

Hemos asignado este método de controlador a la vista `clientes`. Solicitamos al servicio de clientes una lista de todos los clientes y la adjuntamos como un atributo denominado list al modelo.

IMPORTANTE: No os preocupéis si os sale un error en la clase `ClienteService`. El problema es que no está creada todavía ni definidos los métodos con los que tenemos que trabajar.

Operación “Obtener Cliente por id”

En este caso vamos a definir un método que nos permitirá mostrar un formulario con un objeto de tipo `ClienteDTO` que ha sido obtenido mediante su id.

Pensar que tendremos un formulario con una lista de clientes, y al pinchar en uno de ellos nos llevará a un formulario con todos los datos del cliente seleccionado. Para realizar esto utilizaremos esta operación de la que estamos hablando.

El controlador definirá una operación que recibirá el id del cliente seleccionado, invocará a la capa de servicios a que devuelva el cliente y retornará la vista con los datos del cliente seleccionado. El código será el siguiente:

```
1 // Visualizar la informacion de un cliente
2 @GetMapping("/clientes/{idCliente}")
3 public ModelAndView findById(@PathVariable("idCliente") Long idCliente) {
4
5     log.info("ClienteController - findById: Mostramos la informacion del cliente: " +
6     idCliente);
7
8     // Obtenemos el cliente y lo pasamos al modelo
9     ClienteDTO clienteDTO = new ClienteDTO();
10    clienteDTO.setId(idCliente);
11    clienteDTO = clienteService.findById(clienteDTO);
12
13    ModelAndView mav = new ModelAndView("clienteshow");
14    mav.addObject("clienteDTO", clienteDTO);
15
16    return mav;
17 }
```

Ahora, estamos usando una nueva anotación denominada `@PathVariable` para injectar el valor del id del cliente en la ruta de la URL en nuestro controlador como la variable `idCliente` (realmente la URL será `/clientes/{idCliente}`), permitiendo disponer del valor de `idCliente` en el método.

A continuación, le pedimos al servicio de clientes (recordamos que el error se sigue mostrando puesto que no hemos creado todavía el servicio de clientes) que obtenga el cliente a partir del Id, que se encuentra en un `ClienteDTO` que hemos creado y que le hemos asignado el id que tenemos, y el resultado pasa ser devuelto al mismo objeto.

Fijaros que entre la capa controlador y la capa de servicio están viajando objetos DTO (Data Transfer Object), por lo que encaja a la perfección con el modelo propuesto.

Tras tener el cliente lo añadimos al mapa de objetos del modelo y retornamos el objeto `ModelAndView`.

Diferencia entre @PathVariable y @RequestParam

En desarrollo web existen 2 conceptos bastante bien definidos: los parámetros de consulta (query parameters) y las variables de URI (URI parameters). Los URI pueden ser localizadores de recursos uniformes (URL), nombres de recursos uniformes (URN), o ambos.

Los parámetros de consulta son usados (por lo general) para filtrar una petición. Son datos adicionales a la URI del recurso, en formato de par nombre=valor, separados por el carácter & y que se separan de la dirección URL usando el carácter ?. Por ejemplo:

<https://www.google.com/webhp?hl=es> // Castellano

Como se observa, el componente URI es www.google.com/webhp, y el parámetro de consulta es hl=es. Si deseamos cambiar el idioma en que se presenta esta página de Google, simplemente basta con cambiar el parámetro de búsqueda a otro valor, por ejemplo:

<https://www.google.com/webhp?hl=ca> // Catalán

Las variables URI, por otro lado, se usan por lo general para identificar un recurso específico dentro del sistema. De esta forma se diferencian las entidades o modelos de datos sobre los cuales se realizan operaciones. Por ejemplo:

<https://dominio.ext/user> // Ruta al modelo User

<https://dominio.ext/car> // Ruta al modelo Car

Podemos usar una variable URI para traer un dato específico, por ejemplo:

<https://dominio.ext/user/id> //Ruta al modelo User, cuyo identificador es id

¿Cuándo debemos usar una u otra?: `@RequestParam` y `@PathVariable` nos van a servir para extraer la información de acuerdo a su tipo, es decir, ya sean datos pasados como parámetros de consulta o como variables URI.

- `@RequestParam`: nos permite extraer los parámetros de la consulta (par nombre=valor), además que esta anotación de Spring nos permitirá establecer valores por defecto en caso de que la consulta no contenga parámetros. Por ejemplo:

```
1  @GetMapping("/clientes")
2  public ModelAndView findByName(@RequestParam(value = "nombre", defaultValue =
3      "Gabriel") String nombre) {
4      ...
}
```

En el ejemplo se espera que la consulta tenga definido un par nombre=valor, y si no lo tiene se establece uno por defecto.

Un ejemplo de URI para consumir esta ruta es:

</clientes?nombre=Gabriel>

En caso de no querer indicar un valor por defecto quedará de la siguiente forma:

```
1 @GetMapping("/clientes")
2 public ModelAndView findByName(@RequestParam String nombre) {
3     ...
4 }
```

- `@PathVariable`: Nos permite extraer la información que es parte de la estructura de la URI pero que no se trata como un par nombre=valor como el caso anterior.

Para usar esta anotación debemos indicarle de antemano al controlador que la estructura de la ruta contiene variables, mediante el uso del identificador {<variable_name>} donde <variable_name> representa precisamente la variable a ser analizada y/o extraída de la ruta. Por ejemplo:

```
1 @GetMapping("/clientes/{id}")
2 public ModelAndView findById(@RequestParam Long id) {
3     ...
4 }
```

¿Son intercambiables?: A simple vista parece no haber una diferencia en su uso, ambas extraen información de la URI. Sin embargo, tratando de tener un buen diseño orientado a Servicios Web de tipo REST, usaremos las variables URI para identificar un recurso único en nuestro sistema, mientras que los parámetros de consulta los usaremos para filtrar una solicitud.

Por ejemplo, supongamos que nuestros métodos devuelven listas de clientes, de forma que, si deseamos la lista de todos los clientes podemos realizar una petición REST con PostMan, como hemos realizado en la unidad anterior:

```
GET /clientes
```

Ahora deseamos pedir la lista de clientes que se llamen Gabriel, entonces podríamos usar un parámetro de consulta:

```
GET /clientes?nombre=Gabriel
```

Esto nos devolverá (si nuestra API está bien diseñada) la lista de todos los usuarios cuyo nombre sea Gabriel.

Por otro lado, supongamos que deseamos obtener un cliente específico por su id, entonces usaríamos una variable URI:

```
GET /clientes/12345
```

Esta petición nos devolverá la información del cliente identificado por 12345.

También podría ser válido usar un parámetro de consulta, por ejemplo:

```
GET /clientes?id=12345
```

El uso de ambos tipos de anotaciones queda bastante claro en estos ejemplos, y un diseño consistente nos obligará a usar uno u otro o una combinación de ambos, como ya vimos.

La recomendación es que usemos la anotación `@RequestParam` para filtrar una consulta, y la anotación `@PathVariable` para identificar recursos o entidades.

Operación Add

La operación Add es una operación de dos pasos:

1. Mostrar un formulario de creación
2. Guardar la operación post del formulario.

Implementamos el primer paso, que permitirá mostrar el formulario para la creación de un nuevo cliente. Esto se implementa en `ClienteController.java` y solo vamos a mostrar aquí esa parte del código:

```
1 // Alta de clientes
2 @GetMapping("/clientes/add")
3 public ModelAndView add() {
4
5     log.info("ClienteController - add: Anyadimos un nuevo cliente");
6
7     ModelAndView mav = new ModelAndView("clienteform");
8     mav.addObject("clienteDTO", new ClienteDTO());
9
10    return mav;
11 }
```

La anotación `@GetMapping` asigna la URL `/clientes/add` a esta acción del controlador. Crearemos un nuevo objeto de tipo `ClienteDTO` y lo pasaremos al modelo añadiéndolo al mapa, y a continuación devolveremos el objeto `ModelAndView`.

Podemos ver que estamos devolviendo un objeto de la clase `ClienteDTO` vacío a la vista. Esto es más un truco para reutilizar el código de vista tanto para la operación de crear (`add`, que es la que estamos implementando ahora) como para la operación de actualizar (`update`, que la veremos más adelante). Al proporcionar un objeto `ClienteDTO` vacío, reducimos la probabilidad de errores de puntero nulo al representar la vista. Es posible proporcionar un objeto vacío al modelo o realizar muchas comprobaciones nulas en la vista. Ambos caminos son correctos, pero en este caso, hemos optado por el primero.

Nuestra vista para la operación `add` tendrá una operación `Post` en el de formulario. Necesitamos una acción de controlador para manejar esto. El código será el siguiente:

```

1 // Salvar clientes
2 @PostMapping("/clientes/save")
3 public ModelAndView save(@ModelAttribute("clienteDTO") ClienteDTO clienteDTO) {
4
5     log.info("ClienteController - save: Salvamos los datos del cliente:" +
6     clienteDTO.toString());
7
8     // Invocamos a la capa de servicios para que almacene los datos del cliente
9     clienteService.save(clienteDTO);
10
11    // Redireccionamos para volver a invocar el metodo que escucha /clientes
12    ModelAndView mav = new ModelAndView("redirect:/clientes");
13    return mav;
14 }
```

`@PostMapping` es una versión especializada de la anotación `@RequestMapping` que actúa como un atajo para `@RequestMapping (method = RequestMethod.POST)`. Los métodos anotados `@PostMapping` manejan las solicitudes HTTP POST que coinciden con una expresión URI determinada, que en este caso es `/clientes/save`.

Como ya hemos comentado con anterioridad, mediante `@ModelAttribute` podemos acceder al atributo denominado `clienteDTO` que se encuentra en el modelo. Una de las cosas interesantes de Spring MVC es que tomará los parámetros de su formulario y los vinculará automáticamente a un objeto de tipo `ClienteDTO`.

Por lo que el objeto, con los datos vinculados, se recibe en el método `save` del controlador, que lleva, en este caso, la anotación `@PostMapping` con la URL `/clientes/save`.

Ahora invocará a la capa de servicios de cliente para manejar la lógica de negocio y la persistencia. Esto es solo una fachada de cómo se van a almacenar los datos, y esto es debido a que el controlador no conoce como se realiza la persistencia ya que esto acoplaría el código y sería muy complicado de mantener (al controlador y a la capa de servicios no le importa si se usa JPA, JDBC, Servicio Web, etc...). De esta forma desacoplamos el controlador de la lógica y de la persistencia.

Es el momento de invocar a la capa de servicios y ordenar que se almacene el cliente. En este ejemplo que estamos poniendo, si os fijáis en la captura de código, podemos ver que se nos produce un error en la línea que invoca a la capa de servicios de cliente. Esto es debido a que no está creada, y lo haremos posteriormente a definir el controlador al completo.

```

66 // Salvar clientes
67 @PostMapping("/clientes/save")
68 public ModelAndView save(@ModelAttribute("clienteDTO") ClienteDTO clienteDTO) {
69
70     log.info("ClienteController - save: Salvamos los datos del cliente:" + clienteDTO.toString());
71
72     // Invocamos a la capa de servicios para que almacene los datos del cliente
73     clienteService.save(clienteDTO);
74
75     // Redireccionamos para volver a invocar el metodo que escucha /clientes
76     ModelAndView mav = new ModelAndView("redirect:/clientes");
77     return mav;
78 }
```

Este ejemplo solo muestra que todo el proceso de almacenar el cliente ha ido correctamente, pero no está realizando ninguna validación de los datos, de forma que si fallara alguna validación debería redirigir al formulario de nuevo indicando el error de validación. Esta parte de la implementación la omitimos.

Si recordáis la clase `ClienteDTO.java` contiene dentro un objeto de tipo `RecomendacionDTO.java`, y este objeto no lo hemos inicializado al crear el objeto `ClienteDTO` y pasarlo al formulario. Por esta razón vamos a definir el constructor vacío en la clase `ClienteDTO.java` inicializando el objeto `RecomendacionDTO`. La clase quedará 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
7 import lombok.Data;
8 import lombok.ToString;
9
10 @Data
11 public class ClienteDTO implements Serializable{
12
13     private static final long serialVersionUID = 1L;
14     private Long id;
15     private String nif;
16     private String nombre;
17     private String apellidos;
18     private String claveSeguridad;
19     private String email;
20     @ToString.Exclude
21     private RecomendacionDTO recomendacionDTO;
22
23     //Convierte una entidad a un objeto DTO
24     public static ClienteDTO convertToDTO(Cliente cliente) {
25         // Creamos el clienteDTO y asignamos los valores basicos
26         ClienteDTO clienteDTO = new ClienteDTO();
27         clienteDTO.setId(cliente.getId());
28         clienteDTO.setNif(cliente.getNif());
29         clienteDTO.setNombre(cliente.getNombre());
30         clienteDTO.setApellidos(cliente.getApellidos());
31         clienteDTO.setClaveSeguridad(cliente.getClaveSeguridad());
32         clienteDTO.setEmail(cliente.getEmail());
33         // Asignamos la recomendacionDTO pasandole el clienteDTO como parametro
34
35         clienteDTO.setRecomendacionDTO(RecomendacionDTO.convertToDTO(cliente.getRecomendacion()
36             (), clienteDTO));
37
38         // Retorna el DTO
39         return clienteDTO;
40     }
41
42     // Convierte un objeto DTO a una entidad
43     public static Cliente convertToEntity(ClienteDTO clienteDTO) {
44         // Creamos la entidad cliente y le asignamos los valores
45         Cliente cliente = new Cliente();
46         cliente.setId(clienteDTO.getId());
```

```

45     cliente.setNif(clienteDTO.getNif());
46     cliente.setNombre(clienteDTO.getNombre());
47     cliente.setApellidos(clienteDTO.getApellidos());
48     cliente.setClaveSeguridad(clienteDTO.getClaveSeguridad());
49     cliente.setEmail(clienteDTO.getEmail());
50     // Asignamos la recomendacion pasandole el cliente como parametro
51
52     cliente.setRecomendacion(RecomendacionDTO.convertToEntity(clienteDTO.getRecomendacion
53     DTO(), cliente));
54
55     // Retorna la entidad
56     return cliente;
57 }
58
59 // Constructor vacio
60 public ClienteDTO() {
61     super();
62     this.recomendacionDTO = new RecomendacionDTO();
63 }

```

Para finalizar el método contendrá un nuevo `ModelAndView` que nos devolverá a la URL `/clientes`, que mostrará la lista de clientes, pero en este caso no utilizaremos `/clientes` sino que utilizaremos `redirect:/clientes` que invocará al método que escucha en `/clientes`, volviendo a cargar la lista de clientes de la base de datos y tras ello procede a enviarla a la vista a través del modelo. Si pusieramos directamente solo `/clientes` no cargaría la lista de clientes, puesto que no se ha invitado a la capa de servicio que se traiga la lista de clientes.

Operación Update

Las actualizaciones son acciones contra entidades existentes. Las actualizaciones son similares a las acciones de creación, donde tenemos dos acciones que involucran al controlador. Con una creación, mostramos un formulario para un nuevo elemento, mientras que una actualización se completará con datos de un elemento existente. Si bien esto es muy similar a la acción de creación, normalmente querremos una acción de controlador separada para mostrar el formulario de edición con la finalidad de capturar datos para realizar una actualización de la entidad.

```

1 // Actualizar la informacion de un cliente
2 @GetMapping("/clientes/update/{idCliente}")
3 public ModelAndView update(@PathVariable("idCliente") Long idCliente) {
4
5     log.info("ClienteController - update: Modificamos el cliente: " + idCliente);
6
7     // Obtenemos el cliente y lo pasamos al modelo para ser actualizado
8     ClienteDTO clienteDTO = new ClienteDTO();
9     clienteDTO.setId(idCliente);
10    clienteDTO = clienteService.findById(clienteDTO);
11
12    ModelAndView mav = new ModelAndView("clienteform");

```

```

13     mav.addObject("clienteDTO", clienteDTO);
14
15     return mav;
16 }
```

La buena noticia es que, funcionalmente, actualizar un cliente es lo mismo que crear un cliente, por lo que reutilizaremos el método `save` que ya tenemos definido:

```

1 // Salvar clientes
2 @PostMapping("/clientes/save")
3 public ModelAndView save(@ModelAttribute("clienteDTO") ClienteDTO clienteDTO) {
4
5     log.info("ClienteController - save: Salvamos los datos del cliente:" +
6 clienteDTO.toString());
7
8     // Invocamos a la capa de servicios para que almacene los datos del cliente
9     clienteService.save(clienteDTO);
10
11    // Redireccionamos para volver a invocar el metodo que escucha /clientes
12    ModelAndView mav = new ModelAndView("redirect:/clientes");
13    return mav;
14 }
```

Podemos ver que, tras crear o actualizar un cliente lo redirigiremos a la lista de clientes.

Operación Delete

Hay formas diferentes de implementar la acción de eliminación. Una de los más sencillos es utilizar una URL con el ID para la acción de borrar un elemento. Esto luego se puede implementar en los formularios web como una URL simple para hacer clic. A continuación, mostraremos como implementar la acción del controlador de esta forma:

```

1 // Borrar un cliente
2 @GetMapping("/clientes/delete/{idCliente}")
3 public ModelAndView delete(@PathVariable("idCliente") Long idCliente) {
4
5     log.info("ClienteController - delete: Borramos el cliente:" + idCliente);
6
7     // Creamos un cliente y le asignamos el id. Este cliente es el que se va a borrar
8     ClienteDTO clienteDTO = new ClienteDTO();
9     clienteDTO.setId(idCliente);
10    clienteService.delete(clienteDTO);
11
12    // Redireccionamos para volver a invocar al metodo que escucha /clientes
13    ModelAndView mav = new ModelAndView("redirect:/clientes");
14
15    return mav;
16 }
```

Este método tomará el valor de identificación de la URI y lo pasará al método de eliminación del servicio del cliente. Tras ello redirigimos a la vista de clientes para mostrárselos al usuario.

Anotación @Autowired

Si os fijáis en los métodos desarrollados con anterioridad todos muestran un error, que es que no encuentra el objeto que debe ser el servicio de clientes. Es momento de definir como debe aparecer dicho objeto en el controlador y en el siguiente apartado definirlo.

Para ello utilizaremos la anotación `@Autowired` que permite llevar a cabo la inyección de dependencias sobre un atributo declarado en la clase.

Por ello vamos a injectar `clienteService` mediante la anotación `@Autowired` de la siguiente forma:

```
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;
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.service.ClienteService;
17
18 @Controller
19 public class ClienteController {
20
21     private static final Logger log =
LoggerFactory.getLogger(ClienteController.class);
22
23     @Autowired
24     private ClienteService clienteService;
25
26     // Listar los clientes
27     @GetMapping("/clientes")
28     public ModelAndView findAll() {
29
30         log.info("ClienteController - findAll: Mostramos todos los clientes");
31
32         ModelAndView mav = new ModelAndView("clientes");
33         List<ClienteDTO> listaClientesDTO = clienteService.findAll();
34         mav.addObject("listaClientesDTO", listaClientesDTO);
35 }
```

```

36     return mav;
37 }
38
39 // Visualizar la informacion de un cliente
40 @GetMapping("/clientes/{idCliente}")
41 public ModelAndView findById(@PathVariable("idCliente") Long idCliente) {
42
43     log.info("ClienteController - findById: Mostramos la informacion del cliente:" +
44 idCliente);
45
46     // Obtenemos el cliente y lo pasamos al modelo
47     ClienteDTO clienteDTO = new ClienteDTO();
48     clienteDTO.setId(idCliente);
49     clienteDTO = clienteService.findById(clienteDTO);
50
51     ModelAndView mav = new ModelAndView("clienteshow");
52     mav.addObject("clienteDTO", clienteDTO);
53
54     return mav;
55 }
56
57
58 // Alta de clientes
59 @GetMapping("/clientes/add")
60 public ModelAndView add() {
61
62     log.info("ClienteController - add: Anyadimos un nuevo cliente");
63
64     ModelAndView mav = new ModelAndView("clienteform");
65     mav.addObject("clienteDTO", new ClienteDTO());
66
67     return mav;
68 }
69
70 // Salvar clientes
71 @PostMapping("/clientes/save")
72 public ModelAndView save(@ModelAttribute("clienteDTO") ClienteDTO clienteDTO) {
73
74     log.info("ClienteController - save: Salvamos los datos del cliente:" +
75 clienteDTO.toString());
76
77     // Invocamos a la capa de servicios para que almacene los datos del cliente
78     clienteService.save(clienteDTO);
79
80     // Redireccionamos para volver a invocar el metodo que escucha /clientes
81     ModelAndView mav = new ModelAndView("redirect:/clientes");
82     return mav;
83 }
84
85 // Actualizar la informacion de un cliente
86 @GetMapping("/clientes/update/{idCliente}")

```

```

86     public ModelAndView update(@PathVariable("idCliente") Long idCliente) {
87
88         log.info("ClienteController - update: Modificamos el cliente: " + idCliente);
89
90         // Obtenemos el cliente y lo pasamos al modelo para ser actualizado
91         ClienteDTO clienteDTO = new ClienteDTO();
92         clienteDTO.setId(idCliente);
93         clienteDTO = clienteService.findById(clienteDTO);
94
95         ModelAndView mav = new ModelAndView("clienteform");
96         mav.addObject("clienteDTO", clienteDTO);
97
98         return mav;
99     }
100
101
102     // Borrar un cliente
103     @GetMapping("/clientes/delete/{idCliente}")
104     public ModelAndView delete(@PathVariable("idCliente") Long idCliente) {
105
106         log.info("ClienteController - delete: Borramos el cliente:" + idCliente);
107
108         // Creamos un cliente y le asignamos el id. Este cliente es el que se va a
109         // borrar
110         ClienteDTO clienteDTO = new ClienteDTO();
111         clienteDTO.setId(idCliente);
112         clienteService.delete(clienteDTO);
113
114         // Redireccionamos para volver a invocar al metodo que escucha /clientes
115         ModelAndView mav = new ModelAndView("redirect:/clientes");
116
117         return mav;
118     }
119 }
```

Ya solo nos queda comenzar a definir e implementar la capa de servicio.

Definir e implementar el Servicio

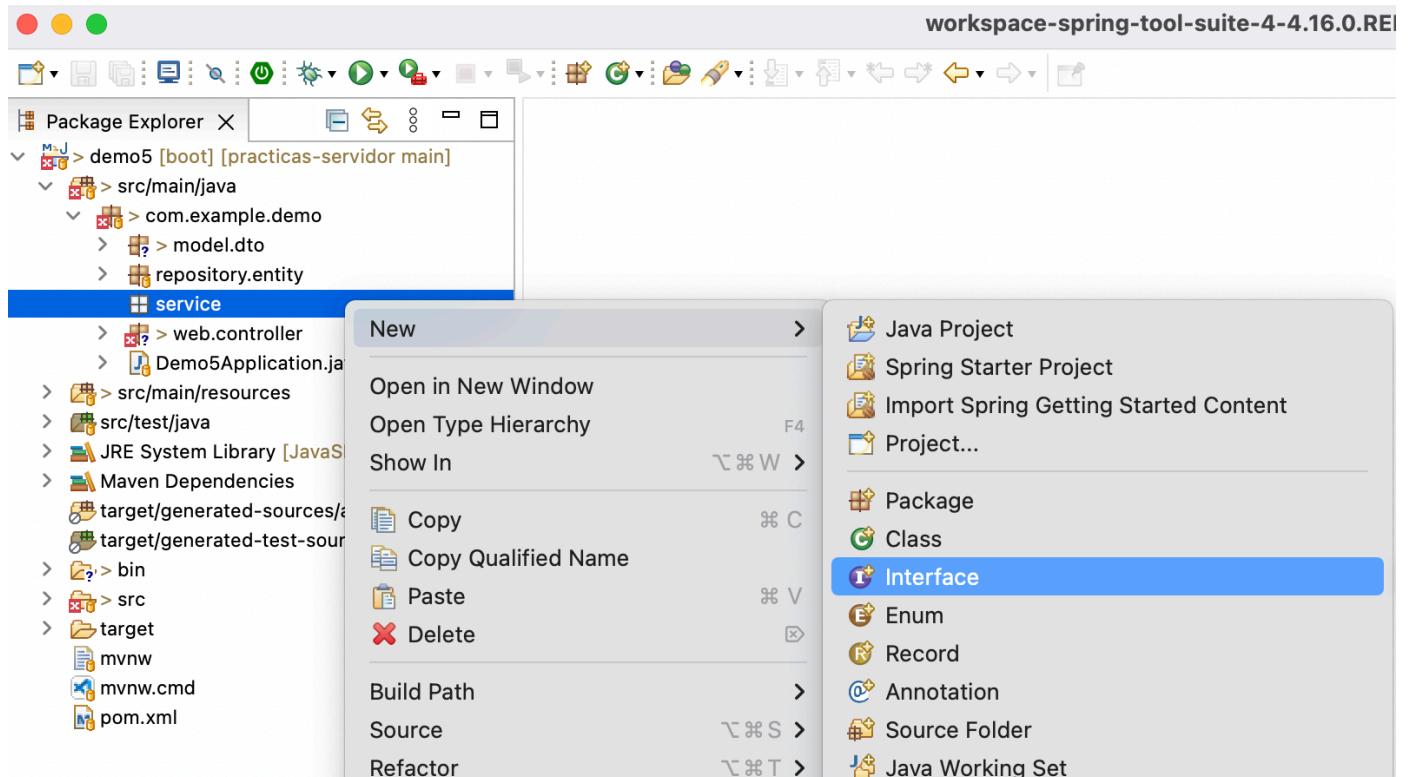
La capa de servicio gestiona la lógica de negocio de nuestra aplicación. Esta lógica de negocio está separada de la lógica web, que se encuentra en el controlador.

Cuando definamos clases que implementan servicios para la lógica de negocio se deben seguir las siguientes reglas:

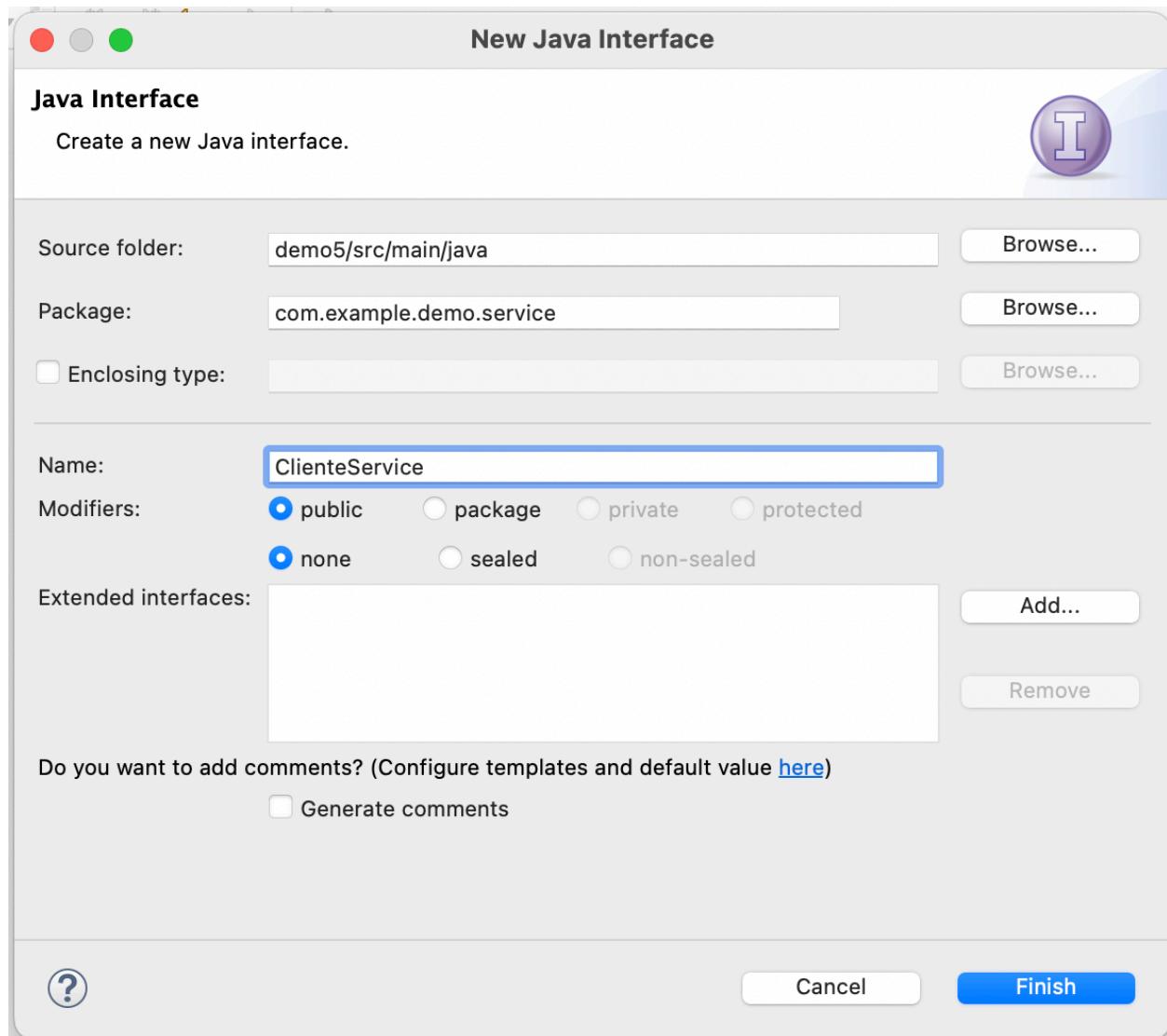
1. Definir una interfaz que tendrá las cabeceras de los métodos que se quieren publicar. De esta forma hacemos uso del patrón Facade y exponemos los métodos del servicio a usar.

2. Definimos una clase (tener en cuenta que una interfaz puede tener varias clases que la implementen) que implemente la interfaz, de forma que podremos implementar todos los métodos del servicio siguiendo la lógica de negocio requerida.
3. La anotación `@Service` indica a Spring que reconozca la clase como un servicio (es similar a la anotación `@Controller`).
4. Haremos uso de `@Autowired` para injectar el servicio en el controlador (esto ya lo hemos realizado en el controlador).
5. Haremos uso de `@Autowired` para injectar al servicio el DAO con el que vayamos a trabajar en la clase que implementa la interfaz de servicio.
6. Tener en cuenta que un método de servicio definirá una operación a nivel de negocio, por ejemplo, la lógica de gestión que conlleva dar de alta una factura. Los métodos de servicio estarán formados por otras operaciones más pequeñas, las cuales estarán definidas en la capa de repositorio.

Con todo esto, vamos a comenzar a definir la capa de servicio. Por ello comenzamos creando un interfaz denominado `ClienteService.java` en el paquete `service`. Sobre la capa de servicio pulsamos botón derecho del ratón para que aparezca el menú contextual y seleccionamos *New > Interface*.



Al nombre le ponemos `clienteService` y pulsamos Finish.



Declaramos las cabeceras de los métodos del servicio que hemos usado en el controlador. 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
7 public interface ClienteService {
8
9     List<ClienteDTO> findAll();
10    ClienteDTO findById(ClienteDTO clienteDTO);
11    void save(ClienteDTO clienteDTO);
12    void delete(ClienteDTO clienteDTO);
13 }
```

Una vez que tenemos el interfaz creamos una nueva clase denominada `ClienteServiceImpl.java`, que se encargará de implementar los métodos que se declaran en la interfaz. Tras ello hacemos que la nueva clase implemente la interfaz creada y tras ello hacemos que implemente por defecto todos los métodos de la interfaz. También indicamos la anotación `@Service` a la clase. Quedará de la siguiente forma:

```

1 package com.example.demo.service;
2
3 import java.util.List;
4 import org.springframework.stereotype.Service;
5 import com.example.demo.model.ClienteDTO;
6
7 @Service
8 public class ClienteServiceImpl implements ClienteService{
9
10    @Override
11    public List<ClienteDTO> findAll() {
12        // TODO Auto-generated method stub
13        return null;
14    }
15
16    @Override
17    public ClienteDTO findById(ClienteDTO clienteDTO) {
18        // TODO Auto-generated method stub
19        return null;
20    }
21
22    @Override
23    public void save(ClienteDTO clienteDTO) {
24        // TODO Auto-generated method stub
25
26    }
27
28    @Override
29    public void delete(ClienteDTO clienteDTO) {
30        // TODO Auto-generated method stub
31
32    }
33}

```

Si os fijáis, en este momento, si nos vamos a la clase `ClienteController.java` sí que podemos realizar el import de la clase `ClienteService`, con lo que los errores que teníamos en el controlador desaparecerán:

```

1 package com.example.demo.web.controller;
2
3 import org.slf4j.Logger;
4
5 import org.springframework.stereotype.Controller;
6
7 import com.example.demo.service.ClienteService;
8
9
10 @Controller
11 public class ClienteController {
12
13     private static final Logger log = LoggerFactory.getLogger(ClienteController.class);
14
15     @Autowired
16     private ClienteService clienteService;
17
18     @GetMapping("/clientes")
19     public ModelAndView findAll() {
20         log.info("findAll - findAll");
21         return new ModelAndView("listado-clientes", "list", clienteService.findAll());
22     }
23
24     @PostMapping("/clientes")
25     public String save(@Valid ClienteDTO clienteDTO) {
26         log.info("save - saveCliente: " + clienteDTO);
27         clienteService.save(clienteDTO);
28         return "redirect:/clientes";
29     }
30
31     @GetMapping("/cliente/{id}")
32     public ModelAndView findById(@PathVariable Long id) {
33         log.info("findById - findById(" + id);
34         ClienteDTO clienteDTO = clienteService.findById(id);
35         if (clienteDTO == null) {
36             log.error("No se encontró el cliente con id: " + id);
37             return new ModelAndView("error", "error", "No se encontró el cliente con id: " + id);
38         }
39         return new ModelAndView("ver-cliente", "cliente", clienteDTO);
40     }
41
42     @DeleteMapping("/cliente/{id}")
43     public String delete(@PathVariable Long id) {
44         log.info("delete - deleteCliente(" + id);
45         clienteService.delete(id);
46         return "redirect:/clientes";
47     }
48 }

```

Ahora ya inyecta el servicio en el controlador mediante la anotación `@Autowired`.

Ahora inyectamos el objeto que hace referencia al repositorio de datos, que hace uso del patrón DAO. Este objeto no está creado, pero volvemos a hacer lo mismo que hemos hecho con el controlador, es decir, inyectarlo y luego lo implementaremos. Quedará de la siguiente forma:

```

1 package com.example.demo.service;
2
3 import java.util.List;
4 import org.springframework.stereotype.Service;
5 import com.example.demo.model.dto.ClienteDTO;
6
7 @Service
8 public class ClienteServiceImpl implements ClienteService{
9
10     @Autowired
11     private ClienteRepository clienteRepository;
12
13     @Override
14     public List<ClienteDTO> findAll() {
15         // TODO Auto-generated method stub
16         return null;
17     }
18
19     @Override
20     public ClienteDTO findById(ClienteDTO clienteDTO) {
21         // TODO Auto-generated method stub
22         return null;
23     }
24 }

```

```

25    @Override
26    public void save(ClienteDTO clienteDTO) {
27        // TODO Auto-generated method stub
28    }
29
30
31    @Override
32    public void delete(ClienteDTO clienteDTO) {
33        // TODO Auto-generated method stub
34    }
35}
36

```

Tras esto nos queda implementar la lógica de negocio en la clase. En este caso, la lógica de negocio es muy sencilla, aunque puede ser más complicada, según los requerimientos. Eso sí, es importante tener en cuenta que la capa de servicio es la que se encarga de mapear objetos de DTO a entidades y viceversa, por lo que esta conversión se realizará en esta capa.

Destacamos el método `findAll()` que devolverá la lista de clientes. En este caso hemos utilizado funciones Lambda que va a convertir los objetos de tipo `Cliente` en objetos de tipo `ClienteDTO` de una forma más óptima y rápida (dejamos comentado el código que hace lo mismo con un bucle). 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.stream.Collectors;
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.repository.entity.Cliente;
14
15 @Service
16 public class ClienteServiceImpl implements ClienteService{
17
18     private static final Logger log =
19         LoggerFactory.getLogger(ClienteServiceImpl.class);
20
21     @Autowired
22     private ClienteRepository clienteRepository;
23
24     @Override
25     public List<ClienteDTO> findAll() {
26
27         log.info("ClienteServiceImpl - findAll: Lista de todos los cliente");
28
29     }
30
31 }
32
33
34
35
36

```

```

28     List<ClienteDTO> listaClientesDTO = new ArrayList<ClienteDTO>();
29     List<Cliente> listaClientes = clienteRepository.findAll();
30     for(int i=0; i < listaClientes.size();i++) {
31         Cliente cliente = listaClientes.get(i);
32         ClienteDTO clienteDTO = ClienteDTO.convertToDTO(cliente);
33         listaClientesDTO.add(clienteDTO);
34     }
35
36     return listaClientesDTO;
37 }
38
39 @Override
40 public ClienteDTO findById(ClienteDTO clienteDTO) {
41
42     log.info("ClienteServiceImpl - findById: Buscar cliente por id: " +
clienteDTO.getId());
43
44     Cliente cliente = new Cliente();
45     cliente.setId(clienteDTO.getId());
46
47     cliente = clienteRepository.findById(cliente);
48     if(cliente!=null) {
49         clienteDTO = ClienteDTO.convertToDTO(cliente);
50         return clienteDTO;
51     }else {
52         return null;
53     }
54 }
55
56 @Override
57 public void save(ClienteDTO clienteDTO) {
58     log.info("ClienteServiceImpl - save: Salvamos el cliente: " +
clienteDTO.toString());
59
60     Cliente cliente = ClienteDTO.convertToEntity(clienteDTO);
61     clienteRepository.save(cliente);
62
63 }
64
65 @Override
66 public void delete(ClienteDTO clienteDTO) {
67     log.info("ClienteServiceImpl - delete: Borramos el cliente: " +
clienteDTO.getId());
68
69     Cliente cliente = new Cliente();
70     cliente.setId(clienteDTO.getId());
71     clienteRepository.delete(cliente);
72 }
73 }
```

Nos seguirá dando error en la clase (mejor el interfaz) `clienteRepository` , puesto que todavía no hemos definido su interfaz y la implementación de la misma. Lo siguiente es la capa del repositorio.

Definir e implementar el Repositorio

La capa de repositorio se encarga de gestionar el acceso a los datos. Esto permite separar la lógica de negocio del acceso a datos, permitiendo así, por ejemplo, poder modificar la base de datos del sistema sin afectar a la lógica.

En esta capa debemos distinguir dos tipos de accesos:

- Acceso a datos del modelo de datos propio del sistema que se hará mediante acceso DAO (Data Access Object). Para este tipo de acceso, usaremos el framework JPA (Java Persistence API) mediante Spring Data.
- Acceso a sistemas externos mediante conectores (webservices, APIs, etc.)

En este ejemplo, como todavía no vamos a trabajar con ninguno de los dos accesos vamos a usar datos estáticos para comprender el ejemplo, permitiendo así que en el siguiente tema podamos integrar JPA.

Por ello vamos a crear un interfaz en el que tengamos las operaciones que vamos a exponer. Este interfaz será `ClienteRepository.java` y se creará en el paquete `dao` de `repository`.

```
1 package com.example.demo.repository.dao;
2
3 import java.util.List;
4
5 import com.example.demo.repository.entity.Cliente;
6
7 public interface ClienteRepository {
8
9     List<Cliente> findAll();
10    Cliente findById(Cliente cliente);
11    void save(Cliente cliente);
12    void delete(Cliente cliente);
13 }
```

Una vez que ya tenemos el interfaz definimos la implementación del repositorio, que en este caso se llama `ClienteRespositoryImpl.java`. Creamos la clase y hacemos que implemente la interfaz, implementando, por defecto, los métodos que expone la interfaz e indicamos con la anotación `@Repository` que este componente es un repositorio. Quedará de la siguiente forma:

```
1 package com.example.demo.repository.dao;
2
3 import java.util.List;
4 import org.slf4j.Logger;
5 import org.slf4j.LoggerFactory;
6 import org.springframework.stereotype.Repository;
7 import com.example.demo.repository.entity.Cliente;
8
9 @Repository
10 public class ClienteRepositoryImpl implements ClienteRepository{
```

```

12  private static final Logger log =
LoggerFactory.getLogger(ClienteRepositoryImpl.class);
13
14  @Override
15  public List<Cliente> findAll() {
16      // TODO Auto-generated method stub
17      return null;
18  }
19
20
21  @Override
22  public Cliente findById(Cliente cliente) {
23      // TODO Auto-generated method stub
24      return null;
25  }
26
27  @Override
28  public void save(Cliente cliente) {
29      // TODO Auto-generated method stub
30
31  }
32
33  @Override
34  public void delete(Cliente cliente) {
35      // TODO Auto-generated method stub
36
37  }
38 }
```

Como todavía no trabajamos con bases de datos ni otro origen de datos, vamos a crear una lista estática y la poblaremos con datos, de forma que podamos trabajar con ella. Para ello, definimos una lista estática dentro del repositorio de datos de clientes y añadimos datos:

```

1 package com.example.demo.repository.dao;
2
3 import java.util.List;
4 import org.slf4j.Logger;
5 import org.slf4j.LoggerFactory;
6 import org.springframework.stereotype.Repository;
7 import com.example.demo.repository.entity.Cliente;
8 import com.example.demo.repository.entity.Recomendacion;
9
10 @Repository
11 public class ClienteRepositoryImpl implements ClienteRepository{
12
13     private static final Logger log =
LoggerFactory.getLogger(ClienteRepositoryImpl.class);
14
15     private static List<Cliente> datos = new ArrayList<Cliente>();
16
17     static {
```

```

18 // Creamos 5 clientes
19 Cliente c1 = new Cliente();
20 c1.setId(Long.valueOf(datos.size() + 1));
21 c1.setNombre("Antonio");
22 c1.setApellidos("López");
23 c1.setNif("11111111A");
24 c1.setEmail("antonio.lopez@prueba.com");
25 c1.setClaveSeguridad("12345");
26
27 Recomendacion r1 = new Recomendacion();
28 r1.setId(Long.valueOf(datos.size() + 1));
29 r1.setObservaciones("No tiene ninguna");
30 r1.setCliente(c1);
31 c1.setRecomendacion(r1);
32 datos.add(c1);
33
34 Cliente c2 = new Cliente();
35 c2.setId(Long.valueOf(datos.size() + 1));
36 c2.setNombre("Belen");
37 c2.setApellidos("Cuenca");
38 c2.setNif("22222222B");
39 c2.setEmail("belen.cuenca@prueba.com");
40 c2.setClaveSeguridad("67890");
41
42 Recomendacion r2 = new Recomendacion();
43 r2.setId(Long.valueOf(datos.size() + 1));
44 r2.setObservaciones("Muy recomendada");
45 r2.setCliente(c2);
46 c2.setRecomendacion(r2);
47 datos.add(c2);
48
49 Cliente c3 = new Cliente();
50 c3.setId(Long.valueOf(datos.size() + 1));
51 c3.setNombre("Juan");
52 c3.setApellidos("Pérez");
53 c3.setNif("33333333C");
54 c3.setEmail("juan.perez@prueba.com");
55 c3.setClaveSeguridad("02468");
56
57 Recomendacion r3 = new Recomendacion();
58 r3.setId(Long.valueOf(datos.size() + 1));
59 r3.setObservaciones("Sin comentarios");
60 r3.setCliente(c3);
61 c3.setRecomendacion(r3);
62 datos.add(c3);
63
64 Cliente c4 = new Cliente();
65 c4.setId(Long.valueOf(datos.size() + 1));
66 c4.setNombre("María");
67 c4.setApellidos("Rodríguez");
68 c4.setNif("44444444D");
69 c4.setEmail("maria.rodriguez@prueba.com");

```

```

70     c4.setClaveSeguridad("13579");
71
72     Recomendacion r4 = new Recomendacion();
73     r4.setId(Long.valueOf(datos.size() + 1));
74     r4.setObservaciones("Le encanta The Mandalorian");
75     r4.setCliente(c4);
76     c4.setRecomendacion(r4);
77     datos.add(c4);
78
79     Cliente c5 = new Cliente();
80     c5.setId(Long.valueOf(datos.size() + 1));
81     c5.setNombre("Juan");
82     c5.setApellidos("Gómez-Jurado");
83     c5.setNif("55555555E");
84     c5.setEmail("juan.gomez-jurado@prueba.com");
85     c5.setClaveSeguridad("98765");
86
87     Recomendacion r5 = new Recomendacion();
88     r5.setId(Long.valueOf(datos.size() + 1));
89     r5.setObservaciones("Escritor famoso");
90     r5.setCliente(c5);
91     c5.setRecomendacion(r5);
92     datos.add(c5);
93 }
94
95 @Override
96 public List<Cliente> findAll() {
97     // TODO Auto-generated method stub
98     return null;
99 }
100
101
102 @Override
103 public Cliente findById(Cliente cliente) {
104     // TODO Auto-generated method stub
105     return null;
106 }
107
108 @Override
109 public void save(Cliente cliente) {
110     // TODO Auto-generated method stub
111
112 }
113
114 @Override
115 public void delete(Cliente cliente) {
116     // TODO Auto-generated method stub
117
118 }
119 }

```

Como trabajamos con una estructura de tipo lista, en este caso un `ArrayList`, vamos a usar métodos que todos conocemos. Por ello vamos a explicar como van a implementarse los 4 métodos:

1. `findAll()`: Devuelve todo el `ArrayList`.
2. `findById(Cliente cliente)`: Busca en el `ArrayList`, mediante el método `indexOf()` la posición del elemento que queremos localizar y devuelve el elemento en dicha posición. El método `indexOf()` hace uso del método `equals()` de la clase `Cliente.java`. Si recordáis el método viene implementado por defecto, por lo que tenemos que redefinir el método `equals()` y `hashCode()` para que use solo el id y no todos los elementos de la clase. Estas modificaciones las haremos cuando acabemos de implementar la clase `ClienteRepositoryImpl`.
3. `delete(Cliente cliente)`: Es muy similar al anterior, ya que busca la posición mediante `indexOf()` y después elimina el elemento de la lista mediante el método `remove()`. Devolverá el elemento eliminado.
4. `save(Cliente cliente)`: En este caso el método es invocado cuando se actualizan los datos o cuando se inserta uno nuevo. En caso de actualizar datos buscamos la posición del cliente en la lista mediante el método `indexOf()` y lo sustituimos mediante el método `set(posicion, elemento)`, mientras que al insertar uno nuevo lo que hacemos es darle valor a los id's, tanto del cliente como de la recomendación, y tras esto añadimos el cliente a la lista. Si la lista está vacía asignamos como id el valor 1.

El código será el siguiente:

```
1 package com.example.demo.repository.dao;  
2  
3 import java.util.ArrayList;  
4 import java.util.List;  
5  
6 import org.slf4j.Logger;  
7 import org.slf4j.LoggerFactory;  
8 import org.springframework.stereotype.Repository;  
9  
10 import com.example.demo.repository.entity.Cliente;  
11 import com.example.demo.repository.entity.Recomendacion;  
12  
13 @Repository  
14 public class ClienteRepositoryImpl implements ClienteRepository {  
15  
16     private static final Logger log =  
LoggerFactory.getLogger(ClienteRepositoryImpl.class);  
17  
18     private static List<Cliente> datos = new ArrayList<Cliente>();  
19  
20     static {  
21         // Creamos 5 clientes  
22         Cliente c1 = new Cliente();  
23         c1.setId(Long.valueOf(datos.size() + 1));  
24         c1.setNombre("Antonio");  
25         c1.setApellidos("López");  
26         c1.setNif("11111111A");
```

```

27     c1.setEmail("antonio.lopez@prueba.com");
28     c1.setClaveSeguridad("12345");
29
30     Recomendacion r1 = new Recomendacion();
31     r1.setId(Long.valueOf(datos.size() + 1));
32     r1.setObservaciones("No tiene ninguna");
33     r1.setCliente(c1);
34     c1.setRecomendacion(r1);
35     datos.add(c1);
36
37     Cliente c2 = new Cliente();
38     c2.setId(Long.valueOf(datos.size() + 1));
39     c2.setNombre("Belen");
40     c2.setApellidos("Cuenca");
41     c2.setNif("22222222B");
42     c2.setEmail("belen.cuenca@prueba.com");
43     c2.setClaveSeguridad("67890");
44
45     Recomendacion r2 = new Recomendacion();
46     r2.setId(Long.valueOf(datos.size() + 1));
47     r2.setObservaciones("Muy recomendada");
48     r2.setCliente(c2);
49     c2.setRecomendacion(r2);
50     datos.add(c2);
51
52     Cliente c3 = new Cliente();
53     c3.setId(Long.valueOf(datos.size() + 1));
54     c3.setNombre("Juan");
55     c3.setApellidos("Pérez");
56     c3.setNif("33333333C");
57     c3.setEmail("juan.perez@prueba.com");
58     c3.setClaveSeguridad("02468");
59
60     Recomendacion r3 = new Recomendacion();
61     r3.setId(Long.valueOf(datos.size() + 1));
62     r3.setObservaciones("Sin comentarios");
63     r3.setCliente(c3);
64     c3.setRecomendacion(r3);
65     datos.add(c3);
66
67     Cliente c4 = new Cliente();
68     c4.setId(Long.valueOf(datos.size() + 1));
69     c4.setNombre("María");
70     c4.setApellidos("Rodríguez");
71     c4.setNif("44444444D");
72     c4.setEmail("maria.rodriguez@prueba.com");
73     c4.setClaveSeguridad("13579");
74
75     Recomendacion r4 = new Recomendacion();
76     r4.setId(Long.valueOf(datos.size() + 1));
77     r4.setObservaciones("Le encanta The Mandalorian");
78     r4.setCliente(c4);

```

```

79     c4.setRecomendacion(r4);
80     datos.add(c4);
81
82     Cliente c5 = new Cliente();
83     c5.setId(Long.valueOf(datos.size() + 1));
84     c5.setNombre("Juan");
85     c5.setApellidos("Gómez-Jurado");
86     c5.setNif("55555555E");
87     c5.setEmail("juan.gomez-jurado@prueba.com");
88     c5.setClaveSeguridad("98765");
89
90     Recomendacion r5 = new Recomendacion();
91     r5.setId(Long.valueOf(datos.size() + 1));
92     r5.setObservaciones("Escritor famoso");
93     r5.setCliente(c5);
94     c5.setRecomendacion(r5);
95     datos.add(c5);
96 }
97
98 @Override
99 public List<Cliente> findAll() {
100    log.info("ClienteRepositoryImpl - findAll: Lista de todos los cliente");
101
102    return datos;
103 }
104
105 @Override
106 public Cliente findById(Cliente cliente) {
107    log.info("ClienteRepositoryImpl - findById: Buscar cliente por id: " +
cliente.getId());
108
109    int posicion = datos.indexOf(cliente);
110    log.info("ClienteRepositoryImpl - findById: Lo encuentra en la posicion " +
posicion);
111    return datos.get(posicion);
112 }
113
114 @Override
115 public void save(Cliente cliente) {
116    log.info("ClienteRepositoryImpl - save: Salvamos el cliente: " +
cliente.toString());
117
118    if (cliente.getId() != null) {
119        int posicion = datos.indexOf(cliente);
120        log.info("ClienteRepositoryImpl - save: Encuentra el cliente en la posicion: " +
+ posicion + " y procede a reemplazarlo.");
121        datos.set(posicion, cliente);
122    } else {
123        // Inicializamos el valor del id en caso de que no tenga datos
124        Long id = Long.valueOf(1);
125        // Comprobamos que tiene datos para asignar un nuevo id. Si no tiene se asigna

```

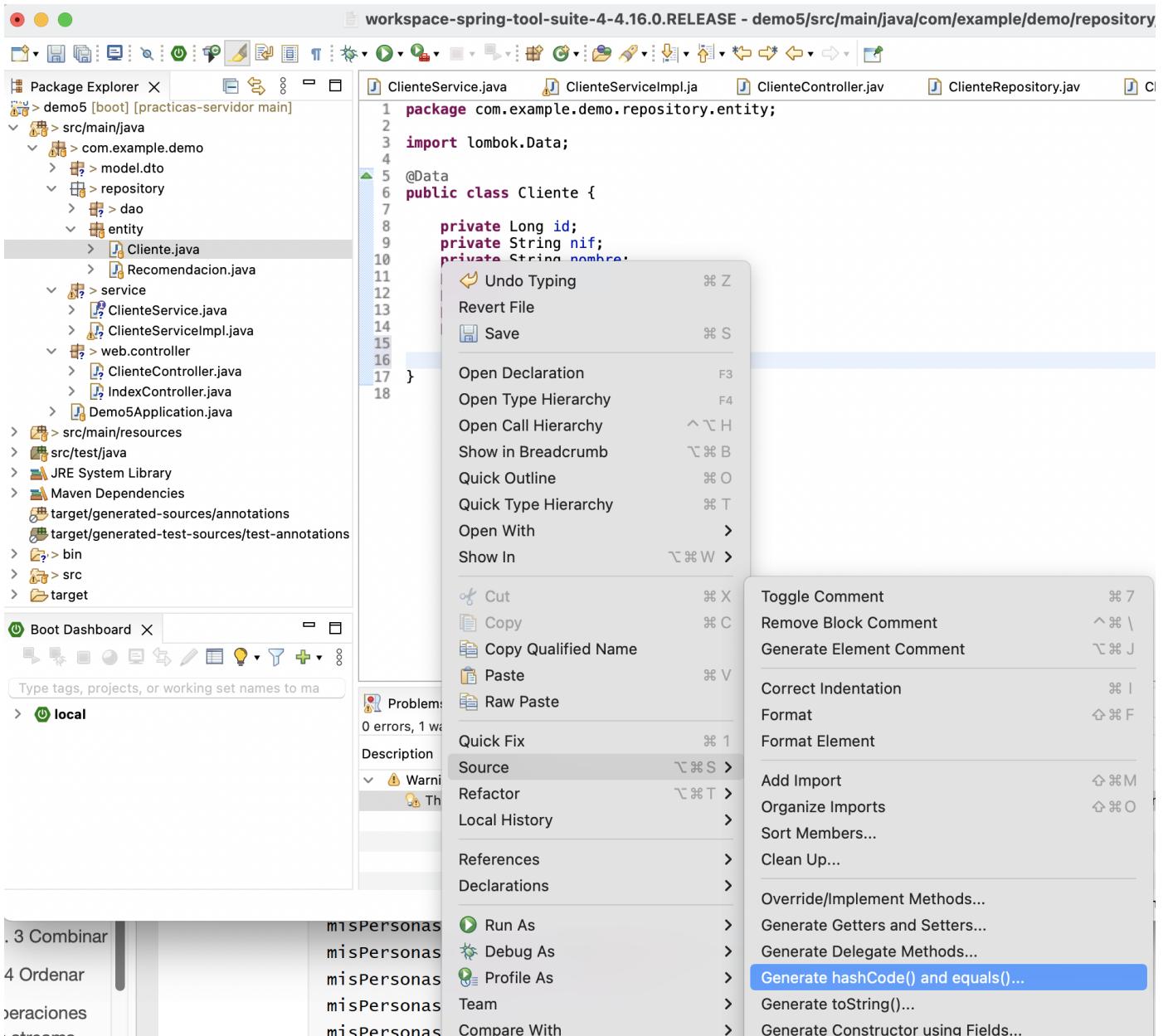
```

126     if (datos.size() > 0) {
127         id = datos.get(datos.size() - 1).getId() + 1;
128     }
129     cliente.setId(id);
130     cliente.getRecomendacion().setId(id);
131     log.info("ClienteRepositoryImpl - save: No encuentra el cliente y lo anyade a la lista.");
132     datos.add(cliente);
133 }
134 }
135
136 @Override
137 public void delete(Cliente cliente) {
138     log.info("ClienteRepositoryImpl - delete: Borramos el cliente: " +
cliente.getId());
139
140     int posicion = datos.indexOf(cliente);
141     log.info("ClienteRepositoryImpl - delete: Lo encuentra en la posicion: " +
posicion);
142     datos.remove(posicion);
143 }
144
145 }

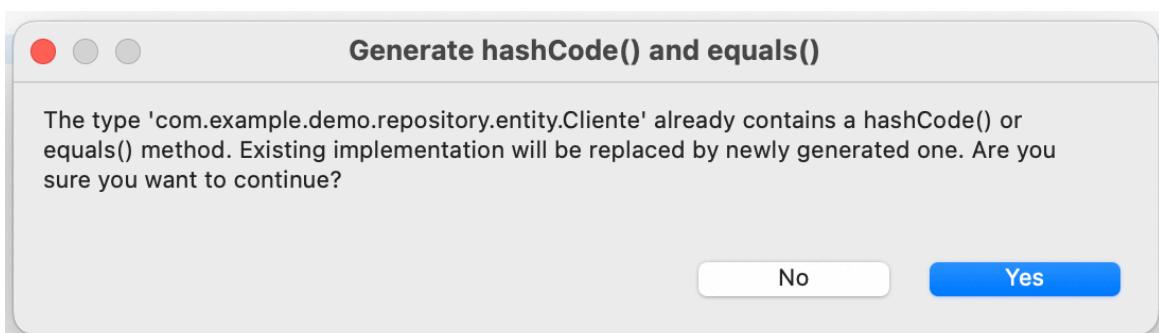
```

IMPORTANTE: Como ya tenemos implementada la clase solo nos falta importar la clase `ClienteRepository.java` en la clase `ClienteServiceImpl.java` para quitar el error que tenemos.

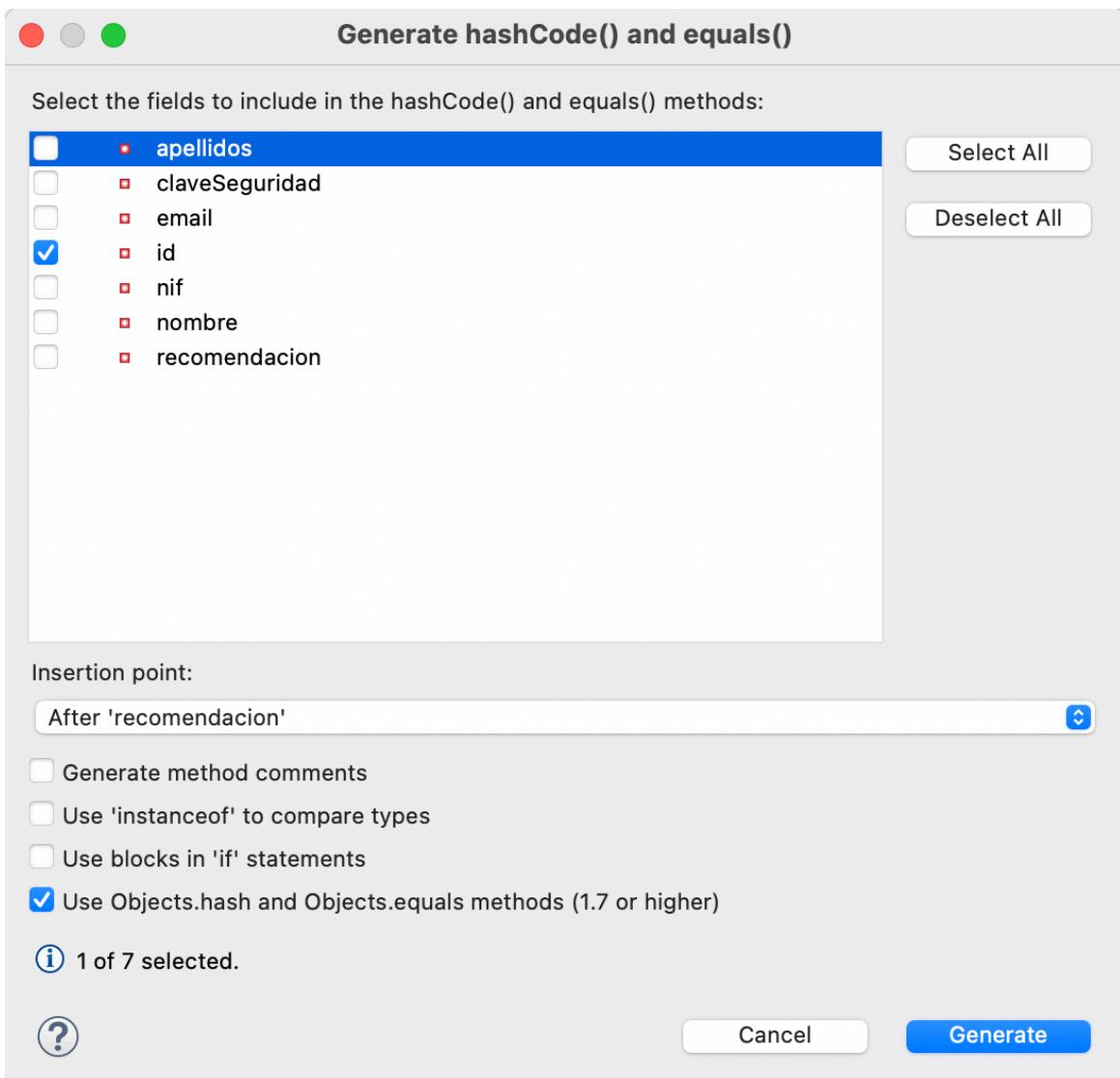
En cuanto a las modificaciones de la clase `cliente.java` haciendo referencia al método `equals()` y `hashTo()`, sobre la clase pulsaremos el botón derecho y seleccionaremos la opción *Source > Generate hashCode() and equals()*.



Nos saldrá el siguiente mensaje, ya que estos dos métodos se encuentran definidos, por defecto, con la anotación `@Data`.



Seleccionamos solamente el id de entre todos los atributos y pulsamos Generate:



El código generado en la clase `Cliente.java` será el siguiente:

```
1 package com.example.demo.repository.entity;
2
3 import java.util.Objects;
4
5 import lombok.Data;
6
7 @Data
8 public class Cliente {
9
10    private Long id;
11    private String nif;
12    private String nombre;
13    private String apellidos;
14    private String claveSeguridad;
15    private String email;
16    private Recomendacion recomendacion;
17
18    @Override
19    public boolean equals(Object obj) {
20        if (this == obj)
```

```

21     return true;
22     if (obj == null)
23         return false;
24     if (getClass() != obj.getClass())
25         return false;
26     Cliente other = (Cliente) obj;
27     return Objects.equals(id, other.id);
28 }
29
30 @Override
31 public int hashCode() {
32     return Objects.hash(id);
33 }
34 }
```

Esto hay que hacerlo en todas las clases que sean entidad, por lo que, en `Recomendacion.java` también hay que hacerlo, quedando de la siguiente forma:

```

1 package com.example.demo.repository.entity;
2
3 import java.util.Objects;
4
5 import lombok.Data;
6 import lombok.ToString;
7
8 @Data
9 public class Recomendacion {
10
11     private Long id;
12     private String observaciones;
13     @ToString.Exclude
14     private Cliente cliente;
15
16     @Override
17     public boolean equals(Object obj) {
18         if (this == obj)
19             return true;
20         if (obj == null)
21             return false;
22         if (getClass() != obj.getClass())
23             return false;
24         Recomendacion other = (Recomendacion) obj;
25         return Objects.equals(id, other.id);
26     }
27
28     @Override
29     public int hashCode() {
30         return Objects.hash(id);
31     }
32 }
```

Tras esto solo nos queda implementar las vistas.

Definir e implementar las Vistas

Ahora vamos a implementar la capa vista de la aplicación. Entendemos como vista aquellos elementos visuales de la aplicación, es decir, la interfaz gráfica de la web.

Recordamos que utilizamos Thymeleaf, que es un motor de plantillas que utilizaremos para renderizar las vistas. A partir de los elementos dados por el controlador, este potente sistema es capaz de devolver una vista.

Comenzaremos definiendo la vista principal de la aplicación (`index.html`), que será la página principal de entrada a la aplicación, que sirve el método `index` del controlador. Quedará de la siguiente forma (recordar crear la página dentro de `src/main/resources/templates`):

```
1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3 <head>
4   <meta charset="UTF-8">
5   <title th:text="${titulo}"></title>
6 </head>
7 <body>
8   <h1 th:utext="${titulo}"></h1>
9   <br>
10  <a th:href="@{/clientes}">Gestión de clientes</a>
11  <br>
12  <br>
13  <p th:utext="${nombreAsignatura}"></p>
14 </body>
15 </html>
```

Vamos a comentar algunas de las etiquetas de Thymeleaf que tenemos en `index.html`:

- `th:utext` :Se utiliza cuando hay un texto con formato. Si el texto a mostrar tuviera anotaciones html (por ejemplo `\<b\>` para especificar negrita) y quisieramos que estas anotaciones fueran interpretadas. La variable asignatura del fichero `application.properties` nos permitirá ver el funcionamiento de la misma. En este caso vamos a usar Expresiones variables, las cuales nos permiten:
 - Acceder a las propiedades de un objeto: `${sesión.usuario.nombre}`
 - Acceder a métodos definidos en nuestros propios objetos: `<p th:text="${myObject.myMethod()}" />`
- `th:text` :Se utiliza para mostrar el texto sin formato, o sea si lleva alguna etiqueta HTML mostrará la etiqueta tal y como está puesta en el texto, y no la interpretará.
- `th:href` :Sirve para construir URLs que podemos utilizar en cualquier tipo de contexto. En este caso usamos una expresión de enlace, que sirve para construir URLs que podemos utilizar en cualquier tipo de contexto, incluido para hacer enlaces para URLs que sean absolutas o relativas al propio contexto de la aplicación, al servidor, al documento, etc. Ejemplo:

```

1 <a th:href="@{/order/list}">...</a> // Esto es equivalente a ...
2 <a href="/myapp/order/list">...</a>

```

```

1 <a th:href="@{order/details(id=${orderId})}">...</a> // Esto es equivalente a ...
2 <a href="/myapp/order/details?id=23">...</a>

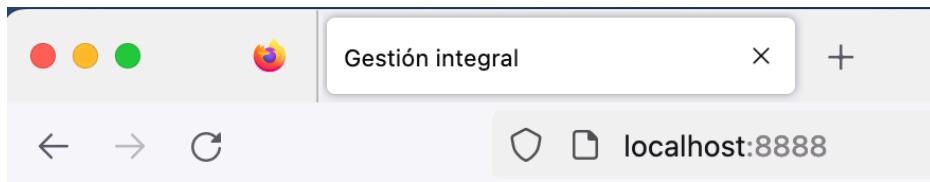
```

```

1 // Algunos ejemplos de uso
2 <a th:href="@{/clientes/update/{idCliente}(idCliente=${cliente.id})}">...</a>
3 <a th:href="@{.../documents/report}">...</a>
4 <a th:href="@{/contenidos/index}">...</a>
5 <a th:href="@{http://www.micom.es/index}">...</a>

```

Se visualizará de la siguiente forma:



Gestión integral

Gestión de clientes

Servidor

Ahora pasamos a definir la vista que muestra la lista de clientes, que será el fichero `clientes.html`, que tendremos que crear. El código será el siguiente:

```

1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3 <head>
4   <meta charset="UTF-8">
5   <title>Mantenimiento de clientes</title>
6 </head>
7 <body>
8   <div>
9     <br>
10    <h2 th:if="${listaClientesDTO.isEmpty()}">No hay clientes</h2>
11    <div th:if="${!listaClientesDTO.isEmpty()}">
12      <h2>Mantenimiento de clientes</h2>
13      <table border="1">
14        <thead>

```

```

15      <tr>
16          <td>id</td>
17          <td>Nombre</td>
18          <td>Apellidos</td>
19          <td>NIF</td>
20          <td>Correo electrónico</td>
21          <td>Editar</td>
22          <td>Eliminar</td>
23      </tr>
24  </thead>
25  <tbody>
26      <tr th:each="item : ${listaClientesDTO}">
27          <td><a
28              th:href="@{/clientes/{idCliente}(idCliente=${item.id})}"
29              th:text="${item.id}"></a></td>
30          <td th:text="${item.nombre}" />
31          <td th:text="${item.apellidos}" />
32          <td th:text="${item.nif}" />
33          <td th:text="${item.email}" />
34          <td><a
35              th:href="@{/clientes/update/{idCliente}(idCliente=${item.id})}"
36              th:text="Editar"></a></td>
37          <td><a
38              th:href="@{/clientes/delete/{idCliente}(idCliente=${item.id})}"
39              th:text="Eliminar"></a></td>
40      </tr>
41  </tbody>
42  </table>
43  </div>
44  <br>
45  <div>
46      <a th:href="@{/clientes/add}">Agregar</a>
47      <br>
48      <a th:href="@{}">Volver al inicio</a>
49  </div>
50 </body>
</html>

```

Las etiquetas a destacar en este fichero son las siguientes:

- `th:if` : Nos permite construir un condicional, de forma que si la lista está vacía muestra un mensaje de que no hay datos sino muestra los datos.

Podemos verlo con más profundidad en <https://o7planning.org/12351/thymeleaf-if-unless-switch>.

- `th:each` : Thymeleaf nos proporciona la posibilidad de implementar un bucle a través de esta etiqueta. Es la única forma de hacer un bucle con Thymeleaf. La sintaxis a utilizar es `th:each="item : ${items}"`, donde `item` hace referencia a un elemento de la lista que está en el modelo denominada `items`, a la cual accedemos mediante `${items}`.

Podemos verlo con más profundidad en <https://o7planning.org/12359/thymeleaf-loops>.

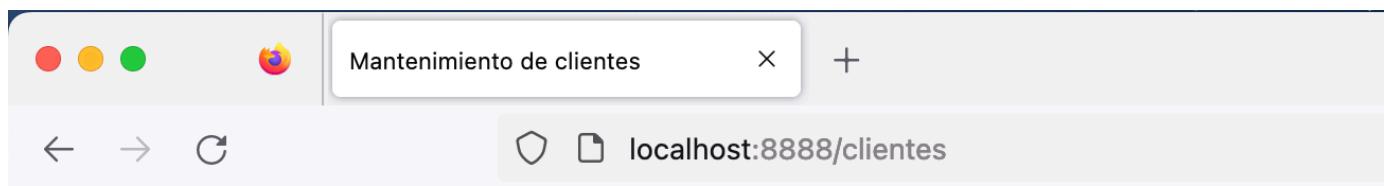
- `th:href="@{/clientes/{idCliente}(idCliente=${cliente.id})}"`: Se trata de una URI que tiene una variable en su PATH. En este caso la variable recibe el nombre de `idCliente`, y a continuación, entre paréntesis, podemos indicar el valor que va a recibir dicha variable.

Fijaros que, mediante la etiqueta `<tr>`, que representa una fila, vamos construyendo los datos de cada cliente en la rejilla. Mediante `th:utext` o `th:text` mostramos el valor de uno de los atributos del cliente.

Mediante `th:href` la usamos con la etiqueta `<a>` para crear un enlace a una URL que el controlador gestionará. Esto lo usaremos con el id para ver los datos del cliente y luego crearemos las opciones para editar el cliente o borrar el cliente.

Al final de la página crearemos dos enlaces a dos URLs: una para la operación de crear un nuevo cliente y otro para ir al índice.

Se visualizará de la siguiente forma:



Mantenimiento de clientes

id	Nombre	Apellidos	NIF	Correo electrónico	Editar	Eliminar
1	Antonio	Lopez	11111111A	antonio.lopez@prueba.com	Editar	Eliminar
2	Belen	Cuenca	22222222B	belen.cuenca@prueba.com	Editar	Eliminar
3	Juan	Pérez	33333333C	juan.perez@prueba.com	Editar	Eliminar
4	María	Rodríguez	44444444D	maria.rodriguez@prueba.com	Editar	Eliminar
5	Juan	Gómez-Jurado	55555555E	juan.gomez-jurado@prueba.com	Editar	Eliminar

[Nuevo cliente](#)

[Volver al inicio](#)

Vamos a comenzar por lo más sencillo, que es definir la vista que muestra la información de un cliente en concreto, que será el fichero `clienteshow.html`, que tendremos que crear. El código será el siguiente:

```

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>

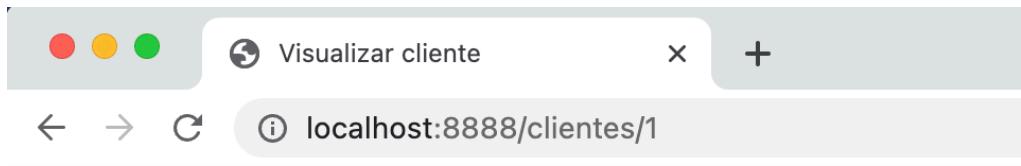
```

```

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>
40        </tr>
41        <tr>
42            <td>Recomendación</td>
43            <td>:</td>
44            <td th:text="${clienteDTO.recomendacionDTO.observaciones}"></td>
45        </tr>
46    </table>
47    <br>
48    <br>
49    <a th:href="@{/clientes}">Volver a la lista de clientes</a>
50 </body>
51 </html>

```

Se visualizará de la siguiente forma la visualizar los datos de un cliente:



Visualizar cliente

ID : 1
Nombre : Antonio
Apellidos : López
NIF : 11111111A
Correo electrónico : antonio.lopez@gmail.com
Clave seguridad : 12345
Recomendación : No tiene ninguna

[Volver a la lista de clientes](#)

Ahora pasamos a definir la vista para dar de alta o actualizar el cliente, que será el fichero `clienteform.html`, que tendremos que crear. El código será el siguiente:

```
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="${clienteDTO}"
12    method="POST">
13    <table border="0">
14      <tr th:if="!${add}">
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>Recomendación</td>
46      <td>:</td>
47      <td> <input type="text" th:field="*{recomendacionDTO.observaciones}" /> </td>
48  </tr>
49  </table>
50  <input type="submit" th:value="${add} ? 'Nuevo' : 'Actualizar'" />
51  </form>
52  <br>
53  <a th:href="@{/clientes}">Volver a la lista de clientes</a>
54  </body>
55  </html>

```

Las etiquetas a destacar en este fichero son las siguientes:

- Operador Elvis (a menudo descrito por `?`, es un operador binario (o es True o es False) que devuelve su primer operando si ese operando se evalúa como un valor verdadero y, de lo contrario, evalúa y devuelve su segundo operando): El operador Elvis es soportado por Thymeleaf. La sintaxis sería la siguiente:

```
<p th:utext="${myVariable} ? ${myValue1} : ${myValue2}"\></p>
```

Donde si `myVariable` tiene el valor true será equivalente a tener `<p th:utext="${myValue1}"\></p>` y en cambio, si tiene el valor false será equivalente a tener `<p th:utext="${myValue2}"\></p>`.

Tener en cuenta que, en Thymeleaf una variable es evaluada como false si tiene el valor: `null`, `false`, `0`, `"false"`, `"off"`, `"no"`.

- `th:action`: Se usa para indicar la acción que debe ejecutarse en el controlador, o sea la URL a la que enviaremos los datos del formulario al presionar el botón *Submit*. El formato es: `th:action="@/url"`
- `th:object`: Nos permite establecer a que atributo el modelo estará enlazado el formulario. Se utilizará con el siguiente formato: `th:object="${atributo}"`

- `th:field` : Nos permite enlazar una propiedad del objeto al que este vinculado el formulario a un elemento de este formulario, por ejemplo, enlazar un campo de texto a la propiedad nombre. Se utilizará el siguiente formato: `th:field="*{campo}"`. En este caso hablamos de Expresiones de selección, que son expresiones que nos permiten reducir la longitud de la expresión si prefijamos un objeto mediante una expresión variable, como por ejemplo `*{...}`.

Con la operación Elvis que hemos comentado la usamos para cambiar el título de la acción que vamos a realizar a partir de una variable denominada `add`, que accedemos a ella mediante `${add}`. Tendremos que modificar estas operaciones en el controlador y asignar el valor `true` a la variable `add` cuando se crea un cliente, y `false` cuando se actualiza un cliente. La modificación en `ClienteController.java` afectará a las operaciones `update` y `add`. Quedará de la siguiente forma:

```

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;
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.service.ClienteService;
17
18 @Controller
19 public class ClienteController {
20
21     private static final Logger log =
LoggerFactory.getLogger(ClienteController.class);
22
23     @Autowired
24     private ClienteService clienteService;
25
26     // Listar los clientes
27     @GetMapping("/clientes")
28     public ModelAndView findAll() {
29
30         log.info("ClienteController - findAll: Mostramos todos los clientes");
31
32         ModelAndView mav = new ModelAndView("clientes");
33         List<ClienteDTO> listaClientesDTO = clienteService.findAll();
34         mav.addObject("listaClientesDTO", listaClientesDTO);
35
36         return mav;
37     }
38 }
```

```

39
40     // Visualizar la informacion de un cliente
41     @GetMapping("/clientes/{idCliente}")
42     public ModelAndView findById(@PathVariable("idCliente") Long idCliente) {
43
44         log.info("ClienteController - findById: Mostramos la informacion del cliente:" +
45         idCliente);
46
47         // Obtenemos el cliente y lo pasamos al modelo
48         ClienteDTO clienteDTO = new ClienteDTO();
49         clienteDTO.setId(idCliente);
50         clienteDTO = clienteService.findById(clienteDTO);
51
52         ModelAndView mav = new ModelAndView("clienteshow");
53         mav.addObject("clienteDTO", clienteDTO);
54
55         return mav;
56     }
57
58     // Alta de clientes
59     @GetMapping("/clientes/add")
60     public ModelAndView add() {
61
62         log.info("ClienteController - add: Anyadimos un nuevo cliente");
63
64         ModelAndView mav = new ModelAndView("clienteform");
65         mav.addObject("clienteDTO", new ClienteDTO());
66         mav.addObject("add", true);
67
68         return mav;
69     }
70
71     // Salvar clientes
72     @PostMapping("/clientes/save")
73     public ModelAndView save(@ModelAttribute("clienteDTO") ClienteDTO clienteDTO) {
74
75         log.info("ClienteController - save: Salvamos los datos del cliente:" +
76         clienteDTO.toString());
77
78         // Invocamos a la capa de servicios para que almacene los datos del cliente
79         clienteService.save(clienteDTO);
80
81         // Redireccionamos para volver a invocar el metodo que escucha /clientes
82         ModelAndView mav = new ModelAndView("redirect:/clientes");
83         return mav;
84     }
85
86     // Actualizar la informacion de un cliente
87     @GetMapping("/clientes/update/{idCliente}")
88     public ModelAndView update(@PathVariable("idCliente") Long idCliente) {

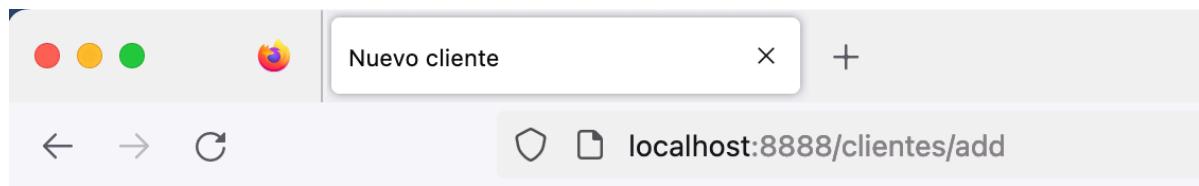
```

```

89 log.info("ClienteController - update: Modificamos el cliente: " + idCliente);
90
91 // Obtenemos el cliente y lo pasamos al modelo para ser actualizado
92 ClienteDTO clienteDTO = new ClienteDTO();
93 clienteDTO.setId(idCliente);
94 clienteDTO = clienteService.findById(clienteDTO);
95
96 ModelAndView mav = new ModelAndView("clienteform");
97 mav.addObject("clienteDTO", clienteDTO);
98 mav.addObject("add", false);
99
100 return mav;
101 }
102
103
104 // Borrar un cliente
105 @GetMapping("/clientes/delete/{idCliente}")
106 public ModelAndView delete(@PathVariable("idCliente") Long idCliente) {
107
108     log.info("ClienteController - delete: Borramos el cliente:" + idCliente);
109
110     // Creamos un cliente y le asignamos el id. Este cliente es el que se va a
111     // borrar
112     ClienteDTO clienteDTO = new ClienteDTO();
113     clienteDTO.setId(idCliente);
114     clienteService.delete(clienteDTO);
115
116     // Redireccionamos para volver a invocar al metodo que escucha /clientes
117     ModelAndView mav = new ModelAndView("redirect:/clientes");
118
119     return mav;
120 }
121 }
```

Fijaros que, cuando llegamos a la recomendación podemos acceder al objeto `RecomendacionDTO` y a su atributo `observaciones`.

Se visualizará de la siguiente forma la acción de dar de alta un nuevo cliente:

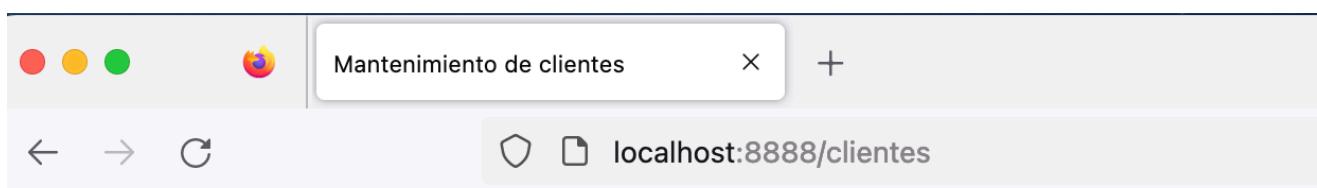


Nuevo cliente

Nombre	:	Violeta
Apellidos	:	Martínez
NIF	:	98765432A
Correo electrónico	:	violeta.martinez@prueba.com
Clave seguridad	:	12345
Recomendación	:	Excelente cliente.

[Nuevo](#)

[Volver a la lista de clientes](#)



Mantenimiento de clientes

id	Nombre	Apellidos	NIF	Correo electrónico	Editar	Eliminar
1	Antonio	Lopez	11111111A	antonio.lopez@prueba.com	Editar	Eliminar
2	Belen	Cuenca	22222222B	belen.cuenca@prueba.com	Editar	Eliminar
3	Juan	Checa	33333333C	juan.checha@prueba.com	Editar	Eliminar
4	María	Rodríguez	44444444D	maria.rodriguez@prueba.com	Editar	Eliminar
5	Juan	Gómez-Jurado	55555555E	juan.gomez-jurado@prueba.com	Editar	Eliminar
6	Violeta	Martínez	98765432A	violeta.martinez@prueba.com	Editar	Eliminar

[Nuevo cliente](#)

[Volver al inicio](#)

Como se puede apreciar el campo de recomendación es bastante pequeño, por lo que vamos a cambiar el componente para que sea un área de texto en vez de un input, ganando así una mejor presentación del mismo y más espacio para escribir. Lo cambiamos en el formulario `clienteform.html` quedando de la siguiente forma:

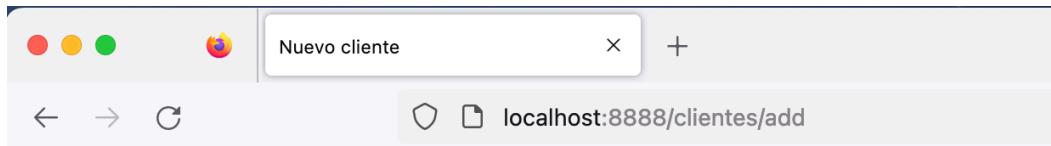
```
1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3 <head>
4 <meta charset="UTF-8">
5 <title th:text="${add} ? 'Nuevo cliente' : 'Actualizar cliente'"></title>
6 </head>
7 <body>
8   <h1 th:text="${add} ? 'Nuevo cliente' : 'Actualizar cliente'" />
9   <br>
10  <br>
11  <form th:action="@{/clientes/save}" 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>Recomendación</td>
46        <td>:</td>
```

```

47      <td> <textarea th:field="*{recomendacionDTO.observaciones}" rows="4"
48      cols="50" /> </td>
49    </tr>
50  </table>
51  <input type="submit" th:value="${add} ? 'Nuevo' : 'Actualizar'" />
52  <br>
53  <a th:href="@{/clientes}">Volver a la lista de clientes</a>
54 </body>
55 </html>

```

Se visualizará de la siguiente forma la acción de actualizar y añadir un cliente:



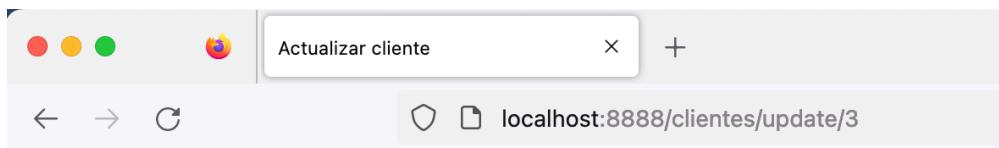
Nuevo cliente

Nombre	:	<input type="text" value="Violeta"/>
Apellidos	:	<input type="text" value="Martínez"/>
NIF	:	<input type="text" value="98765432A"/>
Correo electrónico	:	<input type="text" value="violeta.martinez@prueba.com"/>
Clave seguridad	:	<input type="text" value="12345"/>

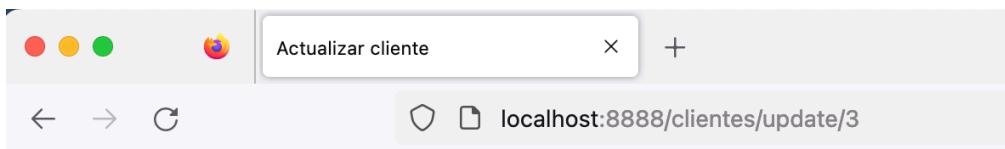
Recomendación :

Excelente cliente.

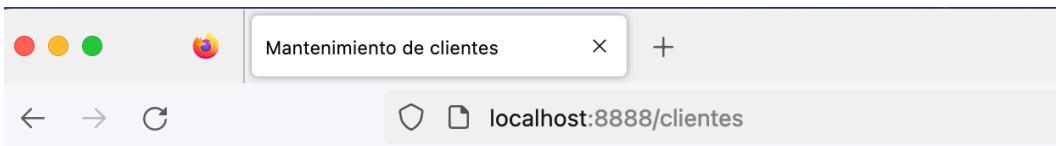
[Volver a la lista de clientes](#)



[Volver a la lista de clientes](#)



[Volver a la lista de clientes](#)



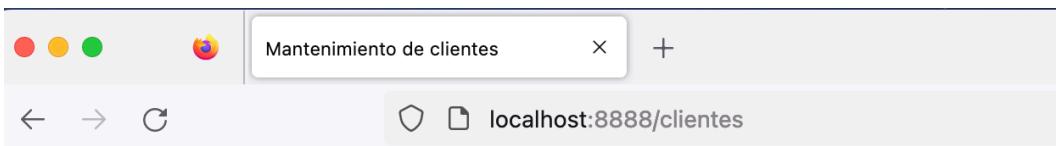
Mantenimiento de clientes

id	Nombre	Apellidos	NIF	Correo electrónico	Editar	Eliminar
1	Antonio	Lopez	11111111A	antonio.lopez@prueba.com	Editar	Eliminar
2	Belen	Cuenca	22222222B	belen.cuenca@prueba.com	Editar	Eliminar
3	Juan	Checa	33333333C	juan.checa@prueba.com	Editar	Eliminar
4	María	Rodríguez	44444444D	maria.rodriguez@prueba.com	Editar	Eliminar
5	Juan	Gómez-Jurado	55555555E	juan.gomez-jurado@prueba.com	Editar	Eliminar
6	Violeta	Martínez	98765432A	violeta.martinez@prueba.com	Editar	Eliminar

[Nuevo cliente](#)

[Volver al inicio](#)

La acción de eliminar un cliente, en esta implementación que hemos realizado, nos lleva a eliminar directamente el cliente y volver a cargar la lista de clientes. Mostramos un ejemplo antes de eliminar al cliente con id 3 y después de eliminarlo.

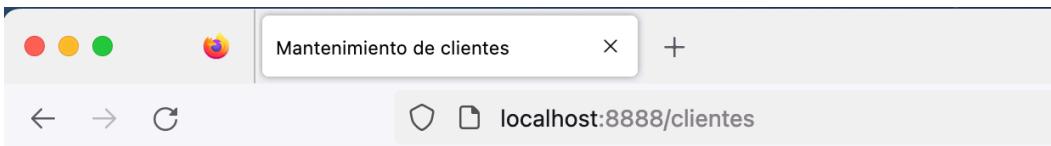


Mantenimiento de clientes

id	Nombre	Apellidos	NIF	Correo electrónico	Editar	Eliminar
1	Antonio	Lopez	11111111A	antonio.lopez@prueba.com	Editar	Eliminar
2	Belen	Cuenca	22222222B	belen.cuenca@prueba.com	Editar	Eliminar
3	Juan	Checa	33333333C	juan.checa@prueba.com	Editar	Eliminar
4	María	Rodríguez	44444444D	maria.rodriguez@prueba.com	Editar	Eliminar
5	Juan	Gómez-Jurado	55555555E	juan.gomez-jurado@prueba.com	Editar	Eliminar
6	Violeta	Martínez	98765432A	violeta.martinez@prueba.com	Editar	Eliminar

[Nuevo cliente](#)

[Volver al inicio](#)



Mantenimiento de clientes

id	Nombre	Apellidos	NIF	Correo electrónico	Editar	Eliminar
1	Antonio	Lopez	11111111A	antonio.lopez@prueba.com	Editar	Eliminar
2	Belen	Cuenca	22222222B	belen.cuenca@prueba.com	Editar	Eliminar
4	María	Rodríguez	44444444D	maria.rodriguez@prueba.com	Editar	Eliminar
5	Juan	Gómez-Jurado	55555555E	juan.gomez-jurado@prueba.com	Editar	Eliminar
6	Violeta	Martínez	98765432A	violeta.martinez@prueba.com	Editar	Eliminar

[Nuevo cliente](#)

[Volver al inicio](#)

Con esto habremos finalizado el diseño de la arquitectura de nuestra aplicación.

ANEXO I: ACLARACIONES ACERCA DE NUESTRA ARQUITECTURA

Vamos a puntualizar ciertos aspectos de nuestra arquitectura que poco a poco iremos refinando:

1. No hemos hecho ninguna validación sobre la entrada de datos de las vistas. Deberían haberse propuesto las validaciones necesarias y haberlas implementado.
2. Para buscar por id o para eliminar un cliente hemos pasado un id dentro de un ClienteDTO. Esto sigue el patrón MVC que habla en todo momento de pasar objetos y no valores directamente, por lo que tener en cuenta que hay que pasar objetos y no valores.
3. La anotación denominada `@RequestMapping` puede ir a nivel de método o a nivel de clase. Por ejemplo, si definimos a nivel de clase `@RequestMapping(value="/clientes")` las anotaciones de los métodos utilizarán el valor de la anotación (que es `/clientes`) de la clase como base o prefijo, y si anotamos un método con `@GetMapping(value="/add")` estaríamos mapeando la base más el nuevo valor del método (es decir `/clientes/add`).
4. Deberíamos haber creado una carpeta dentro de templates denominada `clientesview` y haber metido dentro de ella todas las vistas que hacen referencia a los clientes. Esto nos habría permitido haber estructurado mejor nuestro proyecto. A la hora de retornar las vistas, o sea en el return de los métodos, en vez de poner `return "/clientesform"` pondríamos `return "/clientesView/clientesform"`.

ANEXO II: TABLA DE CODIFICACIONES ENTRE ANSI Y UTF-8

IMPORTANTE: Esto solo es necesario realizarlo si la visualización en el navegador no es correcta, y salen caracteres extraños cuando queremos mostrar acentos, diéresis, etc.

La configuración por defecto de SpringBoot para el juego de caracteres de las aplicaciones es ISO-8859-1, pero nosotros necesitamos trabajar con UTF-8 para mostrar acentos, diéresis, letras ñ, etc. Por ello es necesario cambiar este tipo de caracteres por el código javascript acompañado por una \.

Por ello, en caso de tener errores en la visualización, modificaremos el fichero application.properties y modificamos el acento en el título "Gestión integral". En caso de ponerlo sería de la siguiente forma:

```
1 | aplicacion.nombre = Gestión integral
2 | asignatura = Servidor
3 | server.port = 8888
```

Dejamos parte de la tabla de codificaciones a realizar a continuación, así como el enlace:

ANSI	UTF-8	JAVASCRIPT
Á	Ã	u00c1
á	Ã¡	u00e1
É	Ã‰	u00c9
é	Ã©	u00e9
Í	Ã	u00cd
í	Ã¡	u00ed
Ó	Ã“	u00d3
ó	Ã³	u00f3
Ú	Ãš	u00da
ú	Ãº	u00fa
Ñ	Ã'	u00d1
ñ	Ã±	u00f1
¿	Ã¿	u00bf

<https://www.indalcasa.com/programacion/html/tabla-de-codificaciones-de-caracteres-entre-ansi-utf-8-javascript-html/>