Grado en Ingeniería Informática

Trabajo de Final de Grado

APLICACIÓN WEB PARA GESTIONAR EL CONTROL DE PRODUCCIÓN DE UNA EMPRESA CERÁMICA

Autor:
Iván Muñoz López

Supervisor:
Alejandro Sánchez-Camacho López
Tutor académico:
José Luis Llopis Borrás

Fecha de lectura: 16 de Septiembre de 2020
Curso académico 2019/2020

Resumen

En el presente documento se describe el análisis, planificación, diseño y la implementación del proyecto de creación de una aplicación web orientada al control de producción de un laboratorio de una empresa cerámica. La memoria presentada es parte del desarrollo del Trabajo de Fin de Grado del itinerario de Tecnologías de la Información y las Comunicaciones del Grado de Ingeniería Informática de la Universitat Jaume I.

El proyecto consistió en la renovación completa de una aplicación utilizada para el control de varios indicadores en el proceso de fabricación de material cerámico. Ya se contaba con una aplicación anteriormente para este propósito, pero debido a las tecnologías empleadas esta se estaba quedando obsoleta. Por ello se ha optado por Node.js o el *framework* Angular entre otros como nuevas tecnologías para su desarrollo.

Gracias a la aplicación implementada el trabajo del personal de laboratorio se verá simplificado, ya que aporta muchas facilidades y mejoras respecto a la aplicación anterior.

Palabras clave

Maven, Cliente-Servidor, Aplicación Web, Spring-Boot, Java, Angular, Node.js, MySQL.

Keywords

Maven, Client-Server, Web Application, Spring-Boot, Java, Angular, Node.js, MySQL.

Índice general

Índice de figuras

Índice de cuadros

Índice de listados

Capítulo 1

Introducción

1.1. Contexto y motivación del proyecto

El proyecto detallado en este documento se basa en el desarrollo de una aplicación web orientada al control de producción de una empresa cerámica. Este proyecto se realizó como resultado de las prácticas externas del Grado en Ingeniería Informática.

La principal motivación que dio lugar a este propósito fue la renovación de una aplicación ya existente que cumplía anteriormente con este cometido, pero debido a las tecnologías empleadas quedó obsoleta y se optó por una implementación completa de una nueva aplicación.

1.2. Objetivos del proyecto

El principal objetivo de este proyecto se centró en mejorar la usabilidad y simplificar la aplicación con respecto a la anterior, y así mismo reducir la complejidad de la misma y facilitar ciertas gestiones. Además de este objetivo principal podemos nombrar otros de menor importancia pero que también se han tenido en cuenta:

- Rehacer la aplicación con tecnologías actuales. Puesto que la anterior estaba desarrollada en Java base y AngularJS. La nueva, sin embargo, cuenta con Spring-Boot, Node.js y Angular.
- Modificar el aspecto visual de la nueva aplicación, consiguiendo de esta forma que se vea más actual, sencilla y usable:
    - Simplificar la configuración de nuevos controles de laboratorio. Mediante la utilización de menús desplegables podemos añadir varios elementos a un control sin la necesidad de insertar cada elemento uno a uno y con la posibilidad de hacerlo en varios pasos.
    - Simplificar el menú haciéndolo ocultable y desplegable para que el acceso a ciertas partes de la aplicación sea más rápido, intuitivo y menos invasivo.

    - Facilitar la comprensión de ciertos componentes haciéndolos genéricos de esta manera su uso se puede extender a diversas partes de la aplicación. En este caso los componentes hacen referencia a diversos elementos gráficos como: botones, desplegables y pestañas.
    - Utilizar iconos para diferenciar funciones y distintos apartados.
    - Emplear una distribución responsiva, es decir, que la aplicación se pueda visualizar correctamente tanto en un monitor como en una pantalla de una tableta u otro dispositivo móvil.

- Realizar modificaciones para simplificar parte de la funcionalidad: como por ejemplo la creación de un control completo, es decir, añadir ciertos indicadores agrupados bajo unas etiquetas.

    - Facilitar la creación de controles mediante el uso de plantillas que indica al sistema los indicadores, las medidas, las unidades y los valores de un control de producción.
    - Reducir la cantidad de clics para configurar un maestro, como puede ser un indicador.
    - Agrupar el menú de controles en una barra lateral.

1.3. Alcance

El alcance de este proyecto abarcó la implementación completa de una nueva aplicación basada en una anterior que quedó obsoleta. Para detallar más este aspecto del plan se ha dividido en tres apartados: organizativo, funcional e informático.

El alcance organizativo abarca el personal del laboratorio que realiza la supervisión de la producción de materiales cerámicos, puesto que son ellos los que la emplearán con el objetivo de realizar un seguimiento de ciertos valores.

El alcance funcional es muy parecido al de la aplicación anterior, sin embargo se ha simplificado parte de estas funcionalidades:

- Creación de controles con un fin determinado.
- Configuración con total libertad de medidas, unidades e indicadores.
- Acceso a controles creados y al configurador de nuevos controles.
- Introducción de valores en un control de producción.

Finalmente, el alcance informático es limitado puesto que basta con emplear cualquier tipo de navegador web, ya sea desde un ordenador de sobremesa, portátil, tableta o teléfono inteligente.

1.4. Descripción del proyecto

En esta sección se aporta información adicional sobre el proyecto y las tecnologías que se van a usar en el mismo, así como una descripción de la situación inicial de la que parte.

Este proyecto se ha basado en el desarrollo de una aplicación web encargada de gestionar y supervisar ciertos indicadores de la producción de azulejos de una empresa de este sector. Esta labor ha sido realizada hasta ahora por una aplicación con el mismo cometido que ha quedado desfasada. La parte más importante de esta idea es un control, es decir, una forma de agrupar ciertos indicadores en una estructura determinada. El esqueleto de un control es el siguiente:

- Áreas: Un control suele tener una o dos *áreas* que son una agrupación de ciertas *secciones*.
- Secciones: Una *sección* contiene *iteraciones*, aunque la gran mayoría solo tiene una, puede darse el caso de tener varias, por ejemplo múltiples temperaturas.
- Iteraciones: Una *iteración* contiene varios *indicadores*.
- Indicador: Un *indicador* es una entidad de control, como puede ser temperatura o presión, además contiene una o más *medidas*.
- Medidas: Una *medida* contiene el valor a controlar por los operadores de la aplicación además de una unidad asociada a dicho valor.

En cuanto a la navegación de la propia aplicación se ha decidió cambiar por completo su funcionamiento. Antes se podía acceder a un control desde una pestaña en la cabecera y se desplegaba cada apartado haciendo clic sobre él. Ahora se ha optado por una barra de navegación lateral con menús que permite acceder a toda la funcionalidad.

La herramienta de configuración se ha modificado de tal manera que se hace por partes, es decir, para crear un control desde cero antes se han de crear todos sus componentes: áreas, secciones, indicadores, medidas y unidades. O si se desea usar componentes que ya existen, simplemente se pueden agregarlos mediante un menú desplegable.

Además se ha incluido una nueva funcionalidad en la que a partir de una plantilla se puede obtener un control con todos sus indicadores correctamente ordenados y con todas sus medidas.

1.4.1. Tecnología empleada

Las tecnologías empleadas en este plan deben ser diferenciadas en dos grandes bloques, el primero de ellos es el *back-end*, el encargado de interconectar a la base de datos con el servido; y el *front-end*, cuyo cometido es de visualizar la información del lado cliente y permite a los usuarios interactuar con la aplicación web.

Back-end: la parte servidor se desarrolló en el lenguaje de programación Java[1], en concreto en su versión 13. Es un lenguaje orientado a objetos e independiente de la plataforma *hardware* donde se desarrolla. Además, se usó Spring-Boot[2] una herramienta que nació con la finalidad

de simplificar el desarrollo de aplicaciones. El sistema de gestión de base de datos utilizado es MySQL[5], es un sistema relacional de código abierto con un modelo cliente-servidor. También se usa Hibernate[6], es una herramienta de mapeo objeto-relacional que facilita el mapeo de atributos en una base de datos tradicional y el modelo de objetos de la aplicación, mediante archivos declarativos (XML) o anotaciones en los beans de las entidades que permiten establecer estas relaciones.

Front-end: la parte del cliente se desarrollo en el entorno de ejecución Node.js. En concreto se utilizó Angular versión 8.2.5, además se ha utilizado la biblioteca de estilos Material-Design.

Se han escogido estas tecnologías porque son las más habituales en la empresa durante el desarrollo de una aplicación web como esta.

1.5. Estructura de la memoria

La memoria se encuentra divida en cinco capítulos principales.

Para empezar, en el primer capítulo se introduce el proyecto, por ello se definen los siguientes aspectos: el contexto y la motivación del mismo, objetivos y subojetivos, el alcance funcional, organizativo e informático, la descripción del mismo y las tecnologías utilizadas en sus diferentes partes y por último esta misma estructura.

En el segundo capítulo encontramos la planificación del proyecto. Se encuentra divido en cuatro partes: la metodología utilizada, la planificación, los recursos y costes del proyecto estimados y el seguimiento del plan inicial que se ha llevado a cabo.

Más adelante, en el tercer capítulo, se analiza el sistema, su arquitectura y el diseño visual de su interfaz de usuario.

En el cuarto capítulo se explica con detalle toda la labor de implementación del sistema así como la verificación y validación del mismo.

Por último, en el quinto capítulo se explican las conclusiones desde diferentes puntos de vista.

Capítulo 2

Planificación del proyecto

2.1. Metodología

La metodología empleada en este proyecto es la que suele usar la empresa para todos sus trabajos, una metodología en cascada, como se puede ver en la figura 2.1. La forma de trabajo de esta metodología es secuencial y destaca por la definición de las fases del proyecto a principio del mismo. Aunque en este proyecto en concreto alguna de las fases sufrió algún tipo de modificación debido a decisiones de implementación que se tomaron sobre la marcha.

La empresa suele utilizar este tipo de metodología debido a su simplicidad y eficacia en proyectos de tamaño pequeño o mediano. Es muy útil para ir mejorando la aplicación una vez se ha conseguido llegar a las fases finales, puesto que permite volver a pasar por las primeras fases de nuevo para mejorar lo realizado anteriormente.

Las etapas que forman este tipo de metodología son cinco: análisis, diseño, implementación, pruebas y mantenimiento. Y ,como podemos ver en la figura 2.1, después de cada etapa se puede volver a cualquiera de las anteriores.

2.2. Planificación

La planificación del proyecto está compuesta por pasos que se siguen de forma secuencial. A continuación detallaré los pasos que se siguieron.

En el primer paso se analizaron todas las funcionalidades de la aplicación antigua que se debían incluir en esta. Además, desde el principio se tenía claro las modificaciones necesarias para mejorar parte de la funcionalidad antigua, así como parte del aspecto visual a mejorar.

En el segundo paso se desglosó la aplicación en partes, como pueden ser maestros, el configurar de un control de producción y los mapeadores para poder asignar diferentes apartados al personal de la empresa y que cada uno pudiera centrarse en el módulo que se le asignó. En este

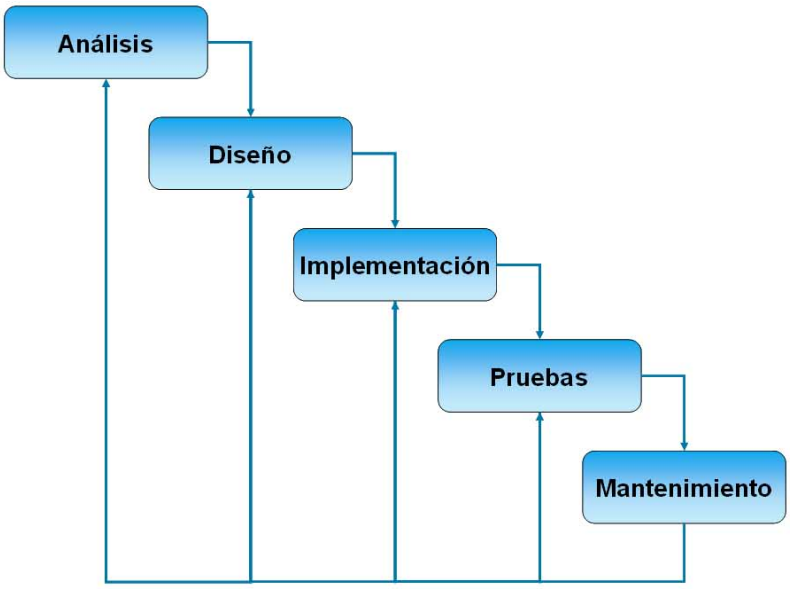

Figura 2.1: Metodología en cascada utilizada en el proyecto

caso en particular empezamos en el proyecto tres personas y posteriormente se unió una más. En las primeras semanas tuve que realizar labor en la parte del servidor mientras que los otros compañeros implementaban también la parte del cliente.

El tercer paso consistió en la implementación de todas las partes definidas en el diseño. Se comenzó por mi parte por el desarrollo del *back-end*. La configuración del proyecto Maven[7], una herramienta de software para la gestión y construcción de proyectos Java, se me otorgó desde un inicio. Este proyecto tenía el mismo esqueleto que el que usan en la empresa para la mayoría de sus proyectos.

En el cuarto paso se realizan las labores de estimación y valoración del progreso realizado hasta ese momento, para ello hay reuniones semanales en las que se corrobora la correcta funcionalidad implementada hasta ese momento.

La quinta fase en este caso al desarrollar solo un módulo del proyecto sin llegar este a estar acabado del todo no se ha podido lanzar la aplicación con toda su funcionalidad pero sí la parte desarrollada. Así pues se pudo probar el módulo principal del configurador de secciones y varios maestros.

Finalmente, en la fase de mantenimiento se han corregido varios aspectos que no funcionaban correctamente después de probar el módulo de secciones. Los maestros por otro lado funcionaron como se esperaba.

2.3. Estimación de recursos y costes del proyecto

Para realizar la estimación de recursos consideramos tres tipos: recursos humanos, materiales e indirectos.

- Recursos humanos: en este caso el proyecto ha sido realizado por cuatro personas, tres de ellas desde el inicio y una cuarta se incorporó al mes. Cada uno de los miembros ha estado realizando módulos independientes en la fase apropiada de su elaboración. Al principio del mismo todos realizamos labores del *back-end* y una vez estuvo el grueso completado de esta parte se empezó el desarrollo del *front-end* conjuntamente.

  EL autor de esta memoria realizó labores de desarrollo y diseño de software, tanto en la parte servidor como en la cliente. Aunque se realizó mucho más desarrollo en la parte del *back-end* que del *front-end*, puesto que tenía más experiencia y habilidad de implementación con Java y Spring-Boot[2] que con Typescript[8] y Angular[4].

- Recursos materiales: en esta sección hay que remarcar la inclusión del *software* junto con el *hardware* empleado.

  El *hardware* utilizado en concreto ha sido un ordenador de sobremesa Lenovo de la oficina de la empresa Integra Consultores, junto con periféricos de la misma marca.

  El *software* empleado en el desarrollo en la parte servidor ha sido el IDE IntelliJ IDEA. Por otro lado, el utilizado en la parte cliente ha sido el IDE Visual Studio Code. Para la visualización de la base de datos MySQL se ha utilizado el programa Navicat[9], es un administrador gráfico de base de datos que cuenta con un explorador como interfaz gráfica de usuario soportando múltiples conexiones para bases de datos locales y remotas. Y como herramienta de gestión de código se ha usado Atlassian Bitbucket[10], un servicio de alojamiento basado en web, para los proyectos que utilizan el sistema de control de versiones Mercurial y Git. Así mismo para gestionar las subidas y descargas de código se ha dispuesto del programa SourceTree[11], este programa simplifica la forma con la que se interactúa con los repositorios.

- Recursos indirectos: estos medios fueron los dispuestos para crear el entorno de trabajo en la empresa como son: el alquiler del local, acondicionamiento del mismo, recursos como muebles y herramientas y por último el consumo de agua y electricidad del local.

Continuando con los costes del proyecto se van a presentar unas tablas definiendo los costes del *software*, los recursos humanos, los recursos indirectos y la suma total de costes. Todos estos costes han sido calculados aplicando un porcentaje de uso o tiempo dedicado con ellos al coste real.

En la tabla 2.1 podemos observar los costes del *software* y del *hardware* empleado durante el desarrollo de la aplicación web.

En la tabla 2.2 podemos ver los costes humanos asociados al desarrollo de la aplicación web. Hay que destacar que el precio por hora es el habitual de la empresa para programadores junior como es el caso.

Recurso	Coste
IntelliJ IDEA	Licencia Gratuita
Visual Studio Code	Licencia Gratuita
SourceTree	Licencia Gratuita
Atlassian Bitbucket	5,00 €
Navicat	120,00 €
Ordenador	25,00 €
Monitor	10,00 €
Periféricos	3,00 €
Total	163,00 €

Cuadro 2.1: Costes del *software* y *hardware*

Recurso	Horas	Coste	Coste Total
Desarrollo, diseño y análisis	300	8,50€	2.550,00 €

Cuadro 2.2: Costes humanos

En la tabla 2.3 se puede ver la estimación de los costes indirectos en función del tiempo aprovechado en la oficina que fue todo el tiempo de desarrollo.

Recurso	Coste
Alquiler	150,00 €
Mobiliario	125,00 €
Consumo de agua	15,00 €
Consumo eléctrico	45,00 €
Limpieza	30,00 €
Total	365,00 €

Cuadro 2.3: Costes indirectos

Y por último, en la tabla 2.4 podemos ver la suma de todos los costes.

2.4. Seguimiento del proyecto

Durante las primeras semanas se realizaron múltiples reuniones cada semana. Esto propició que los detalles y requisitos del sistema quedarán claros antes de comenzar con el grueso de la implementación. Más adelante, según se iban acabando módulos o partes de la aplicación final, se realizaba una reunión para asegurarnos que se cumplían los requisitos de esa parte (ver la planificación de la figura 2.2). Aunque cabe remarcar que el diagrama sufrió alguna modificación de fechas debido a los detalles de implementación que fueron surgiendo durante el desarrollo del proyecto.

En la reuniones de las primeras semanas se me enseñó el esqueleto base del proyecto Maven[7]

Recurso	Coste Total
Software y *Hardware*	163,00 €
Humanos	2550,00 €
Indirectos	365,00 €
Total	3078,00 €

Cuadro 2.4: Costes totales

ACTIVIDAD	INICIO DEL PLAN	FIN DEL PLAN	DURACIÓN DEL PLAN	HORAS
			DIAS	
Proyecto laboratorio cerámico	17/02/2020	22/05/2020	60	300
Inicio del proyecto	17/02/2020	22/05/2020	60	300
Definición del proyecto	19/02/2020	24/02/2020	2	10
Reuniones	21/02/2020	24/02/2020	2	10
Establecer alcance y objetivos	22/02/2020	24/02/2020	2	10
Definición de tareas	21/02/2020	24/02/2020	2	10
Estimación de duraciones	22/02/2020	24/02/2020	2	10
Desarrollo del proyecto	24/02/2020	24/04/2020	36	180
Back-end (SpringBooT)	24/02/2020	09/04/2020	24	120
Modulo plantilla	24/02/2020	27/02/2020	4	20
Modulo control-indicador/areas	02/03/2020	06/03/2020	5	25
Modulo secciones	09/03/2020	24/03/2020	7	35
Inicio front-end con Angular 3	26/03/2020	24/04/2020	19	95
Implementación configurador	26/03/2020	09/04/2020	11	55
Maestro de secciones	14/04/2020	24/04/2020	9	45
Revisión	27/04/2020	06/05/2020	7	35
Validaciones	07/05/2020	11/05/2020	4	20
Test de la app web	11/05/2020	14/05/2020	4	20
Propuesta de mejora	15/05/2020	19/05/2020	3	15
Evalución de los resultados	19/05/2020	22/05/2020	4	20

Figura 2.2: Planificación temporal

con el que la empresa suele trabajar, así como la división de paquetes y su funcionalidad dentro del mismo. En las siguientes semanas se me fueron encargando desarrollar funcionalidades del *back-end*, una vez acabado era revisado por el supervisor. Hay que añadir que en general se mantuvo una reunión semanal para valorar el progreso del proyecto y decidir los siguientes pasos a tomar.

Debido a la situación de emergencia sanitaria provocada por el virus COVID-19 se optó por la modalidad de teletrabajo a partir del 23/03/2020. Justo antes de comenzar el teletrabajo se mantuvo una reunión en línea para acabar de configurar todo lo necesario para realizar el trabajo de forma telemática. Además coincidió este periodo con el comienzo por mi parte del desarrollo del *front-end* en Angular. Para ello de nuevo se mantuvo una reunión (a partir de este momento todas en línea) para exponer la estructura del proyecto de la parte cliente. A partir de aquí la forma de trabajo y revisión del mismo fue muy similar a la anterior, pues semanalmente se me informaban de mis tareas y una vez finalizadas se revisaban y se empezaban con otras.

Por último, durante la última semana se realizaron labores de revisión del trabajo hecho hasta ese momento así como la mejora y corrección de algunos detalles de implementación y desarrollo.

Capítulo 3

Análisis y diseño del sistema

3.1. Análisis del sistema

En este apartado se describen los requisitos funcionales del sistema así como los casos de uso.

Para una mejor comprensión de estos requisitos se realizará un división por módulos.

- Visor de controles: este módulo se emplea para el visionado de un control ya terminado, para ello es necesario disponer de una plantilla creada. En esta plantilla se encuentran todos los indicadores y parámetros necesarios a incluir en el control, para ello la parte servidor se encargar de, a partir de una referencia, realizar una copia de indicadores, medidas y unidades a un control. Una vez esto se ha realizado también se debe de ordenar de una determinada manera, siendo esta la siguiente: área, sección, iteración, indicador, medidas y unidad. Esta estructura se puede ver en la figura 3.1.
- Modificación de cabecera: cada control dispone de una cabecera situada antes de los indicadores en la parte superior de la página, como se puede ver en la parte superior de la figura 3.1, esta cabecera ha de ser siempre visible y se debe poder editar en cualquier momento.
- Configurador: es el apartado que dispone de mayor funcionalidad pues desde él se crean todos los componentes de un control (áreas, secciones, iteraciones, indicadores, medidas y unidades) y la creación de controles en sí mismos. También se puede ir añadiendo módulos a un control de producción para formarlo desde cero. Hay que añadir que es en todo momento modificable y se pueden eliminar o añadir nuevas partes. Para explicarlo mejor se ha divido el configurador en tres partes:
  - Maestros: los maestros son las piezas que forman un control, cada uno de ellos es una entidad propia en la base de datos, por ello cada maestro tiene la funcionalidad básica de crear, modificar o borrar.
  - Generación de plantillas: es la funcionalidad encargada de fabricar el esqueleto de un control, una vez se configuren todos sus módulos con partes ya existentes se puede

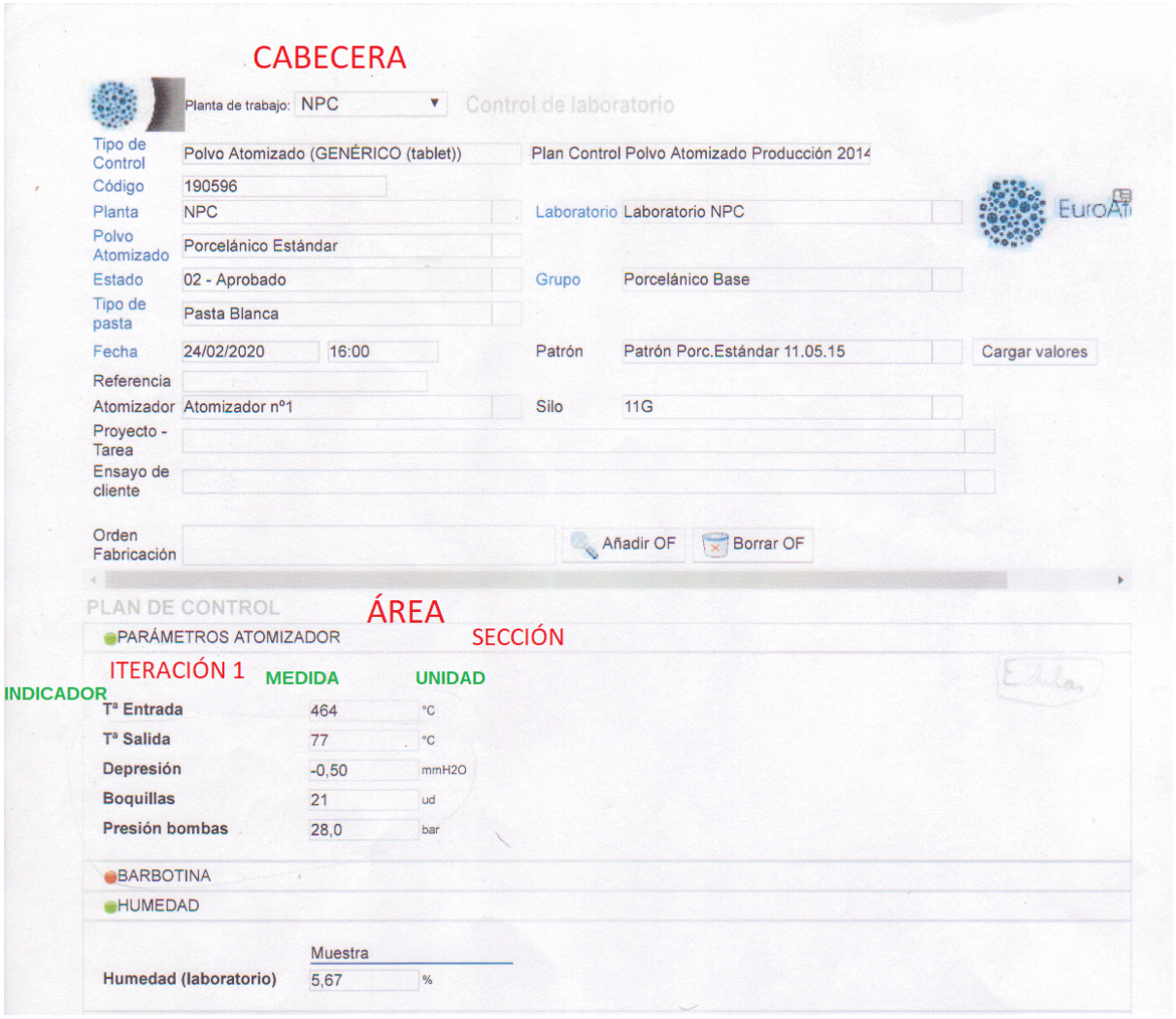

Figura 3.1: Estructura de un control

generar la vista del control para comenzar a realizar el seguimiento de los indicadores.

- Editor: como se ha comentado anteriormente todos los módulos o controles creados pueden ser modificados, así como su orden, pues ciertas partes, como son los indicadores, contienen medidas que según su orden se mostrarán de una determinada forma.

- Iteraciones: una iteración contiene varios indicadores.
- Indicador: un indicador es una entidad de control, como puede ser temperatura o presión, además contiene una o más medidas.
- Medidas: una medida contiene el valor a controlar por los operadores de la aplicación además de una unidad asociada a este valor.

En este último apartado se definen los casos de uso de la aplicación web así como los actores que la emplearan en cada ocasión. Hay que remarcar que las situaciones para emplear el sistema

están íntimamente relacionadas con la funcionalidad descrita en la sección **??**. Se ha optado por la tabla 3.1 para relacionar los casos de uso con los roles de usuario.

Casos de uso	Rol de usuario
Iniciar sesión	Todos los usuarios
Control de indicadores dentro de un rango	Personal de laboratorio
Creación de áreas, secciones, indicadores, medidas y unidades	Personal de laboratorio
Revisión de temperaturas	Operario de maquinaria de producción
Revisión de presión	Operario de maquinaria de producción
Establecer rangos de control	Personal de laboratorio
Creación de plantillas de control	Personal de laboratorio
Asignar roles	Administrador
Gestión de usuarios	Administrador

Cuadro 3.1: Casos de uso

3.2. Diseño de la arquitectura del sistema

En este apartado se va a describir el diseño de la arquitectura del sistema. El esqueleto es el habitual en una aplicación cliente-servidor como esta. Existen dos módulos principales, el primero de ellos el servidor se encarga de conectar con la base de datos y con el otro módulo, éste es el encargado de dibujar los datos de la aplicación e interaccionar con el usuario.

Este tipo de sistemas tiene una arquitectura Modelo Vista Controlador[12], como se puede ver en la figura 3.2. Éste es un estilo de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos.

El modelo contiene una representación de los datos que maneja el sistema, su lógica de negocio, y sus mecanismos de persistencia.

La vista, o interfaz de usuario, que compone la información que se envía al cliente y los mecanismos de interacción con éste.

El controlador, actúa como intermediario entre el modelo y la vista, gestionando el flujo de información entre ellos y las transformaciones para adaptar los datos a las necesidades de cada uno.

Visionando el patrón Modelo Vista Controlador desde la perspectiva de la empresa hay que destacar que para todos los proyectos se utilizan dos módulos.

El primero de ellos representa la parte servidor. Está desarrollado en Java 13 además del uso del *framework* Spring-Boot. Se encarga de la conexión con la base de datos mediante objetos JSON[15], este tipo de formato es de lectura y escritura simple para los humanos mientras que para las máquinas es simple interpretarlo y generarlo. Cada representación o tabla del almacén de datos tiene un retrato muy similar en este módulo. Estas representaciones se llaman maestros. Su estructura es la siguiente:

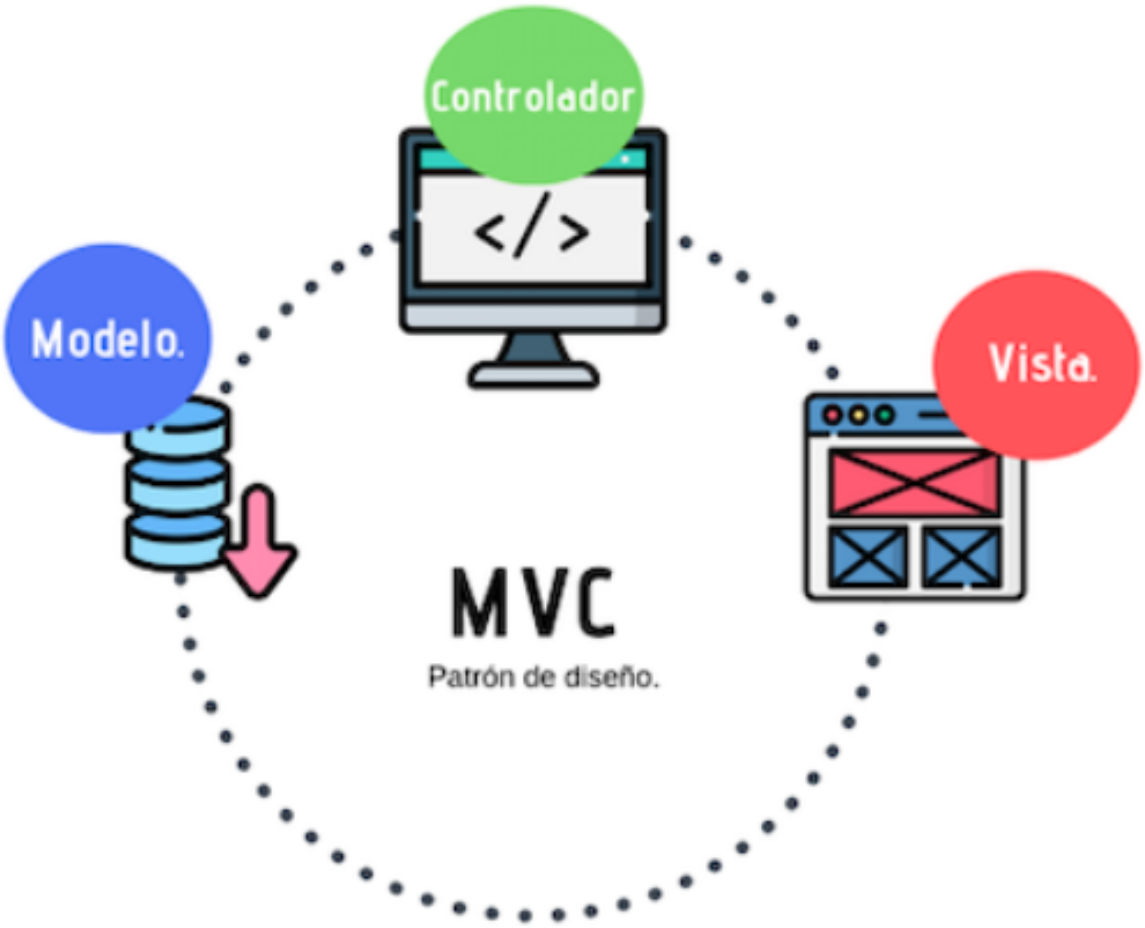

Figura 3.2: Estructura Modelo Vista Controlador

- Dominios: directorio de los datos, suele contener varios ficheros para el manejo de los mismos, su intercambio y su recepción desde el *front-end*
  - Entidades: son una copia de las tablas de la base de datos en la aplicación. Se utilizan para la gestión de los datos.
  - Objectos de acceso a los datos: son interfaces que se usan para relacionarse con la base de datos. Su función principal es obtener y almacenar los datos.
  - Objetos de transporte de datos: se utilizan para el intercambio de información con la parte cliente.
- Managers: en esta parte se encuentran todas las operaciones que se realizan a los datos. Normalmente se manipulan los objetos entidad y se crean a partir de ellos los objetos de transporte de datos para su posterior envío.
- Mapeadores: se encargan de transformar un objeto entidad en un objeto de transporte.

El segundo de ellos representa la capa de representación de los datos, aunque también se

utiliza en ciertas partes como capa de control. Está desarrollado en el lenguaje de programación Typescript (subconjunto de Javascript) integrado con Node.js[13], que es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor basado en el lenguaje de programación JavaScript y asíncrono. Además se ha utilizado el *framework* Angular, en su versión 8.

El proyecto desarrollado en Angular tiene un esqueleto muy diverso y amplio separando cada vista en componentes muy reducidos. La estructura del proyecto sería la siguiente:

- Componentes: pueden ser genéricos, es decir, están disponibles para todos los proyectos y se suelen utilizar en la mayoría de ellos. O bien pueden estar creados a propósito para un proyecto en concreto. En este caso los componentes son partes de una vista que se extrae de ella para simplificar su funcionamiento.
- Módulos: es la representación de los datos en el *front-end*, aquí se encuentra la información de cada maestro. Además realizan las labores de mapeado con los datos que recibe del *back-end*.
- Servicios: sirven como flujo de la información entre la parte cliente y la servidor. En este apartado se localizan las llamadas al controlador del *back-end* y se recogen sus resultados.
- Vistas: se utilizan para mostrar la información deseada de la manera apropiada. Este apartado suele contener un fichero HTML, un archivo Typescript y opcionalmente un fichero de estilo CSS.
- Formularios: en este apartado se encuentran los datos requeridos y su definición para ser recolectados.

3.2.1. Diseño de la base de datos

En este apartado se va a describir la base de datos utilizada durante este proyecto. Se han omitido algunas tablas de control de acceso y edición que no están relacionadas con la naturaleza del proyecto, además de otras tablas referentes a otros módulos en los que no tuve participación en ninguna parte de su proceso de implementación.

Como se puede ver en la figura 3.3 existen nueve tablas y se pueden dividir en dos grandes grupos:

- Maestros: en este apartado entran las tablas *medidas*, *unidades*, *indicadores*, *secciones* y *áreas*. Cada una de ellas representa unos datos que unidos formarán un *control*. Contienen información para poder organizar los indicadores dentro de una estructura determinada, así como mostrar y recoger valores. La tabla *medidas* se relaciona con las demás en la lógica de la aplicación, tomando una referencia guardada en formato JSON de la tabla *controles indicadores*.
- Configuradores: las tablas *plantillas*, *plantillas indicadores*, *controles* y *controles indicadores* son las encargadas de agrupar los maestros dentro de una determinada estructura.

Por lo tanto la información que contiene son identificadores a la mayoría de las tablas de maestros.

3.3. Diseño de la interfaz

En esta sección se van a exponer los criterios de diseño utilizados en la elaboración de la interfaz gráfica de la aplicación web. Para hacerlo de manera más sencilla se incluirán capturas de la propia aplicación para una mejor comprensión de los detalles. Aunque estas capturas se harán solo de los módulos que realizó el autor de esta memoria.

Hay que destacar que no se realizaron bocetos previos, pues el aspecto visual desde un principio se analizó y decidió que iba a ser parecido al de otro proyecto de la empresa, por lo tanto se cogió esa base y sobre ella se realizó todo el trabajo de implementación visual.

En la figura 3.4 podemos ver el menú desplegable en el cual se puede observar el acceso a los maestros, al configurador de controles y a los propios controles.

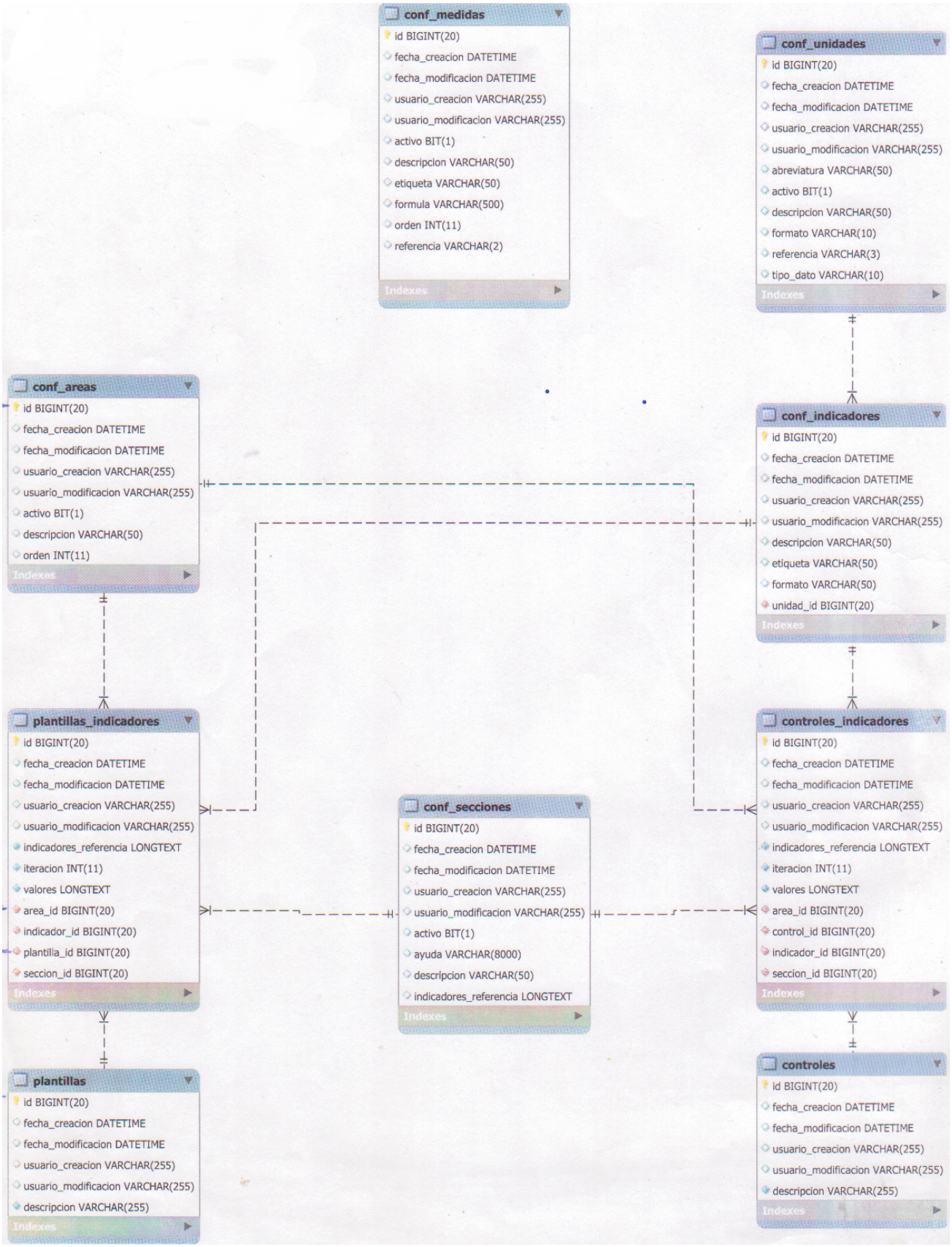

Figura 3.3: Base de datos MySQL

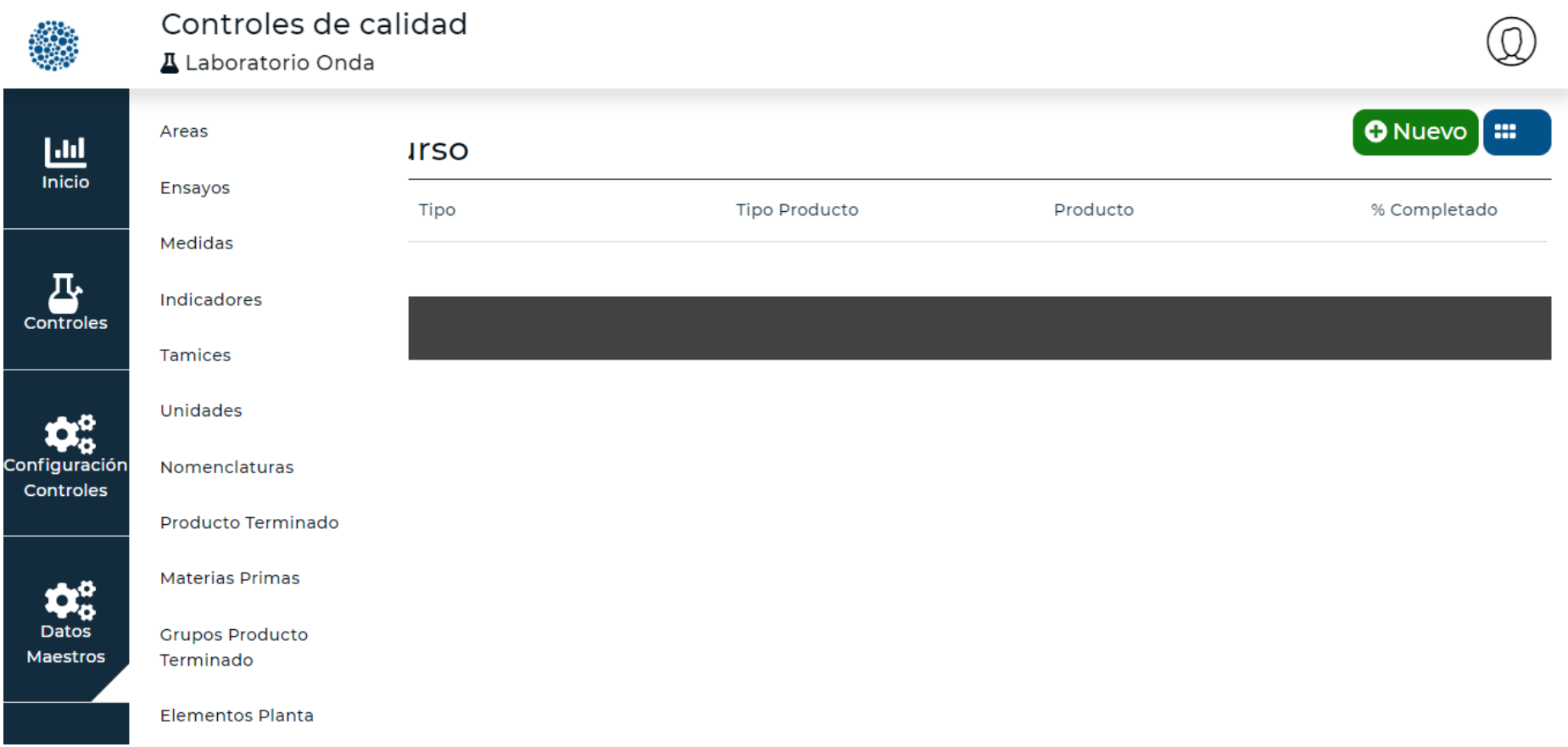

Figura 3.4: Menú lateral de la aplicación web.

Poniendo el foco en los apartados desarrollado por el autor de esta memoria podemos ver en la figura 3.5 el maestro de medidas. Como se puede ver hay un *grid* o tabla paginada en la que se nos muestran todas las medidas existentes. Esta tabla es un componente genérico que se suele usar un muchos proyectos de la empresa. Como también lo es el botón de *medida nueva* que se puede observar. Si queremos realizar modificaciones sobre una medida simplemente hay que hacer clic sobre ella para ver la ventana detalle, como se puede observar en la figura 3.6.

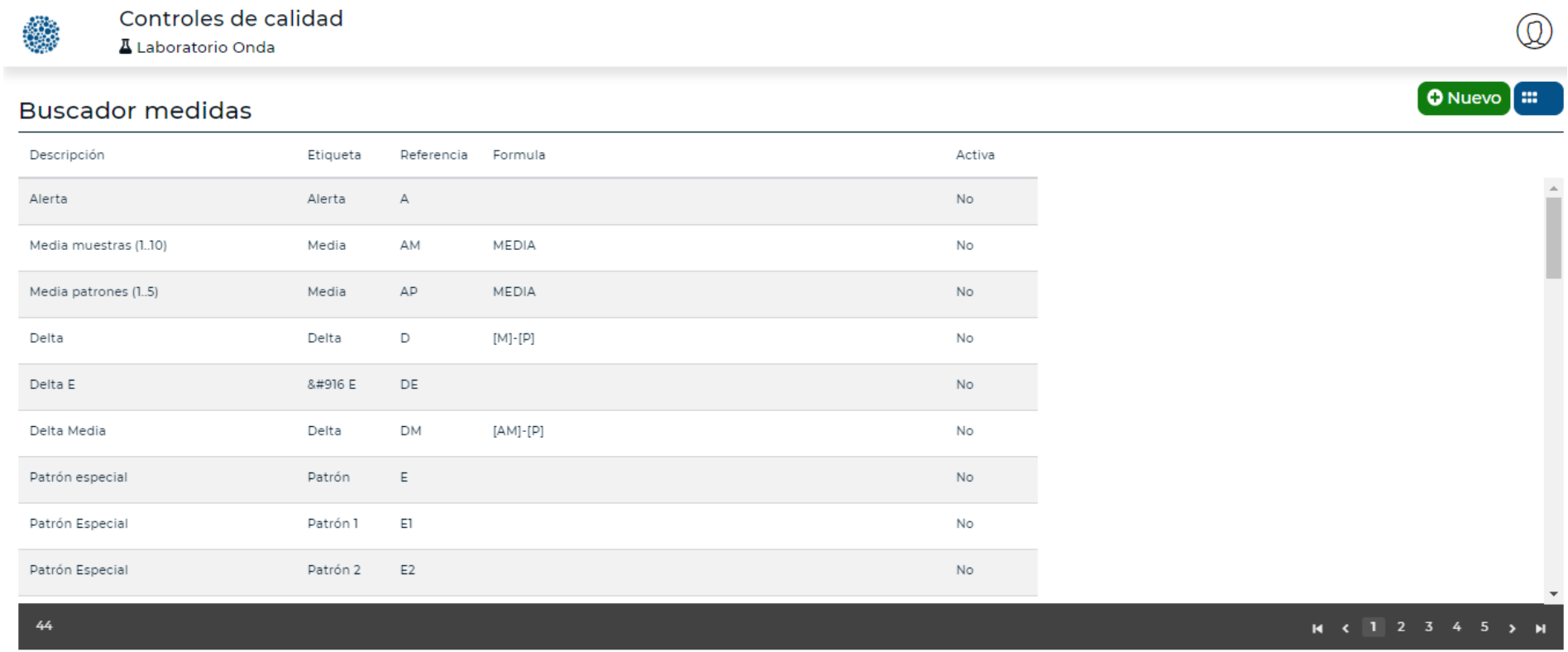

Figura 3.5: Maestro de medidas.

Figura 3.6: Ventana modificar medida.

Como se puede observar en las figuras 3.4, 3.5 y 3.6 los colores predominantes son el blanco, el azul y el gris. Se reservan por otro lado el color rojo como es habitual para la acción *eliminar*, el color verde para *crear* y el color azul para la opción de *modificar*.

La mayoría de los maestros son muy parecidos al de *medidas*, cada uno de ellos tiene una información requerida para su creación, en el maestro *indicadores* se encuentra un menú desplegable para seleccionar una unidad como se ve en la figura 3.7. Por otro lado el configurador

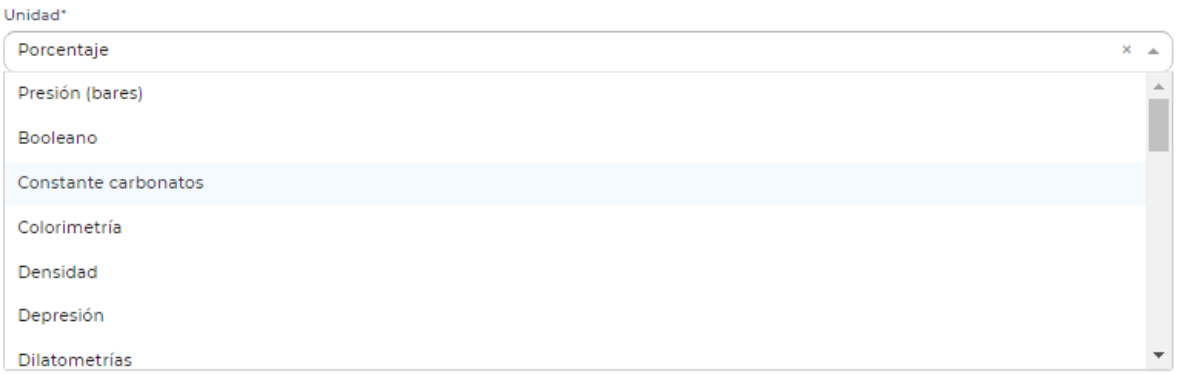

Figura 3.7: Desplegable selección de unidad en el maestro indicadores.

de secciones tiene más complejidad. En la figura 3.8 se pueden ver los datos de la cabecera de la sección en la que nos encontramos y en la figura 3.9 sus indicadores. Esto es así puesto que dentro de una sección nos encontraremos con unos indicadores determinados y una lista que puede ser modificada. Estas listas a su vez contienen listas de medidas que pueden ser modificadas, incluso su campo orden como se puede apreciar con los botones flecha de la figura 3.9. Hay que remarcar que todos estos botones están creados a partir de iconos *favicon*[14], también conocidos como iconos de página, es una pequeña imagen asociada con una página o sitio web en particular.

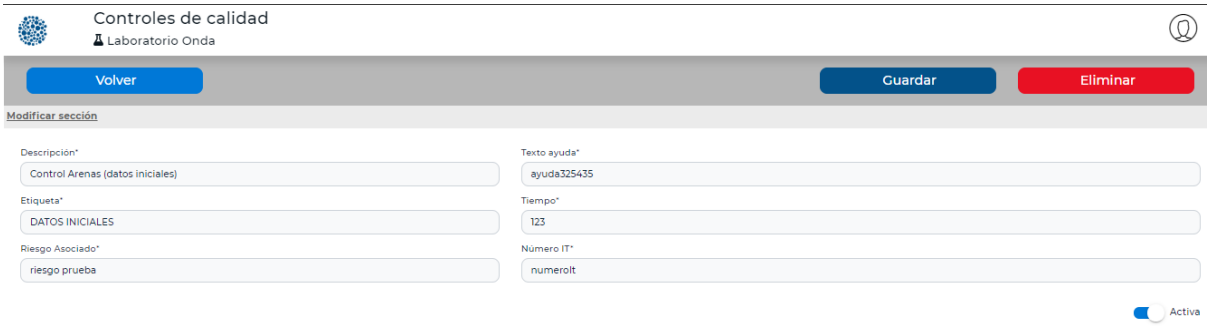

Figura 3.8: Cabecera de una sección

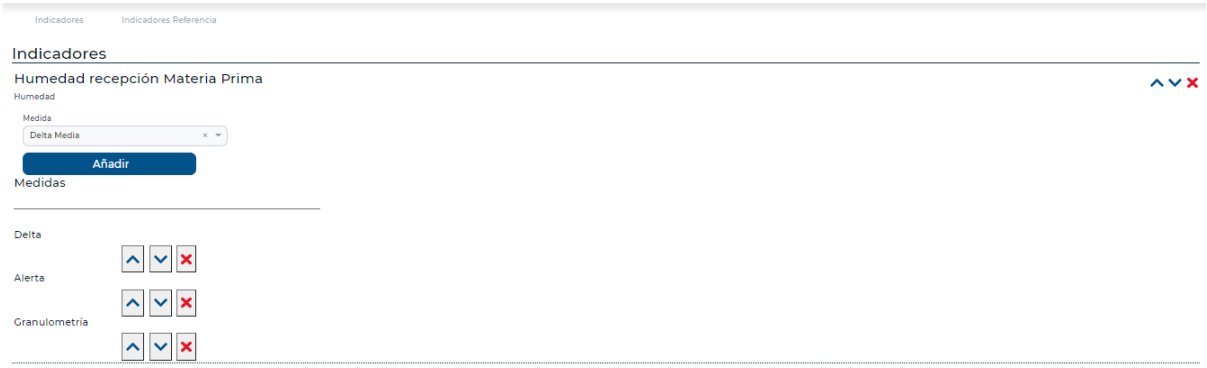

Figura 3.9: Lista de indicadores de una sección

Capítulo 4

Implementación y pruebas

En este capítulo se detalla la implementación tanto de la parte *back-end* como la del *front-end*. Para ello veremos los requisitos de la aplicación web y como estos se han modelado.

4.1. Detalles de implementación

En este capítulo inicial se presentan los detalles de implementación, los patrones de programación y las estrategias empleadas. También se expondrán las decisiones tomadas y los cambios que han surgido en el proyecto fruto de los problemas y dificultades que han ido surgiendo.

Para empezar se va a exponer la estructura del proyecto, tanto de la parte servidor como la cliente. Posteriormente se detallará la labor de programación de los maestros y la parte del configurador de secciones.

4.1.1. Estructura del proyecto

Antes de entrar en detalle con la estructura de ficheros y carpetas de los proyectos hay que destacar que ambas estructuras siguen una línea común que marca la empresa en la mayoría de sus proyectos. Esto es así ya que llevan tiempo incorporando mejoras y han optimizado esta estructura consiguiendo sencillez y potencia.

Back-end

El primer módulo desarrollado fue el *back-end*, fue implementado bajo el *framework* Spring-Boot que simplifica la estructura Modelo-Vista-Controlador. El lenguaje de programación utilizado en esta parte fue Java en su versión 13 y se hizo servir Hibernate como herramienta de mapeo de datos entre la base de datos MySQL y los objetos que la propia aplicación utiliza internamente.

Las labores de este módulo son diversas, siendo la más importante el intercambio de información entre la base de datos, en las que realiza labores de lectura y escritura, y el envío y recepción de datos con el módulo *front-end*. El intercambio con la base de datos se realiza en formato JSON, sin embargo, con el módulo cliente se realiza mediante dos tipos de clases que se verán más adelante. Además del manejo de los datos también realiza modificaciones sobre ellos, ya que es el módulo principal de la capa de control de datos.

Para mostrar la estructura de este componente se verá una visión general en la que se explicarán los principales directorios y por otro lado el esqueleto completo de clases de un maestro en concreto, en este caso el de *áreas* ya que todos los maestros tienen una estructura muy similar. Por último se detallará la implementación del configurador de controles así como de varios métodos relevantes.

La estructura global de este módulo la podemos ver en la figura 4.1. Para exponer con más detalle cada directorio se expondrá una lista de los mismos detallando los aspectos más relevantes y su principal labor.

- Infra: contiene parte de la configuración del proyecto así como las clases administradoras que contienen todos los *manager* desarrollados durante el proyecto para poder ser usados en cualquier parte del mismo. También se localizan en esta carpeta algunas clases abstractas utilizadas en los mapeadores. Estas contienen elementos genéricos de cada tabla de la base de datos como son la fecha de creación, la fecha de modificación, el usuario creación y el usuario modificación, así como el identificador interno que se utiliza para identificar una fila de la base de datos. En este caso los componentes son partes de una vista que se extrae de ella para simplificar su funcionamiento.
- Archivos: este apartado no fue implementado por el autor de esta memoria pero para comprender el proyecto se darán algunos detalles sobre el mismo. Contiene un conversor de las clases lógicas de los controles a un documento, se utiliza para crear estos documentos a partir de un control.
- Controles: en este directorio se realiza la gestión de un nuevo control a partir de una plantilla. Se realiza la copia de los indicadores y se crea la estructura definida por las áreas y las secciones. A partir de los identificadores se completan todos los módulos necesarios para crear un control. En este apartado se realiza la organización mediante listas del esqueleto de un control. Es decir, se van ordenando los campos por indicador, iteración, sección y finalmente por áreas y de esta forma se estructuran adecuadamente.
- Ensayos Clientes: este apartado no fue implementado por el autor de esta memoria. Este módulo fue el primero de varios que se implementarán posteriormente, se utilizarán para realizar distintos controles en función de las distintas sedes de los clientes.
- Maestros: en este apartado se encuentran todos los maestros del sistema, es decir los componentes que configuran un control, estas clases son representaciones de las tablas de la base de datos con las que se puede operar para realizar modificaciones sobre estos datos.
- Plantillas: el módulo de plantillas es muy similar al de controles aunque es mucho más simple, pues la información de cómo crear un control no está estructurada ni ordenada ya que de esto se encarga el módulo controles. Aquí sin embargo se encuentran todos los componentes que debe tener un control sin estar organizados.

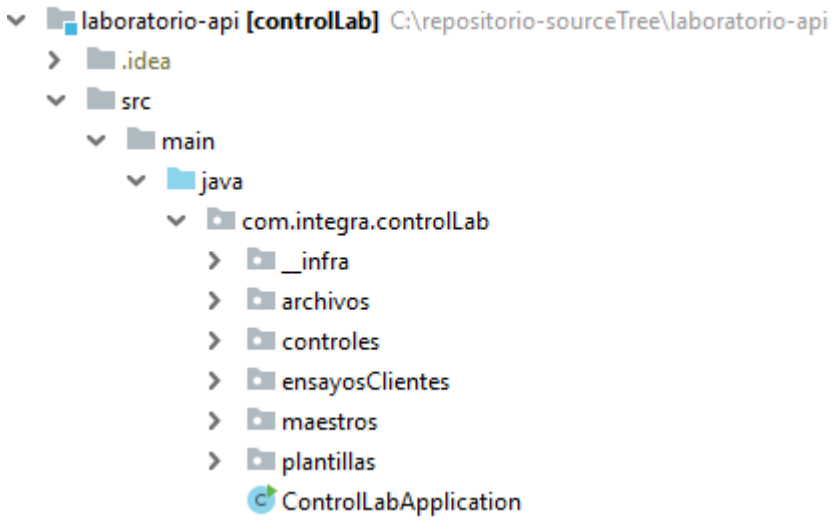

Figura 4.1: Estructura general del *back-end*

Una vez detallada la estructura general del proyecto vamos a exponer con más detalle el módulo de maestros, para ello se expondrá detenidamente el maestro de Áreas. Dado que todos los maestros son muy parecidos y las particularidades de cada uno son muy pequeñas observando éste podemos entender como funcionan todos ellos. En la figura 4.2 se pueden ver todos los maestros, aunque el autor de esta memoria desarrolló los siguientes: áreas, indicadores, medidas, secciones y unidades.

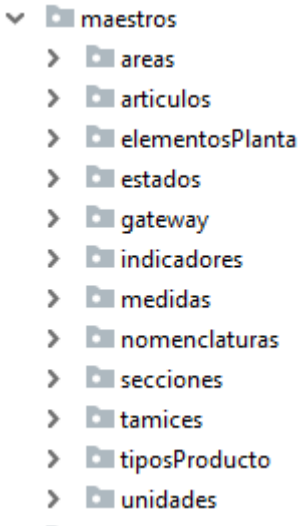

Figura 4.2: Módulo de maestros del *back-end*

En la figura 4.3 se pueden ver las distintas agrupaciones de clases de este módulo. Se describirá cada directorio por separado:

- Controladores: estas clases son las encargadas del intercambio de información con el *front-end* para ello aquí se definen los métodos que utilizará la parte cliente para realizar modificaciones e intercambio de datos. A su vez estos métodos llaman a otros definidos

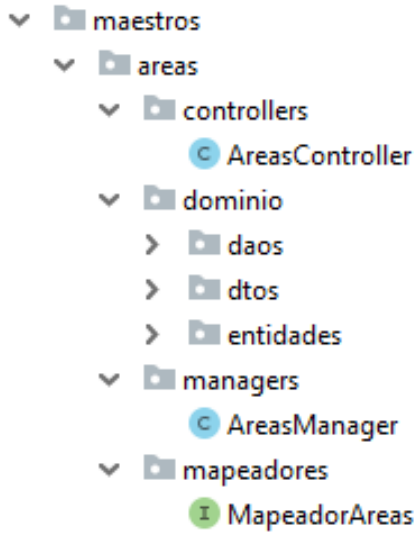

Figura 4.3: Estructura del maestro de Áreas del *back-end*

en el mánager, este módulo se definirá más adelante.

Por otro lado se utilizan anotaciones para diversas funciones, en el listing 4.1 podemos ver estas anotaciones y los métodos que contienen todos los maestros. Las anotaciones son las siguientes:

- `@RestController`: esta anotación es una forma simplificada de utilizar las anotaciones `@Controller` y `@ResponseBody`, es una manera muy cómoda de configurar nuestros servicios REST[16]. REST es una interfaz para conectar varios sistemas basados en el protocolo HTTP y nos sirve para obtener y generar datos y operaciones, devolviendo esos datos en formatos muy específicos, como XML y JSON. Después de utilizarla cada método de esta clase serializa los objetos devueltos en objetos HttpResponse.
- `@RequestMapping`: puede ser aplicada a nivel de clase o de método en este caso se está definiendo la ruta por la cual se va a llamar a los métodos de esta clase (api/v1/areas), es la forma por la que se asignan peticiones HTTP a métodos de un controlador MVC y REST.
- `@Autowired`: esta potente anotación realiza automáticamente la inyección de los campos que se quieran utilizar, en este caso de un AreaManager. Simplemente hay que tener cuidado de que no hayan dos *beans* que sean idénticas puesto que no sabrá cual de las dos utilizar.
- `@ResponseStatus`: se usa para especificar la respuesta HTTP que se obtendrá al utilizar este método.
- `@Get`, `@Post`, `@Put` y `@Delete`: se utilizan para definir el tipo de petición HTTP del método.

Como vemos hay un total de cuatro métodos que son cómunes para todos los maestros, aunque en alguno de ellos se incluye uno más para obtener un elemento por un identificador concreto. Estos métodos son: un tipo *Get* que obtiene una lista con todas las áreas, un tipo *Post* para crear un área nueva, un tipo *Put* para modificar un área y uno del tipo *Delete* para eliminar un recurso.

```
@RestController
@RequestMapping(value = "/api/v1/areas")
public class AreasController {

        @Autowired
        private AreasManager areasManager;

        @ResponseStatus(OK)
        @GetMapping
        public List<AreaInfoDto> obtenerAreas() {
              return areasManager.obtenerAreas();
        }

        @ResponseStatus(CREATED)
        @PostMapping
        public AreaInfoDto crear
        (@RequestBody @Valid AreaFormDto datos) {
              return areasManager.guardar(datos);
        }

        @ResponseStatus(CREATED)
        @PutMapping(value = "{id:\\d+}")
        public AreaInfoDto modificar(@PathVariable Long id,
         @RequestBody @Valid AreaFormDto datos) {
              datos.setId(id);
              return areasManager.guardar(datos);
        }

        @DeleteMapping(value = "{id:\\d+}")
        public ResponseEntity borrar(@PathVariable Long id) {
              try {
                      areasManager.borrar(id);
              return new ResponseEntity(OK);
              } catch (NotFoundException e) {
                      return new ResponseEntity(NOT_FOUND);
              }
        }

}
```

Listing 4.1: Control de Áreas del *back-end*

- Dominios: esta agrupación de clases son las transportadoras de información, cada una de ellas se utiliza para recibir, enviar o modificar datos. Se dividen en tres directorios como podemos ver en la figura 4.4:

  - DAOs o *Data Access Object*: son interfaces que extienden a una interfaz común para el proceso de copia desde la base de datos a los objetos del *back-end* para su manipulación. En esta interfaz se definen métodos que implican inserciones, consultas o modificaciones.
  - DTOs o *Data Transfer Object*: son las clases de los objetos empleados en el intercambio de información entre el *back-end* y el *front-end*.
    Siempre tenemos dos tipos, los que incluyen Form en su nombre son los objetos que nos devolverá la parte cliente, en ellos se puede especificar mediante anotaciones que un campo no puede ser nulo con la anotación `@NotNull` o que otro campo tiene que tener una longitud máxima de 50 caracteres con la anotación `@Size("max=50")` por ejemplo. El otro tipo son los que contienen *info* en su nombre. Estas son las clases que

la parte servidor envía a la cliente, con la anotación `@Data` de lombok se automatizan los *setters* y *getters* de todos los campos que definamos en la clase, por lo tanto nuestra clase queda muy limpia y su código se reduce a los campos que nos interesa enviar.

- Entidades: las entidades son las clases que representan fielmente a las tablas de la base de datos y son de este tipo los objetos con los que vamos a operar principalmente en el back-end, ya que el proceso de manipulación de los datos suele ser el siguiente, se obtiene un objeto *Form* del *front-end* se transforma a un objeto entidad para operar con él, una vez se ha operado sobre el objeto se transforma a un objeto de tipo *Info* para enviarlo de vuelta al cliente.

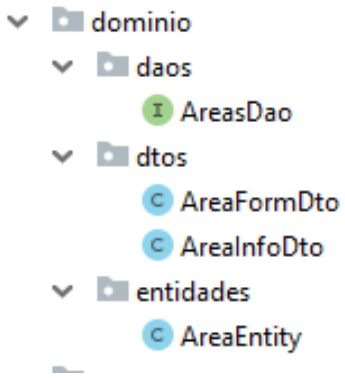

Figura 4.4: Dominio de Áreas del *back-end*

- Mánagers: estas clases se encargan de la transformación de los datos, todos los procesos por los que pasan los datos ocurren en este módulo se puede decir que son el núcleo de control de datos del *back-end*.

  Como se ha comentado anteriormente los controladores llaman a métodos de esta clase para que procesen unos datos y sean devueltos. Normalmente recibe objetos de tipo *Form*, los transforma a un objeto tipo *Entidad* para operar con él y envía un objeto tipo *Info* con el resultado. Todas estas transformaciones de objetos las realiza la clase mapeadora que veremos a continuación.

  Por otro lado todos los mánagers implementan una interfaz que contiene unos métodos que son utilizados en todos los proyectos de la empresa, estos métodos son:

  - Guardar: si el elemento ya existe lo modifica y si no existe lo crea.
  - Guardad nuevo: solo guarda el elemento si este no existe aún.
  - Modificar: modifica el estado de un elemento si este existe.
  - Obtener Datos Salida: transforma un objeto Entidad en un objeto *Info*
  - Existe: devuelve si un elemento existe o no.
  - Obtener: obtiene un elemento por un identificador.
  - Borrar: elimina un elemento si este existe.

  Además de estos métodos se ha implementado otro adicional que devuelve una lista con todas las áreas. Podemos ver en el listing 4.2 algunos de los métodos implementados además de algunas anotaciones, en concreto la anotación @Transaccional hace que la

operación sea transaccional, en este caso además se especifica que no se llevará a cabo la operación si se produce una excepción de esta clase utilizando el parámetro *rollback*.

```
@Service
public class AreasManager implements
BasicManagerInterface<AreaInfoDto, AreaEntity, AreaFormDto> {
        @Autowired
        private DaoManager daoManager;
        @Autowired
        private MapperManager mapperManager;

        public List<AreaInfoDto> obtenerAreas() {
                return obtenerDatosSalida(
                daoManager.getAreasDao().findAll());
        }

        @Transactional(rollbackFor = Exception.class)
        @Override
        public AreaInfoDto guardar(AreaFormDto datos) {
                try {
                        AreaEntity areaEntity =
                        mapperManager.
                        getMapeadorAreas().toEntidad(datos);

                        if (!existe(areaEntity.getId())) {
                                return
                                obtenerDatosSalida(guardarNuevo(areaEntity));
                        } else {
                                return
                                obtenerDatosSalida(modificar(areaEntity));
                        }
                } catch (Exception e) {
                        throw new NotFoundException(
                        InfoExceptionEnum.ERROR_GUARDANDO_CONTROL);
                }
        }

        @Override
        public AreaEntity guardarNuevo(AreaEntity areaEntity) {
                return daoManager.getAreasDao().save(areaEntity);
        }
```

Listing 4.2: Mánager de Áreas del *back-end*

- Mapeadores: como se ha especificado anteriormente el mapeador se encargar de la transformación de objetos Form, Info y Entidad entre sí mismos, para ello se utiliza la anotación `@Mapper(componentModel = "spring")` que automatiza el mapeado de datos de aquellos atributos que tengan el mismo nombre y el mismo tipo en los dos objetos. Sin embargo si en un objeto tenemos contenido un dato que es del tipo de otra clase del proyecto debemos definir la clase origen y destino de mapeado de este objeto con la anotación `@Mapping(source = 'AreaFormDto', target = 'AreaEntity')`.

En esta segunda parte del módulo servidor se va a detallar parte de la funcionalidad desarrollada en el módulo Control. Los métodos que se explicarán a continuación consiguen que a partir de una plantilla se pueda obtener un control con una organización y ordenación correcta. Además de otras modificaciones sobre un control ya hecho.

En la clase controladora del control, listing 4.3, se pueden ver los tres métodos que detallaré su funcionalidad general a continuación:

```
@ResponseStatus(OK)
@GetMapping(value = "{id:\\d+}")
public ControlInfoDto obtenerControl(@PathVariable Long id) {
        return managerManager.
        getSerializarControlManager()
        .obtenerControlSerializado(id);
}
@ResponseStatus(CREATED)
@PostMapping(value = "/modificar-iteracion")
public ControlInfoDto modificarIteracionControl(
        @RequestBody @Valid List<IndicadoresEnIteracionFormDto> datos) {
        return managerManager.getModificarIteracionControlManager()
        .modificarIteracionControl(datos);
}

@ResponseStatus(CREATED)
@PostMapping(value = "/indicador-referencia")
public ControlInfoDto modificarIndicadorReferencia(
        @RequestBody @Valid ControlIndicadorFormDto datos) {
        return managerManager.getModificarIndicadorReferenciaControlManager()
        .modificarIndicadorReferencia(datos);
}


@ResponseStatus(OK)
@GetMapping(value = "/copiar-control/{id:\\d+}")
public ControlInfoDto copiarControl(@PathVariable Long id) {
        return managerManager.getNuevoControlManager().copiarControl(id);
}

@DeleteMapping(value = "{id:\\d+}")
public ResponseEntity borrar(@PathVariable Long id) {
        try {
                controlesManager.borrar(id);
                return new ResponseEntity(OK);
        } catch (NotFoundException e) {
                return new ResponseEntity(NOT_FOUND);
        }
}
```

Listing 4.3: Métodos del controlador de un control del *back-end*

- Obtener control: en este método obtenemos una variable que viene desde la ruta especificada, este parámetro es un identificador de una plantilla desde la cual debemos crear el nuevo control.

  Para ello, como se puede ver en el listing 4.3, en el primer método se llama a la función de la clase Serializar Control Mánager, en esta clase se especifican todos los métodos para obtener un control bien estructurado. A continuación se detallarán todos los métodos que se utilizan para este cometido.

  El primer método que observamos en el listing 4.4 es el que se utiliza en el controlador. Como se puede ver en las primeras líneas se utiliza el identificador proporcionado para obtener un nuevo control. Este control se encuentra vacío pues aún no tiene áreas ni secciones, para ello lo que se hace es ir añadiendo áreas y a estas los identificadores por

sección, manteniendo en todo momento un control de no repetidos en ambos casos. Para ello se utiliza el método del listing 4.5. Este proceso se realiza con cada Control Indicador, este objeto tiene asociado un área, una sección, una iteración y unas medidas, por lo que cada elemento se organiza en función de todos estos parámetro. Todos ellos se guardan en listas pues primero se intentó la implementación con diccionarios pero se desechó por la complejidad extra que suponían estos.

```
        public ControlInfoDto obtenerControlSerializado(Long idControl) {
                ControlEntity control = obtenerControl(idControl);
                List<ControlIndicadorEntity> listControlIndicadorEntity =
                control.getIndicadores();
                List<ControlAreaInfoDto> listAreas = new ArrayList<>();

                for (ControlIndicadorEntity controlIndicadorEntity :
                listControlIndicadorEntity) {
                        serializadoAreasSecciones(listAreas, controlIndicadorEntity);
                        serializadoIteracionesIndicadoresPorSeccion(
                        listAreas, controlIndicadorEntity);
                }
                ControlInfoDto controlInfoDto =
                mapperManager.getMapeadorControles().toInfoDto(control);
                listAreas.sort(
                Comparator.comparing(ControlAreaInfoDto::getOrden));
                controlInfoDto.setAreas(listAreas);
                return controlInfoDto;
        }
```

Listing 4.4: Método principal del mánager del Control del *back-end*

```
private void serializadoAreasSecciones(List<ControlAreaInfoDto> listAreas
, ControlIndicadorEntity controlIndicadorEntity) {
        ControlSeccionInfoDto controlSeccionInfoDto;
        ControlAreaInfoDto controlAreaInfoDto =
        getControlAreaInfoDto(listAreas, controlIndicadorEntity);
        //Area nueva
        if (controlAreaInfoDto == null) {
                controlAreaInfoDto = getControlAreaInfoDto(controlIndicadorEntity);
                controlSeccionInfoDto =
                 getControlSeccionInfoDto(controlIndicadorEntity);
                controlAreaInfoDto.setSecciones(new ArrayList<>());
                controlAreaInfoDto.getSecciones().add(controlSeccionInfoDto);
                listAreas.add(controlAreaInfoDto);
        } else { //anyadir seccion a una Area existente
                controlSeccionInfoDto = getControlSeccionInfoDtoFromArea(
                controlIndicadorEntity, controlAreaInfoDto);
                //Solo la anyado en caso de que no la tenga
                if (controlSeccionInfoDto == null) {
                controlSeccionInfoDto =
                getControlSeccionInfoDto(controlIndicadorEntity);
                controlAreaInfoDto.getSecciones().add(controlSeccionInfoDto);
                }
        }
}
```

Listing 4.5: Método organizador de áreas y de secciones del mánager del control

Justo después se utiliza el método serializar iteraciones e indicadores por sección. Realiza una ordenación de los indicadores por iteración dentro de cada sección, aunque normalmente cada sección suele tener solo una iteración .Puede haber controles de múltiples

temperaturas dentro de una sección lo que ocasiona que hayan varias iteraciones de un indicador.

Cabe destacar la utilización de los *streams* de Java para la búsqueda dentro de las listas, para ello los *streams* tiene el método *filter* que permite comparar cualquier atributo del objeto que se recorre. Esto es necesario puesto que se requiere la búsqueda por varios parámetros de los objetos, como son el identificador de un área o de una sección. Se puede observar un ejemplo del uso de *streams* para este propósito en el listing 4.6. Como se puede ver primero se extrae la lista de secciones de un área que concuerda con el identificador de área del control indicador pasado como parámetros y finalmente antes de devolver la lista de secciones se filtra la sección por el identificador de sección del control indicador.

Por último antes de devolver el control bien construido se transforma el objeto Entidad a InfoDto y antes de ser enviado la lista de áreas se ordenan por su atributo llamado orden y se añade al control.

```
private ControlSeccionInfoDto getControlSeccionInfoDtoFromArea(
        ControlIndicadorEntity controlIndicadorEntity,
        List<ControlAreaInfoDto> listAreas) {
                List<ControlSeccionInfoDto> listaSecciones =
                listAreas.stream()
                .filter(area -> controlIndicadorEntity.getArea()
                .getId().equals(area.getId()))
                .findAny()
                .orElse(null).getSecciones();
                return listaSecciones.stream()
                .filter(seccion -> controlIndicadorEntity.getSeccion()
                .getId().equals(seccion.getId()))
                .findAny()
                .orElse(null);
}
```

Listing 4.6: Empleo de *streams* en la clase mánager de un Control

- Modificar iteración: esta función se encarga de la modificación concreta de una iteración dentro de una sección cambiando sus medidas por la lista de medidas enviada como argumento. La dificultad que entrañaba este método es el hecho de que las medidas de un indicador están guardadas en base de datos en formato JSON, por lo tanto se me encargó la tarea de buscar una librería adecuada para el manejo de este tipo de datos. Por ello se utilizó la librería Gson[17] puesto que su uso más frecuente es para convertir un objeto en su representación JSON y a la inversa.

  Para realizar el cambio de medidas primero se necesitan deserializar las medidas existentes en base de datos para comprobar aquellas que coincidan con la lista pasada para modificar unicamente su valor. Se puede observar en el primer comentario del código del listing 4.7 como el primer paso es convertir los valores de la base de datos, que son un *string* en formato JSON, a una lista de objetos JSON de medidas.

```
public Long guardarIndicadoresPorIteracion(
IndicadoresEnIteracionFormDto iteracionIndicadoresFormDto) {

        ControlIndicadorEntity controlIndicadorEntity =
        obtenerControlIndicador(iteracionIndicadoresFormDto.getId());
        String valores = controlIndicadorEntity.getValores();
        List<ValoresFormDto> listaValoresForm =
        iteracionIndicadoresFormDto.getListaValores();
        //Deserializar el JSON de la BBDD
```

```
                List<MedidasJsonEntity> listaValores =
                getValoresDeserializados(valores);
                //Setear el campo valor en los campos coincidentes.
                addValorInListaMedidas(listaValoresForm, listaValores);
                //Serializar la lista de medidas para guardarla en BBDD
                Gson customGson = getGsonCustom();
                String valoresNuevos = customGson.toJson(listaValores);
                controlIndicadorEntity.setValores(valoresNuevos);
                //vemos si el indicador esta completado
                marcarIndicadorComoCompletado(
                iteracionIndicadoresFormDto, controlIndicadorEntity);
                //Guardar el control indicador en BBDD
                if (existe(controlIndicadorEntity.getId())) {
                        try {
                                daoManager.getControlesIndicadoresDao()
                                .save(controlIndicadorEntity);
                                return controlIndicadorEntity.getControl().getId();
                        } catch (Exception e) {
                                throw new
                                NotFoundException(
                                InfoExceptionEnum.ERROR_GUARDANDO_CONTROL);
                        }
                }
                return null;
        }
```

Listing 4.7: Método para modificar las medidas por iteración

Este proceso se encarga de hacerlo el método del listing 4.8 el funcionamiento es bastante trivial pues con un objeto Gson solo tenemos que indicarle el tipo de elemento al que tiene que convertir el *string* y los valores a convertir para que haga el proceso automáticamente, para ello los campos de JSON deben coincidir con los del objeto a transformar.

```
        private List<MedidasJsonEntity> getValoresDeserializados(String valores) {
                Type listType =
                new TypeToken<ArrayList<MedidasJsonEntity>>() {}.getType();
                List<MedidasJsonEntity> listaMedidas =
                new Gson().fromJson(valores, listType);
                return listaMedidas;
        }
```

Listing 4.8: Deserializado de medidas desde la base de datos

Luego simplemente se recorre la lista de objetos y se cambia el campo valor de las medidas para aquellas con un identificador coincidente. A continuación hay que volver a guardar el *string* JSON en la base de datos, para ello se utilizó un objeto Gson creado a propósito para este cometido. Como se puede ver en el listing 4.9 se indican el nombre de las propiedades y su valor a continuación, para que este sea convertido a un formato cadena.

Por último se comprueba que el indicador esté completo y se guarda en base de datos la nueva cadena con los valores actualizados de las medidas, esto se puede ver en las últimas líneas de código del listing 4.7

```
private Gson getGsonCustom() {
        GsonBuilder gsonBuilder = new GsonBuilder();
        gsonBuilder.serializeNulls();
        JsonSerializer<MedidasJsonEntity> serializer =
        new JsonSerializer<MedidasJsonEntity>() {
        @Override
                public JsonElement serialize(
```

```
                        MedidasJsonEntity src, Type typeOfSrc,
                        JsonSerializationContext context) {
                        JsonObject jsonSeccionIndicador = new JsonObject();
                        jsonSeccionIndicador.addProperty("medida", src.getMedida());
                        jsonSeccionIndicador.addProperty("readOnly", src.getReadOnly());
                        jsonSeccionIndicador.addProperty("orden", src.getOrden());
                        jsonSeccionIndicador.addProperty("valor", src.getValor());
                        return jsonSeccionIndicador;
                }
        };
        gsonBuilder.
        registerTypeAdapter(SeccionIndicadorValoresFormDto.class, serializer);
        return gsonBuilder.create();
}
```

Listing 4.9: Serializador *ad hoc* para las medidas del control

- Modificar indicador referencia: este método se encarga de modificar el indicador de referencia, listing4.10. Este indicador representa una relación con un indicador con sus medidas vacías, es decir, está por completar aún. Este indicador puede ser uno o varios, por lo tanto como pasaba con el método anterior se guarda en formato JSON, por lo que el proceso es muy parecido al anterior.

  Primero se busca el indicador a modificar, pero ahora los datos no se deserializan desde la base de datos ya que la cadena que obtenemos debemos macharla sobre la que hay en base de datos, por lo tanto solo tenemos que serializarla y guardarla en el lugar adecuado. El código que realiza esta función se puede observar en la figura

```
public ControlInfoDto modificarIndicadorReferencia(
ControlIndicadorFormDto datos) {
        ControlIndicadorEntity cie =
        daoManager.getControlesIndicadoresDao().getOne(datos.getId());
        // tenemos que modificar todos los registros
        //que tengan la misma area, seccion, iteracion y control
        List<ControlIndicadorEntity>
        listaIndicadoresModificar =
        daoManager.getControlesIndicadoresDao().
        obtenerIndicadoresIguales(
                cie.getControl().getId(),
                cie.getArea().getId(),
                cie.getSeccion().getId(),
                cie.getIteracion());
        String nuevoJson =
        generarNuevoJsonIndicadorReferencia(cie, datos.getValor());
        for (ControlIndicadorEntity controlIndicadorEntity :
        listaIndicadoresModificar) {
                controlIndicadorEntity.setIndicadoresReferencia(nuevoJson);
                daoManager.getControlesIndicadoresDao()
                .save(controlIndicadorEntity);
        }
        return managerManager.getControlesManager().
        obtenerDatosSalida(
        managerManager.
        getControlesManager().obtener(cie.getControl().getId()));
}
```

Listing 4.10: Función para modificar los indicadores de referencia.

- Obtener Datos Salida: transforma un objeto Entidad en un objeto

Front-end

En el módulo cliente se utilizó un *framework* para aplicaciones web desarrollado en TypeScript llamado Angular. Para ello se ha seguido un proyecto base que es el que utiliza la empresa en la mayoría de sus proyectos. Esta estructura dispone de algunos módulos integrados para su utilización en cualquier proyecto, estos suelen ser funciones básicas como un botón de guardado o de borrado, un *grid* o tabla para listar elementos o un desplegable para escoger un elemento.

Por otro lado el esqueleto del proyecto contiene múltiples directorios y ficheros puesto que se busca la máxima separación de los elementos para que todos los archivos contengan el código justo y necesario.

En la figura 4.5 se puede observar la disposición general de directorios del proyecto en el editor de código Visual Studio Code. La mayoría de directorios contienen archivos de configuración tanto del propio editor como del entorno utilizado Node.js. Sin embargo, el directorio base contiene los módulos comunes comentados anteriormente y que se han ido añadiendo para que se puedan utilizar en todos los proyectos de la empresa sin necesidad de volver a implementarlos. Y el directorio en el que se ha desarrollado casi la totalidad de código es el directorio funcionalidad, esta carpeta contiene todas las vistas y componentes de cada uno de los maestros que tenemos en el sistema. En concreto el autor de esta memoria desarrolló los maestros

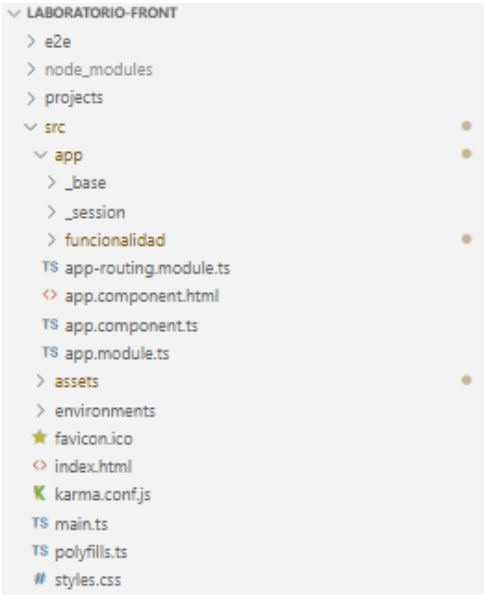

Figura 4.5: Estructura general del *front-end*

medidas, indicadores y secciones, además de implementar el configurador de una sección por el que podemos crear una sección desde cero e ir añadiéndole elementos hasta que esta quede completa, para un mayor detalle se expondrán cada maestro por separado.

- Maestro de medidas: en la figura 4.6 podemos ver la estructura de un maestro. A continuación se explicará cada uno de los directorios mostraros en la figura 4.6:
  - Componentes: este directorio se utiliza para crear un componente propio de este

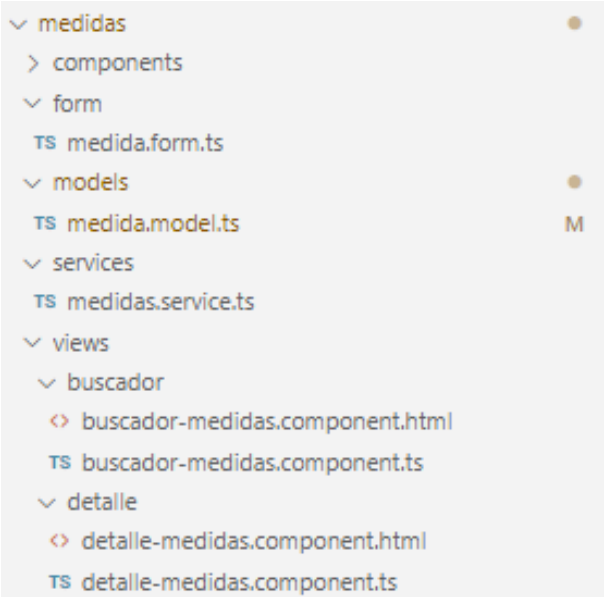

Figura 4.6: Estructura del maestro de medidas del *front-end*

maestro, en este caso se han utilizado componentes del módulo base para su implementación por lo que se encuentra vacío.

- Formulario: esta carpeta contiene la clase TypeScript medida.form que especifica los requisitos del formulario para la creación o modificación de una medida. Como podemos ver en el listing 4.11 se especifican todos los campos como obligatorios y se les asigna una longitud máxima, esta viene determinada por el tamaño de registro que hayamos definido en base de datos.

```
import { Validators } from '@angular/forms';

export const MedidaForm = {
        descripcion: ['', [Validators.required, Validators.maxLength(50)]],
        etiqueta: ['', [Validators.required, Validators.maxLength(50)]],
        referencia: ['', [Validators.required, Validators.maxLength(2)]],
        formula: ['', [Validators.required, Validators.maxLength(500)]]
};
```

Listing 4.11: Formulario del maestro medidas.

Hay que destacar que posteriormente en la vista HTML se pueden añadir alertas de formulario cuando un campo este incompleto o su formato no sea el adecuado.

- Modelos: en este apartado se definen los atributos que tendrá el maestro, además se pueden realizar operaciones en función del estado de algún campo, como se puede ver en el listing 4.12 si el campo activo contiene el valor *True* se asigna a una nueva variable llamada activaMostrar la cadena 'Si'.

  Además en el listing 4.12 se pueden ver los métodos por defecto que contiene un modelo, estos son: la función *deserialize*; empleada para obtener el objeto con todos los campos correctos desde el objeto que se ha recibido desde el módulo back-end, *prepare*; en función del valor de algunos campos recibido asigna un valor u otro a una nueva variable, y *initialize*; prepara el modelo para poder ser usado en otro lugar.

```
export class MedidaModel extends BaseModel {
```

```
        descripcion: string;
        etiqueta: string;
        formula: string;
        referencia: string;
        activo: boolean;
        activoMostrar: string;
        orden: number;
        readOnly: boolean;


        initialize(): MedidaModel {
                this.activo = true;
                return this;
        }

        deserialize(input: any) {
                Object.assign(this, input);
                return this.prepare();
        }


        prepare() {
                if (this.activo) {
                        this.activoMostrar = 'Si';
                } else {
                        this.activoMostrar = 'No';
                }
                        return this;
        }
}
```

Listing 4.12: Modelo del maestro medidas del *front-end*.

- Servicios: contiene los métodos que realizan las llamadas al módulo servidor como podemos ver en el listing 4.13. Como vemos también se define aquí la URL base que debe coincidir con la declarada en el *back-end* para el maestro de medidas.

```
@Injectable()
export class MedidasService extends CrudService {
        urlBase = 'medidas';

        constructor(public httpClient: HttpClient) {
                super(httpClient);
        }

        get(id: number): Observable<MedidaModel> {
                return this.httpClient.get(this.urlBase + '/' + id)
                .pipe(
                map((body: any) => new MedidaModel().deserialize(body))
                );
        }

        getTodos(): Observable<Array<MedidaModel>> {
                return this.httpClient.get(this.urlBase)
                .pipe(
                map( (body: any) => body.map( (dato: MedidaModel) =>
                new MedidaModel().deserialize(dato) ) )
                );
        }
```

```
}
```

Listing 4.13: Servicio del maestro de medidas del *front-end*

En este caso podemos ver que contiene dos métodos, uno de ellos se encarga de obtener una medida por identificador y el otro obtiene una lista con todas las medidas. Esto nos servirá en las vistas cuando queramos ver en una tabla todas las medidas o cuando hagamos clic en una de ellas para observar una vista detallada de la misma.

- Vistas: las vistas de todos los maestros incluyen dos divisiones, el buscador muestra una tabla con todos los maestros y una barra de navegación para paginar y poder ver todos los registros, si se hace clic en un elemento de la tabla se muestra la vista detalle, que permite modificar un maestro y ver sus atributos completos. Además cada uno de estos dos apartados contiene dos ficheros, uno HTML y un componente TypeScript en el que se suelen definir métodos utilizados en el HTML.

  En concreto la vista buscador prácticamente no contiene líneas de código puesto que utiliza el componente común *grid* para mostrar en una tabla todas las medidas. Para ello se crea una clase llamada *medidas* dentro del directorio del componente grid y se exportan todos los atributos a mostrar como se puede ver en el listing 4.14. Simplemente utilizando el componente común se obtiene la vista que podemos ver en la figura 4.7.

```
const propiedades: Array<GridModel> = [
        { name: 'Descripción', prop: 'descripcion', sort: true, width: 300 },
        { name: 'Etiqueta', prop: 'etiqueta', sort: true, width: 100 },
        { name: 'Referencia', prop: 'referencia', sort: true, width: 100 },
        { name: 'Formula', prop: 'formula', sort: true, width: 500 },
        { name: 'Activa', prop: 'activoMostrar', sort: true, width: 100 }
];

const propiedadesExportador = (obj: MedidaModel) => ({
        descripcion: obj.descripcion,
        etiqueta: obj.etiqueta,
        formula: obj.formula,
        referencia: obj.referencia,
        activa: obj.activoMostrar
});
```

Listing 4.14: *Grid* del maestro medidas del *front-end*.

Por otro lado la vista detalle que se puede ver en la figura 4.8 se implementó gracias a un archivo TypeScript detalle-medidas-component.ts en el que se definen las llamadas al servicio de medidas para obtener los datos de una medida concreta. Hay que destacar que la barra superior en la que se encuentran los botones volver, guardar y eliminar es un componente del módulo base que se emplea en todos los proyectos de la empresa y se utiliza porque simplifica de gran manera la implementación de guardar una medida o eliminarla, pues lo realiza de forma automática ejecutando la llamada *POST* para modificar y la llamada *DELETE* para eliminar del *back-end*. El fichero HTML por otro lado se encarga de crear una tabla con los inputs necesarios del modelo de medidas, se puede ver parte de este fichero en el listing 4.15. Mediante el uso del componente app-input se puede especificar el nombre del control, el valor, si es un campo o no requerido, su longitud máxima, si es solo de lectura y cual es su campo de control entre muchas otras opciones.

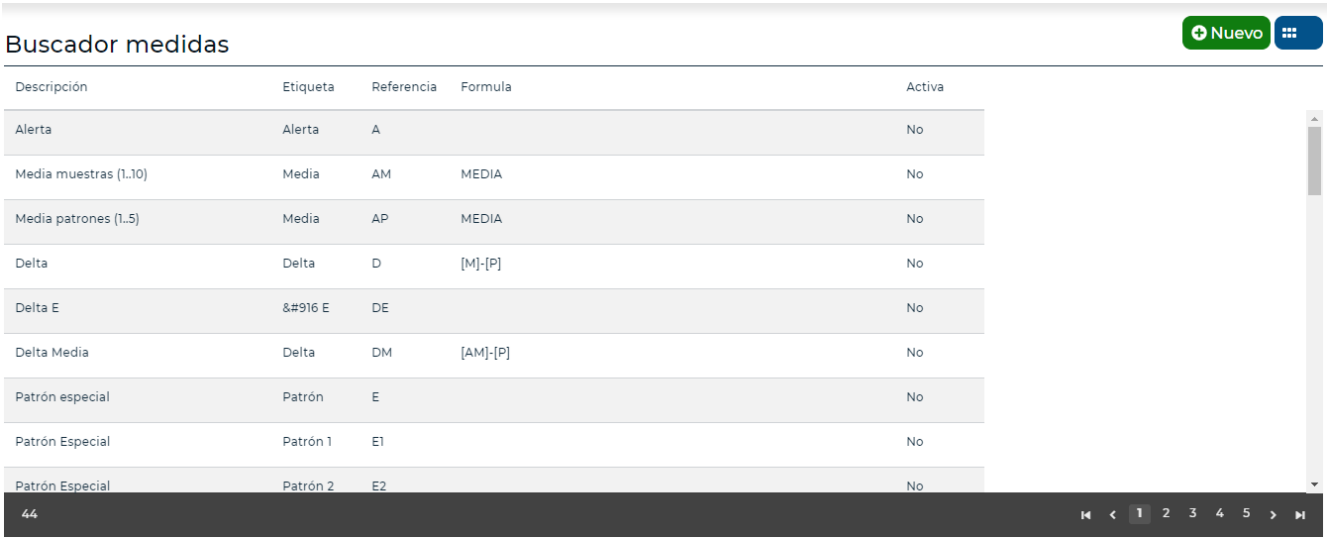

Figura 4.7: Captura del buscador de medidas de la aplicación web.

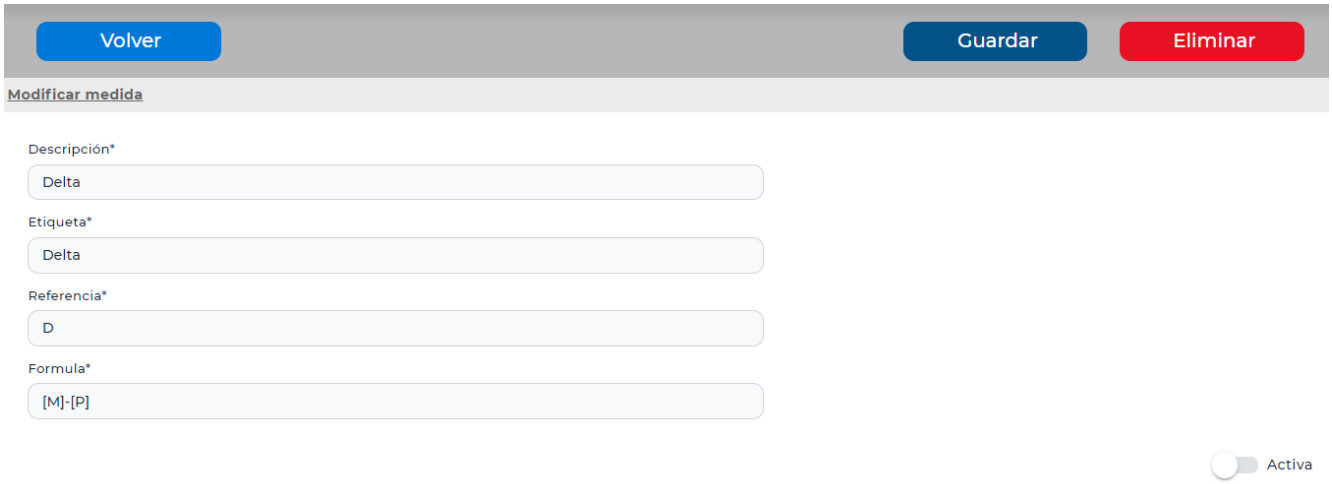

Figura 4.8: Captura del detalle de una medida de la aplicación web.

```
<div class="row">
        <div class="col-12 pt-3">
                <form [formGroup]="form" *ngIf="modelo" class="p-2">
                <div class="row">
                        <div class="col-7">
                        <app-input [etiqueta]="'medidas.descripcion'"
                                [requerido]="true"
                                [controlName]="descripcion"
                                [(valor)]="modelo.descripcion"
                                [type]="'text'"
                                [maxLength]="50"
                                [readonly]="false"
                                [control]="form.controls.descripcion">
                        </app-input>
                        </div>
                        <div class="col-7">
                        <app-input [etiqueta]="'medidas.etiqueta'"
                                [requerido]="true"
                                [controlName]="etiqueta"
                                [(valor)]="modelo.etiqueta"
                                [type]="'text'"
                                [maxLength]="50"
                                [readonly]="false"
                                [control]="form.controls.etiqueta">
                        </app-input>
                        </div>
                        <div class="col-7">
                        <app-input [etiqueta]="'medidas.referencia'"
                                [requerido]="true"
                                [controlName]="referencia"
                                [(valor)]="modelo.referencia"
                                [type]="'text'"
                                [maxLength]="2"
                                [readonly]="false"
                                [control]="form.controls.referencia">
                        </app-input>
                        </div>
                        <div class="col-7">
                        <app-input [etiqueta]="'medidas.formula'"
                                [requerido]="true"
                                [controlName]="formula"
                                [(valor)]="modelo.formula"
                                [type]="'text'"
                                [maxLength]="500"
                                [readonly]="false"
                                [control]="form.controls.formula">
                        </app-input>
                        </div>
                        </div>
                        <div class="row">
                        <div class="col-12">
                        <app-input-toggle
                        class="float-right mt-4"
                                [(valor)]="modelo.activo"
                                [etiqueta]="'medidas.activo'"></app-input-toggle>
                        </div>
                </div>
                </form>
        </div>
```

```
</div>
```

Listing 4.15: Fichero HTML del detalle de una medida del front-end

- Maestro de indicadores: este maestro es muy parecido al explicado en el punto anterior salvo que se utiliza un componente desplegable. Esto es así puesto que cada indicador tiene asociada una unidad, por ejemplo el indicador temperatura tiene asociada la unidad grados centígrados.

  Para poder obtener todas las unidades del sistema y mostrarlas en la lista desplegable hay que hacer una llamada al cargar la vista para obtener todas las unidades desde el back-end. Esto se consigue con el fragmento de código del listing 4.16 que utiliza la llamada HTTP del servicio de las unidades al servidor para obtener un listado de esas unidades que necesitamos.

```
obtenerUnidades() {
        this.loaderService.show();
        this.unidadesService.getTodos().pipe(finalize(() => {
                this.loaderService.hide();
        })).subscribe((response) => {
                const data = response.map(
                        (unidad: UnidadModel) =>
                        new UnidadModel().deserialize(unidad)
                );
                this.unidades = data;
                this.mostrar = true;

        });
}
```

Listing 4.16: Método para cargar las unidades del desplegable de un indicador del *front-end*.

  Por otro lado en el fichero HTML para utilizar el componente *select* se utilizó la etiqueta `'<app-input-select>'` y como parámetros se le pasa, entre otras cosas, la lista de unidades que debe mostrar. El resultado lo podemos observar en la figura 4.9

- Maestro y configurador de secciones: en el apartado de secciones además de disponer todos los datos en una tabla en la vista detalle se ha incluido funcionalidad para poder configurar una sección.

  Una sección contiene indicadores y estos indicadores a su vez contienen medidas, este configurador nos permite añadir, ordenar y eliminar indicadores de una sección y sobre estos indicadores nos permite añadir, ordenar y eliminar sus medidas. Además se crearon dos pestañas para en un futuro implementar algo parecido con los indicadores de referencia. En la figura 4.10 se puede ver el resultado del detalle de una sección sobre la que se pueden configurar los parámetros comentados anteriormente y en la figura 4.11 se observan los datos de la cabecera de cada sección.

  Hay que destacar que por simplicidad se decidió que cada vez que se configurara un elemento, ya sea un indicador o una medida, de una sección se realizaría una llamada al *back-end* para modificar los datos en base de datos. La otra opción fue la inclusión de un botón editar o un botón guardar pero se creyó conveniente que cada operación fuera atómica para no perder datos ante un despiste mientras se editaba una sección.

  El servicio del configurador de secciones tiene dos llamadas HTTP más que los otros maestros, esto es así debido a que cuando se utilizan los botones flechas para modificar el

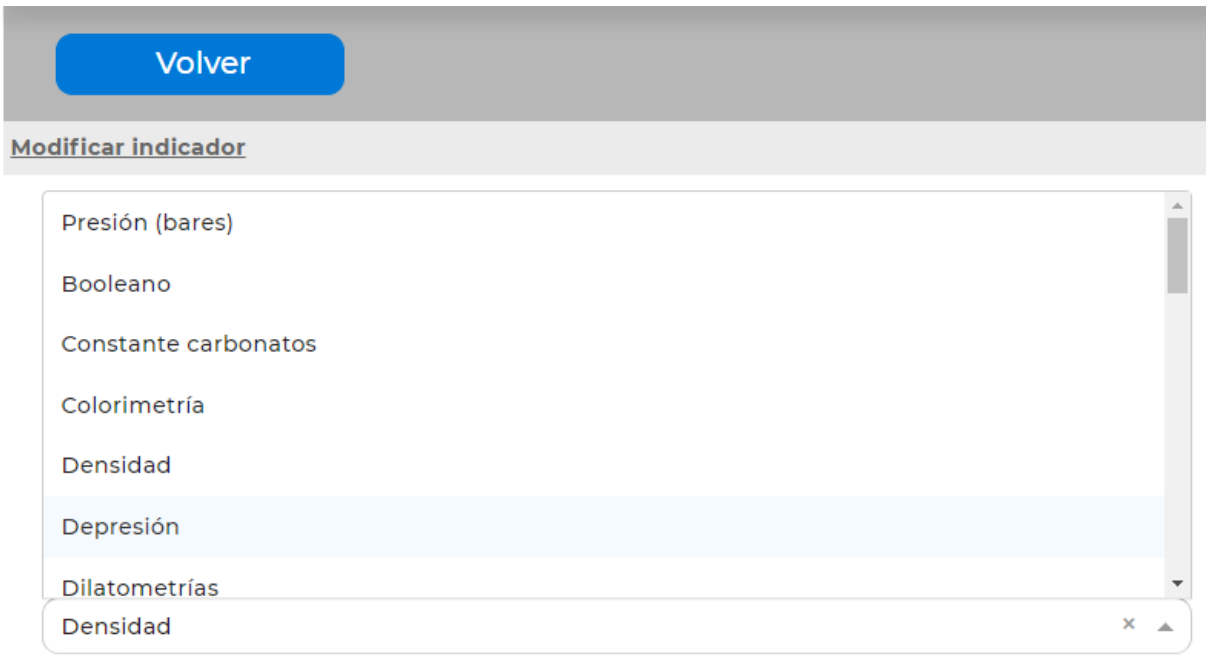

Figura 4.9: Captura del desplegable del maestro de indicadores en la aplicación web.

orden se añade una medida con el menú desplegable o se elimina una medida se envía la lista de medidas modificadas y el identificador de la sección a modificar. El otro método se encarga de eliminar un indicador concreto de la sección, en este caso solo pasamos el identificador y el servidor se encargará de eliminarlo. Estos dos métodos los podemos observar en el listing 4.17.

```
modificarMedidas( medidas:  Array<MedidaModel>, idSeccionIndicador: number):
Observable<Array<MedidaModel>>{
        const params = {
        medidas: medidas,
        idSeccionIndicador: idSeccionIndicador
        };
        return this.httpClient.post(
        this.urlBase + '/modificarMedidas',  params)
        .pipe(
                map((body: any) => body.map((dato: MedidaModel) =>
                new MedidaModel().deserialize(dato)))
        );
}

eliminarIndicador(idSeccionIndicador: number):
Observable<SeccionModel>{
        return this.httpClient.post(
        this.urlBase + '/eliminarIndicador', idSeccionIndicador)
        .pipe(
                map((body: any) => body.map((dato: MedidaModel) =>
                new MedidaModel().deserialize(dato)))
        );

}
```

Listing 4.17: Métodos del configurador de secciones en el *front-end*.

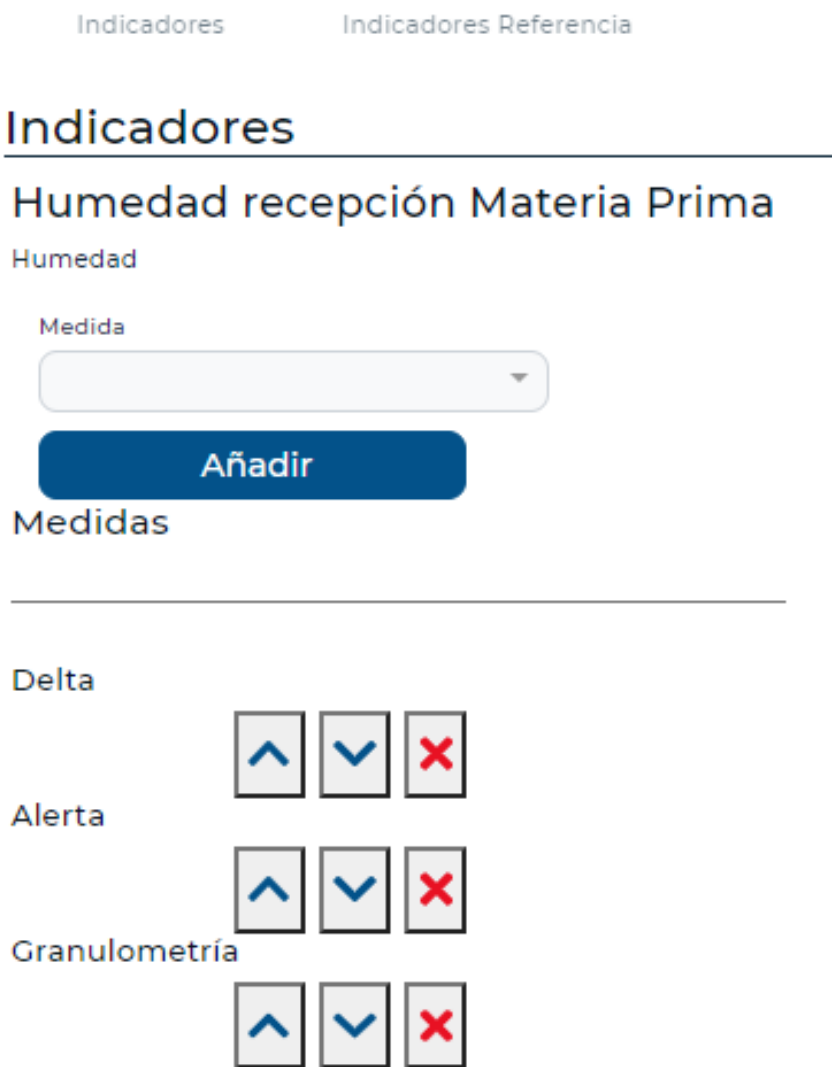

Figura 4.10: Captura del configurador de secciones en la aplicación web.

Por otro lado en el fichero HTML se utilizaron dos bucles *for* anidados, de esta manera se listan los indicadores de la sección y dentro de ellos sus medidas. Además si un indicador no contiene medidas la cabecera del listado de las mismas no aparece visible. Para los botones se utilizaron iconos del módulo común del proyecto, ya que estos se usan habitualmente para estos propósitos en otros planes de la empresa.

4.2. Verificación y validación

En este apartado se presentarán las diferentes revisiones y validaciones que se usaron para comprobar el correcto funcionamiento del código implementado.

Debido a la forma de trabajo de la empresa Integra Consultores no se realizan test al acabar módulos o partes de código, si no que el propio supervisor del proyecto comprueba el

Figura 4.11: Captura de la cabecera del configurador de secciones en la aplicación web.

funcionamiento del código desarrollado. Aún así, si se encuentran defectos en algún módulo estos se comunican al desarrollador que lo realizó para que solucione el problema o la deficiencia.

Cabe destacar que para probar todo lo desarrollado en el back-end se utilizó la herramienta web llamada swagger. Este *software* de código abierto ayuda a realizar pruebas de los servicios web RESTful. De esta manera se puede visualizar e interactuar con los recursos de la API sin tener implementada la parte cliente, lo que facilita en gran medida el desarrollo de código de la parte servidor. Para introducir parámetros en esta herramienta se utiliza el formato JSON. De esta manera le podemos dar datos de entrada que el servidor procesa y observar si la salida es correcta.

Por último, en el módulo *front-end* se fueron probando las vistas y su funcionalidad en un navegador web, para ello se observaba si la salida por pantalla era la deseada y si la funcionalidad era correcta. Para analizar los distintos elementos o realizar depuración de código se inspeccionaron las distintas páginas en el propio navegador.

Capítulo 5

Conclusiones

En este último capítulo se presentan las conclusiones a las que se ha llegado después de la implementación de esta aplicación web. Para ello se ha optado por un división en tres apartados. En el primero de ellos se hablará sobre los conocimientos adquiridos, la experiencia y el aprendizaje recibo durante la realización del proyecto, en el segundo se hablará de la experiencia adquirida en un entorno de trabajo y la tercera se expondrá todo aquello que se experimentó personalmente durante estas prácticas curriculares en una empresa.

5.1. Ámbito formativo

Durante la realización de esta aplicación web pude poner a prueba los conocimientos adquiridos durante la carrera sobretodo en la primer mitad de las prácticas, puesto que supuso todo el desarrollo del *back-end* en Java, lenguaje que ya conocía y tenía experiencia con él anteriormente. El uso de la herramienta *Swagger* supuso una gran ayuda durante el desarrollo del código pues cada parte podía ser probada inmediatamente y se solventaban los problemas rápidamente.

Durante la parte del *front-end* el desarrollo de código costó más debido a que se utilizaron tecnologías en las que hasta el momento no tenía experiencia con ellas. Aún así, gracias a diversos manuales y a la ayuda del supervisor del proyecto, se consiguió el desarrollo esperado en este módulo.

5.2. Ámbito profesional

En cuanto la experiencia profesional en la empresa valoro de forma muy positiva todo lo aprendido. Esto ha sido así gracias a las características de la empresa, pequeña y con pocos trabajadores dedicados al desarrollo de código. Por ello se puede aprender de los compañeros e interactuar con ellos de manera habitual ante algún problema o alguna duda de implementación.

Gracias a la buena comunicación dentro de la empresa y a las reuniones semanales siempre tenía claro que había que hacer en ese momento en concreto y si íbamos consiguiendo los objetivos. Aún así, he de decir que los proyectos base que utilizan tanto en el *back-end* como en el *front-end* deberían de tener documentado un manual para su comprensión, de esta forma se ahorrarían muchas consultas en los primeros días de desarrollo.

Por todo lo comentado anteriormente valoro muy positivamente el trabajo en grupo y personal realiza durante la estancia en la empresa Integra Consultores.

5.3. Ámbito personal

En el aspecto personal me he sentido muy cómodo e integrado en el ambiente de trabajo desde las primeras semanas. Los compañeros no dudaban en ayudar ante alguna dificultad o duda. Valoro también muy positivamente que en algunos aspectos me dejaran libertad de decisión sobre algunos aspectos, como pudieron ser el manejo de datos JSON o el uso de ciertas estructuras de datos en algunos de los maestros desarrollados para el *back-end*.

Para acabar me gustaría remarcar las facilidades que la empresa me facilitó en todo momento, incluso cuando debido a las circunstancias excepcionales se tuvo que optar por el teletrabajo. En este caso recibí ayuda por parte del supervisor para configurar todo lo necesario para realizar la labor de desarrollo de forma telemática.

Bibliografía

[1] Manual de Java versión 13. [Consulta: 12 de julio de 2020]
https://docs.oracle.com/en/java/javase/13/

[2] Manual del *framework* Spring. [Consulta: 27 de junio de 2020]
https://spring.io/projects/spring-boot

[3] Manual de la herramienta web Swagger. [Consulta: 19 de junio de 2020]
https://swagger.io/tools/swagger-ui/

[4] Manual de Angular. [Consulta: 17 de julio de 2020]
https://angular.io/docs

[5] Manual de MySQL. [Consulta: 19 de junio de 2020]
https://dev.mysql.com/doc/

[6] Hibernate. [Consulta: 19 de junio de 2020]
https://hibernate.org/

[7] Maven. [Consulta: 3 de junio de 2020]
https://maven.apache.org/

[8] Typescript.
https://www.typescriptlang.org/

[9] Navicat.
https://www.navicat.com/es/. [Consulta: 3 de junio de 2020]

[10] Atlassian Bitbucket [Consulta: 3 de junio de 2020]
https://www.atlassian.com/es/software/bitbucket

[11] Sourcetree. [Consulta: 3 de junio de 2020]
https://www.sourcetreeapp.com/

[12] Modelo-vista-controlador. [Consulta: 20 de julio de 2020]
https://desarrolloweb.com/articulos/que-es-mvc.html

[13] Node.js.
https://nodejs.org/es/. [Consulta: 17 de julio de 2020]

[14] Favicon. [Consulta: 17 de julio de 2020]
https://es.wikipedia.org/wiki/Favicon

[15] JSON. [Consulta: 15 de junio de 2020]
https://www.json.org/json-es.html

[16] REST. [Consulta: 19 de junio de 2020]
https://openwebinars.net/blog/que-es-rest-conoce-su-potencia/

[17] Gson. [Consulta: 3 de junio de 2020]
https://sites.google.com/site/gson/gson-user-guide