Facultad
de
Ciencias

VISOR WEB DE ARCHIVOS OFIMÁTICOS EN UN TABLONES VIRTUALES

(Web viewer of office files on virtual boards)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Manuel García de la Torre

Director: Pablo Sánchez Barreiro

Co-Director: Ángel Tezanos Ibáñez

04-09-2020

ÍNDICE CONTENIDO

INDICE ILUSTRACIONES

INDICE DE TABLAS

RESUMEN

En la actualidad, muchas empresas continúan utilizando paneles de corcho para mostrar a sus empleados diversos tipos de informes, planificaciones, gráficas o resultados empresariales, entre otros elementos. Estas corcheras contienen tanto elementos estáticos, por ejemplo, protocolos de fabricación o normas de seguridad o higiene, como elementos dinámicos, por ejemplo, turnos semanales, órdenes de producción o indicadores de productividad. Algunos de estos elementos pueden necesitar de su actualización diaria, o, incluso en algunos casos, de varias actualizaciones a lo largo del día. Para actualizar estos documentos, los responsables de mantener estos tablones deben modificar uno o más documentos ofimáticos, imprimir los documentos actualizados y sustituir los documentos viejos por los nuevos en el tablón de corcho. Esto supone un considerable gasto de papel y un notable esfuerzo para los responsables de mantenerlos al día.

Para intentar aliviar estos problemas, en este Trabajo Fin de Grado se ha desarrollado una aplicación, denominada OCDB (Office Content Display Board), para la virtualización de estas corcheras. Para llevar a cabo esta virtualización, se ha generado una aplicación web que permita crear y visualizar tablones de corcho digitales.

Palabras clave

React, corcheras, documentos ofimáticos, digitalización

ABSTRACT

Today, many companies continue using cork boards to show their employees several types of reports, business plans, graphs or results, among other elements. These cork boards contains static elements as for example, manufacturing protocols or health or safety standards, such as dynamic elements, as weekly shifts, work orders, production or productivity indicators. Some of these elements may need to be updated daily, or even in some cases, from various updates throughout the day. To actualize these documents, those responsible for maintaining these cork boards must modify one or more office documents, print the documents updated and replace old documents with new ones in the cork table. This involves a considerable waste of paper and remarkable effort for those responsible to keep them up to date.

To try to solve these problems, this Final Degree Project has developed an application, called OCDB (Office Content Display Board), the virtualization of these corks. To carry out this virtualization, a web application has been generated, which allows creating and visualize digital cork boards. The application has been developed using Java Spring framework with MySQL for the background-end, and React for the front-end part.

Keywords

React, cork boards, office documents, digitization

1. Introducción, objetivos y metodología

1.1. Introducción

Este Trabajo de Fin de Grado ha consistido en la elaboración de una aplicación para la visualización de tableros virtuales, Y a sido llevada a cabo dentro de un programa de prácticas extracurriculares en la empresa *SOINCON – (Soluciones Industriales de Conectividad*). *SOINCON* es una empresa dedicada al desarrollo software, sobre todo para la industria, la llamada la cuarta revolución industrial o Industria 4.0. El producto estrella de la compañía es una familia de aplicaciones llamada *EMI Suite 4.0* en la que encontramos aplicaciones como *Easy Gmao, Visual Tracking, 5S Digital*, entre otras. Todas estas aplicaciones estan dedicadas al aumento de la productividad, control del mantenimiento o la mejora de la calidad dentro del entorno de la fabricación industrial.

En esta empresa, realicé mis prácticas curriculares en las que me inicié, entre otras cosas en la tecnología *React* (Altadill, 2019). Tras un periodo de aprendizaje, me dediqué a realizar trabajos de mantenimiento en aplicaciones y desarrollo de componentes de *React*. Una vez finalizadas mis prácticas curriculares, se me ofreció realizar este Trabajo de Fin de Grado en un nuevo periodo de prácticas extracurriculares.

El proyecto que se ha desarrollado nace de la idea de un directivo de la empresa, concebida tras detectar un problema común a la práctica totalidad de las empresas clientes de *SOINCON*. La mayoría de estos clientes disponen de inmensas corcheras donde se cuelgan multitud de documentos. Aunque algunos de estos documentos son estáticos, como, por ejemplo, protocolos de actuación o normas de seguridad e higiene, la mayoría de ellos deben ser actualizados periódicamente, ya que pueden contener indicadores de productividad, turnos, ordenes de producción, entre otros elementos. Estos documentos pueden necesitar actualizarse semanalmente, diariamente, o incluso, en el peor de los casos, varias veces por día. Por ejemplo, en el caso de los indicadores de calidad, estos deben actualizarse cada vez que se termina una nueva tanda de producción. Este trabajo precisa de la edición de un documento digital, su impresión y sustitución en la corchera o corcheras de la empresa. En muchos casos, las corcheras están sitas cerca de las líneas de producción, que podrían estar alejadas de las oficinas donde se editan estos documentos. Por lo tanto, esta situación supone un gasto de papel y de recursos humanos, ya que muchas emplean a varias personas para mantener actualizadas las corcheras, sobre todo en empresas grandes.

Tras varias reuniones con el directivo antes mencionado, se me ofreció la posibilidad de realizar una aplicación denominada OCDB *(Office Content Display Board*) la cual se integraría dentro de la familia de productos de *Emi Suite 4.0*. Esta aplicación debía permitir crear, editar y visualizar corcheras virtuales que puedan sustituir a las corcheras tradicionales. Estas corcheras virtuales dispondrían de un modo de edición donde añadir varios elementos dinámicos que muestren diferentes documentos ofimáticos, como por ejemplo un cuadrante de turnos semanales de trabajo. Estos ítems, deberían poder

configurarse en tamaño, color, nombre o tasa de refresco pudiendo ser desde pocos minutos, hasta semanas o meses. Además, los documentos ofimáticos que se carguen en la aplicación deberían poder estar alojados en local, carpetas de red o servidores entre otros elementos. La aplicación se desarrollaría utilizando *React*, *Spring* (Cosmina, Harrop, & Schaefer, 2017), *Java y MySQL* como tecnologías principales, para así mantener la homogeneidad con *EMI Suite*.

1.2. Objetivos

De acuerdo con lo expuesto en la sección previa, el objetivo general de este Trabajo Fin de Grado es el de crear una aplicación para mostrar corcheras virtuales en dispositivos como, por ejemplo, pantallas murales, ordenadores o proyectores. Además, obviamente, la aplicación debe permitir crear y editar estas corcheras, añadiendo diferentes documentos e imágenes, simulando así el funcionamiento de las corcheras convencionales. Este objetivo principal puede descomponerse en los siguientes subobjetivos:

- Admitir distintos archivos ofimáticos e imágenes.
- Disponer de un modo visualización en el que no se pueda editar el tablero.
- Compartir el tablero y visualizarlo en diferentes dispositivos, como pantallas o proyectores.
- Permitir la actualización periódica de los diferentes documentos añadidos a un tablero.

Tal como se comentó en la introducción, la finalidad última de esta aplicación sería ayudar a reducir costes en una empresa, mediante la virtualización de sus corcheras, provocando esto un ahorro anual considerable en papel y recursos humanos. En otras palabras, estos objetivos deberían ayudar a que mejore la productividad de la empresa, al liberarse parte de sus recursos, que podrán emplear ahora el tiempo ahorrado en actividades más productivas.

Respecto a la empresa *SOINCON*, se espera que la incorporación de esta nueva aplicación a su familia de soluciones para la Industria 4.0 le ayude a destacar sobre otros productos de la competencia, potenciando así, poder llegar a nuevos clientes.

1.3. Metodología

La metodología utilizada para desarrollar este proyecto ha sido *SCRUM* (Sutherland et al., 2018), un tipo de metodología ágil para desarrollo de software. De acuerdo con esta metodología, el desarrollo de un producto software se divide en un conjunto de iteraciones denominadas *sprints*.

En mi caso concreto, los sprints tuvieron una duración de 1 semana, siendo cada jueves cuando me reunía con el *Product Owner*, que, en este caso, era el directivo de la empresa que había detectado una necesidad en los clientes y deseaba crear una herramienta innovadora para satisfacerla. Los jueves, después de cada revisión del producto, se decidía qué funcionalidades deseaba ver implementada en la siguiente reunión.

El Trabajo de Fin de Grado tuvo una duración aproximada de 12 semanas, realizadas de forma presencial en la empresa, donde se tomaba la temperatura con cámara térmica al entrar a todos los trabajadores y era obligatorio el lavado de manos con hidrogel de forma continua y el uso de mascarilla, la cual se nos proporcionaba cuando era necesario. Los sprints para el desarrollo del producto tuvieron una duración de 9 semanas. Anterior a las 9 semanas correspondientes a los sprints, se concertaron varias reuniones sobre la elaboración y preparación del proyecto. Posteriormente se estuvieron retocando detalles finales que no precisaban de un sprint.

De acuerdo con la metodología Scrum, al comienzo de cada sprint seleccionaba las funcionalidades a implementar con el *Product Owner*. Una vez elegida las funcionalidades a implementar, las analizaba para saber cómo afectaban a la implementación de la aplicación en ese momento, las diseñaba, implementaba y, por último, realizaba pruebas para comprobar su correcto funcionamiento. Una vez llegaba a este punto, el *Product Owner* me proporcionaba *feedback*, el cual podía traducirse en una serie de modificaciones a implementar en futuros sprints. Tras añadir estas modificaciones a las funcionalidades a implementar, empezaba una nueva iteración.

Para gestionar los elementos generados por la metodología *SCRUM* hemos utilizado la herramienta *ClickUp*[1]. Esta herramienta, nos permite definir *sprints*, *Backlog*, tareas, gestionar las tareas en un tablero *Kanban*. Este tablero *Kanban* se descomponía con cuatro columnas: (1) *por hacer*, (2) *en desarrollo*, (3) *en revisión* y (4) *finalizado*. Un ejemplo de uso de *ClickUp* en nuestro proyecto se proporciona en la *Ilustración 1*, la cual muestra los elementos a implementar en el sprint 5. Estos elementos eran, tal como se puede ver, una nueva funcionalidad y dos modificaciones a otras dos funcionalidades previamente existentes.

[1] https://clickup.com/

La funcionalidad que se agregó fue la de *Cargar los documentos desde remoto.* Las que se modificaron fueron:

1. *Cargar documento desde local*: se solucionaba cambiando los dos botones que había, uno para cargar y otro para añadir el documento, por uno solo que hiciese la misma funcionalidad que dos.
2. *Actualización del documento periódicamente:* se pedía cambiar el formulario, para recibir minutos en lugar de segundos, y en el caso de que no se introdujese nada, se tomará como que era un documento estático.

Ilustración 1. Tareas del sprint 5.

1.4. Infraestructura tecnológica

En esta aplicación hemos utilizado diversas tecnologías y/o herramientas, dependiendo de la parte del proyecto que sea.

1.4.1. Capa de Persistencia

En esta capa hemos utilizado herramientas como:

- *MySQL* para la base de datos, que es uno de los sistemas de gestión de base de datos relacional más famoso del mundo y el más utilizado de código abierto. Es utilizado por plataformas tan conocidas como *Google*, *WordPress* o *Twitter*.
- *DBeaver*, como sistema de administración de la base de datos.
- *JPA* (*Java Persistence API*) para la construcción de un puente objetivo-relacional para un conjunto de clases *Java*.
- *Maven*, herramienta para la gestión de proyectos y dependencias *Java*.

1.4.2. Capa de Negocio

En esta capa se han utilizado herramientas tales como:

- *Spring* (Keith, Schincariol, & Nardone, 2018), un framework de código abierto para el desarrollo de aplicaciones empresariales ampliamente utilizado en Java.
- Una librería, contenida dentro del paquete ofimático *LibreOffice*, para la transformación de documentos ofimáticos en archivos PDF.

1.4.3. Capa de Presentación

En esta capa se han utilizado estas herramientas:

- *ReactJS* (Altadill, 2019) es una librería de *JavaScript* de código abierto para el desarrollo de front-ends (Interfaz de usuario) desarrollada por *Facebook*. Empresas como *Facebook*, *AirBnB* o *Netflix* utilizan *ReactJS* o algún derivado de este, como *React Native*, para desarrollar sus productos.

- *PDF.js*, una librería de JavaScript para la gestión y visualización de PDFs en navegador web.
- Sass (Giraudel, 2016), un lenguaje de hojas de estilo.
- *Yarn* (Yarn, s.f), herramienta para la gestión de paquetes de JavaScrith

1.4.4. Herramientas de Desarrollo

Para el control de la calidad, solo se ha controlado de la parte front-end, a través de un componente perteneciente a *SOINCON* para esta finalidad. Este componente mostraba mostrando advertencias a la hora de compilar el código en el caso de que no se cumpliesen los criterios que se le hubiese proporcionado a esa herramienta.

Cabe destacar que no se ha utilizado integración continua en el desarrollo de la aplicación.

Para la gestión de versiones se ha utilizado *Git* y *GitLab*. *Git* es un sistema moderno de gestión de versiones distribuido y que puede considerarse en la actualidad como el estándar de facto para la gestión de versiones.

En el caso de esta aplicación, disponía de dos repositorios, uno de la parte *front-end* y otro de la parte *back-end*. El repositorio de *front-end* tenía dos ramas, una rama *master*, donde se publicaban las versiones después de cada sprint, otra rama *develop*, donde trabajaba en el desarrollo de la aplicación. En algunos casos se crearon ramas alternativas para probar ideas de implementación. Por otra parte, el repositorio de *back-end*, tenía tres ramas, dos eran la *master* y *develop*, y la otra era una rama llamada *v-alternative* donde desarrollé una prueba de *back-end* en *Node* que al final no se utilizó en la aplicación. *GitLab* es un servicio de alojamiento de repositorios *Git* en la nube donde teníamos copias de los dos repositorios.

Por último, se han utilizado otras herramientas como *Eclipse*, *Visual Studio Code* o *Postman*[2], una herramienta que nos proporciona funcionalidades para probar *APIs*.

2. Contexto y Requisitos

2.1. Análisis del contexto de la aplicación

Para poder entender mejor el proyecto desarrollado en este Trabajo Fin de Grado, vamos a poner en contexto a la empresa para la que se ha realizado el producto. *SOINCON*

[2] https://www.postman.com/

es una empresa que se localiza en el centro de empresas de Guarnizo. Dispone de dos oficinas: una de consultoría donde se encuentran los temas más administrativos de la empresa, y otra es *Software Factory* donde se encuentra el equipo de producción de software. *SOINCON*, tal como se ha comentado, se dedica al desarrollo profesional de software para todo tipo de empresas, aunque actualmente, se centra sobre todo en la Industria 4.0.

SOINCON ha desarrollado diversas aplicaciones específicas para diferentes clientes, pero sobre todo se dedica al desarrollo y comercialización de su producto estrella, llamado *EMI Suite 4.0*, una familia de aplicaciones principalmente enfocada a la industria. En *EMI Suite 4.0* encontramos aplicaciones como:

- *Visual Tracking* para el seguimiento de la producción de una determinada fabrica en la que se gestiona las órdenes de trabajo, los puestos de trabajo, la calidad del producto, entre otras funcionalidades.
- *EasyGmao*, que es básicamente una aplicación de Gestión de Mantenimiento Asistido por Ordenador (GMAO) y que se puede utilizar de forma sencilla para gestionar el mantenimiento de la maquinaria de una empresa.
- A parte de esto, encontramos otras aplicaciones como *5 salas, Digital People o Machine Connect*, cada una para una tarea concreta diferente dentro del proceso global de gestión de una fábrica.

Además, *SOINCON* dispone de módulos pequeños dentro de *EMI Suite*, que aportan soluciones software a problemas que se han detectado en la mayoría de las empresas, pero que no han sido pedidas explícitamente por ningún cliente. Estos módulos, suelen ser añadidos como extras a cualquier aplicación principal adquirida por una nueva empresa. En nuestro caso, la aplicación que vamos a desarrollar se encuentra dentro de este tipo de módulos extra

2.2. Captura de Requisitos y Requisitos Funcionales

Dentro del ámbito de este Trabajo de Fin de Grado no se ha desarrollado ningún proceso de captura de requisitos, ya que sus requisitos ya habían sido definidos de manera informal por el directivo de la empresa que había concebido la idea de desarrollar esta aplicación. Por tanto, la única labor realizada durante el desarrollo de este Trabajo Fin de Grado ha sido la de definir estos requisitos de manera más explícita mediante tarjetas de funciones a implementar. Esta lista de requisitos inicial puede verse en la *Tabla 1*.

En esta aplicación, se identifican dos actores principales:

1. *Usuario anónimo*: Este actor puede ser cualquier persona que visualice un tablero publicado.

2. *Administrador/Editor*: Es el encargado de gestionar los tableros, es decir, el responsable de crear los tableros, añadir los documentos necesarios, su tiempo de refresco entre otras cuestiones.

Identificador	**Descripción**
R001	El usuario anónimo desea ver el tablero sin poder modificar nada.
R002	El Administrador/Editor debe poder crear/eliminar uno o varios tableros.
R003	El Administrador/Editor debe poder editar un tablero.
R004	El Administrador/Editor debe añadir ficheros desde su ordenador.
R005	El Administrador/Editor debe añadir ficheros desde un equipo/servidor remoto.
R006	El Administrador/Editor debe poder cambiar de color y nombre a los ítems del tablero que contienen los ficheros.
R007	El Administrador/Editor debe poder borrar los ficheros añadidos.
R008	El Administrador/Editor desea poder hacer que un fichero se actualice dinámicamente desde su archivo origen cada cierto tiempo.
R009	El Administrador/Editor desea publicar el tablero de forma que sea visible el contenido de los ficheros sin que se puedan editar o eliminar.
R0010	El Administrador/Editor debe poder guardar los tableros creados.

Tabla 1. Requisitos funcionales.

Para poner por escrito los requisitos de la aplicación, se mantuvieron un par de reuniones con la persona que había ideado la aplicación y donde se me explicaron las funcionalidades básicas que debían tener. Además, se me proporcionaron unos *mockups* hechos a mano para que me hiciera una idea de cómo querían que fuese la aplicación. En estas reuniones iniciales. También propuse algunas ideas de cambio y expliqué como abordaría desde mi punto de vista alguna funcionalidad. Tras debatirlo con compañeros presentes en las reuniones, decidimos cuál iba a ser la estrategia a seguir. A continuación, añadí al *Backlog* de *Cicklup* las tareas correspondientes a las funciones que se me habían ido contando y otorgué una prioridad conforme a su importancia como observamos en la *Ilustración 3*. La *Ilustración 2* muestra, a modo de ejemplo, una de estas funciones con su descripción.

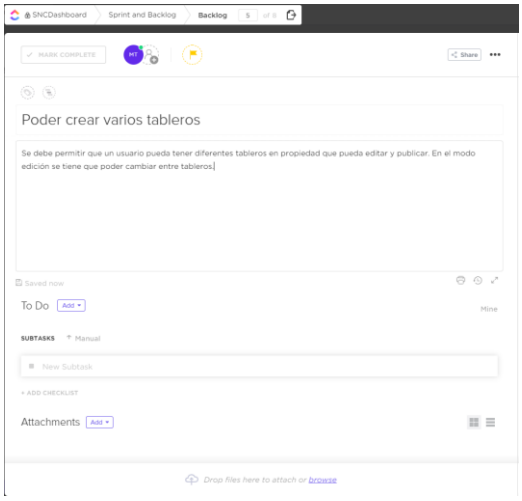

Ilustración 2. Tarea del Backlog.

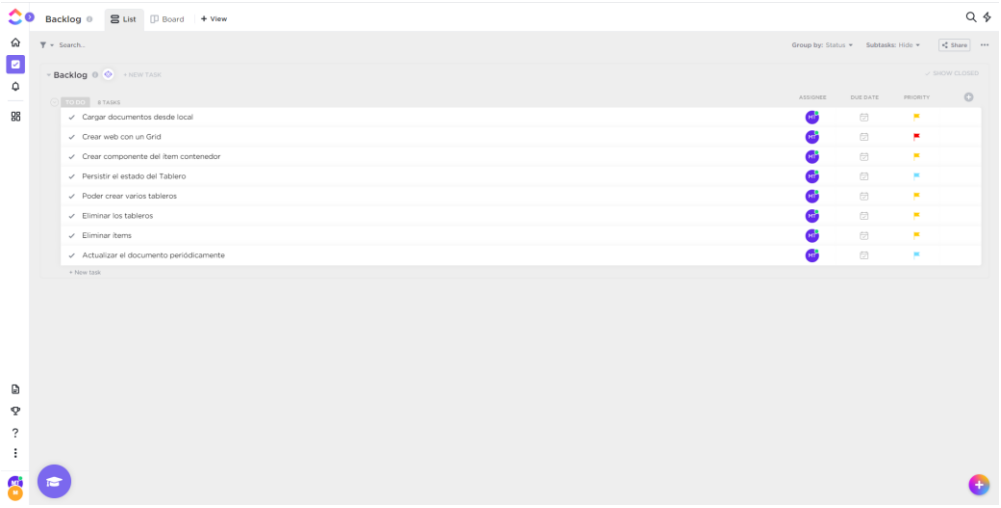

Ilustración 3. Backlog Inicial.

Después de esto, se acordó de que cada jueves se iba a revisar lo que se había realizado y cómo avanzaba el proyecto, para así proponer ideas de mejora o nuevas funcionalidades que pudiesen ir surgiendo conforme avanzaba el desarrollo de la aplicación. En este caso, como la última reunión se produjo un lunes, procedí a preparar todo el material necesario para empezar el primer *sprint* el siguiente jueves.

Para ello creé un tablero *Kanban* (descrito en la sección de metodología) para todos los *sprints*. Posteriormente, cada jueves se revisaba con el directivo y algunos compañeros lo que había realizado y se proponían ideas, tanto nuevas como de mejora, y se añadían al *Backlog*. Después debatíamos qué tareas iba a realizar en el siguiente *sprint*.

2.3. Requisitos no funcionales

Con relación a los requisitos no funcionales, se identificaron las siguientes necesidades. En primer lugar, la aplicación debía seguir unas líneas de diseño moderna, semejante a la del resto de los productos de la compañía y utilizar un conjunto de colores corporativos bien definidos.

Además, la aplicación debía poder ser utilizada y visualizada desde cualquier sistema operativo. Debido a que *SOINCON* dispone de clientes que tienen sedes en diferentes países, la aplicación debía poder traducirse de manera sencilla a diferentes idiomas.

Por otro lado, la aplicación debe mantener los archivos seguros, es decir, sin que sean accesibles desde fuera de la red de la empresa, y no debe permitir editar los tableros a personas que no deban hacerlo. Por eso la aplicación solo debe funcionar dentro de la red local de una compañía y solo lo podrán editar las personas que estén dados de alta en un servicio de seguridad que gestiona los permisos de las aplicaciones corporativas. Este servicio dispone de roles y permisos asociados a cada rol. En este caso disponemos de dos roles, un rol de visitante y otro de administrador, correspondiente a los dos actores de la aplicación. Al rol de administrador se le conceden todos los permisos, mientras que al rol de visitante solo puede ver los tableros sin posibilidad de visualizar ninguna labor de edición.

Finalmente, la aplicación cumple por defecto diferentes normativas y regulaciones actuales, ya que la aplicación no gestiona ella misma nada que no esté regulado por la ley. Por ejemplo, la gestión de usuarios es realizada por una aplicación de seguridad paralela a esta aplicación, por lo cual no tiene acceso a datos personales. Esta aplicación recibe una serie de permisos que se los proporciona el servicio de seguridad externo dependiendo del usuario que esté conectado, y a través de eso, gestiona lo que cada usuario puede ver o realizar en la web.

3. Arquitectura, diseño e implementación

3.1. Diseño arquitectónico

La aplicación desarrollada puede considerarse como una aplicación web cuya apariencia emula o evoca un tablón de corcho y cuyos datos han de ser actualizados por un administrador desde una especie de *backoffice*. Por tanto, la aplicación puede considerarse como una aplicación web empresarial y se desarrollará siguiendo el estilo arquitectónico habitual para este tipo de aplicaciones, que es el de una arquitectura en tres capas (Fowler, 2002) como la que vemos en *Ilustración 4*.

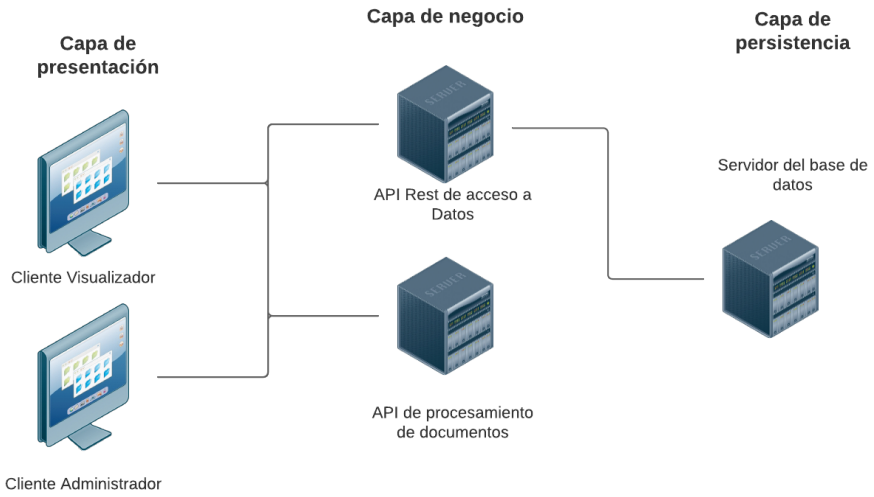

Ilustración 4. Diagrama modelo de tres capas.

La capa de presentación se desarrolló utilizando *React JS* (Altadill, 2019), ya que este es el framework de *JavaScript* actualmente utilizado por la empresa donde realicé este proyecto.

Para la comunicación entre la capa de presentación y de negocio, se creó una API REST, realizando la comunicación entre cliente y servidor mediante llamadas AJAX. Para ala gestión de estas llamadas AJAX, se utilizó la librería de *ReactJS Axios*.

La capa de negocio necesitaba llevar a cabo dos tipos de acciones distintas:

1. Localizar e importar documentos ofimáticos que debían ser convertidos en PDFs para su visualización en las corcheras virtuales.
2. Acceder a la base de datos para recuperar o actualizar información sobre las corcheras.

La parte de localización, importación y conversión de documentos está gestionada por un servicio propio, mientras que la parte de acceso a datos disponía de otro servicio distinto. Estos dos servicios de la capa de negoció están desarrollados a través del framework *Spring* en *Java*.

Para finalizar, en la capa de persistencia, se definió el modelo de la base de datos a través de *Java Persistence API* (Keith, Schincariol, & Nardone, 2018), sobre el gestor de base de datos relacional *MySQL*.

3.2. Diseño e implementación de la capa de persistencia

3.2.1. Diseño

El primer paso para desarrollar la capa de persistencia fue crear un diagrama de clases en UML de la información que era necesario persistir. Dicho diagrama lo podemos ver en la *Ilustración 5*.

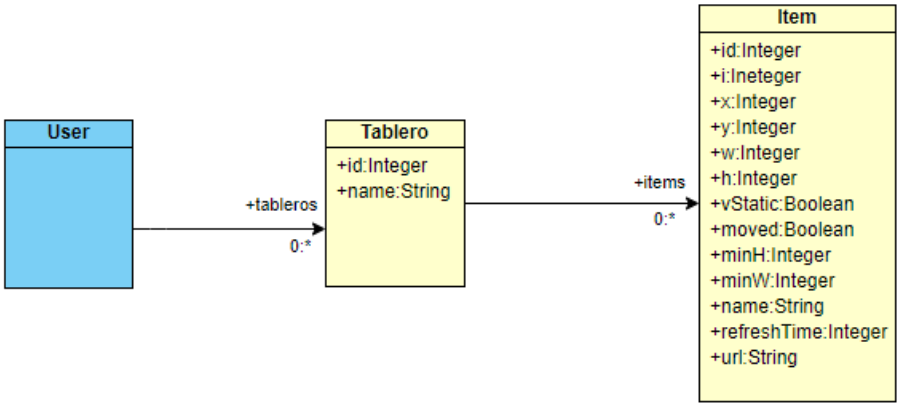

Ilustración 5. Diagrama de clases.

Como observamos en la *Ilustración 5* , tenemos un diagrama bastante sencillo en el cual un usuario tiene una serie de tableros asociados a él, y en estos tableros tenemos los ítems con los datos necesarios para su representación. La clase *User* es una clase externa a la aplicación que está dentro del *snc-core*, un módulo del framework base utilizado por *SOINCON*, la empresa donde realicé este trabajo, para el desarrollo de todos sus proyectos. En este *Core* se gestiona todo lo relativo a los usuarios y sus permisos, entre otras cosas. En el caso de que un cliente quisiese la aplicación, en el *core* que se le va a instalar dispone de la clase usuarios con la lista de tableros asociadas.

Respecto a la clase tablero, solo disponemos de tres atributos: (1) el *id* del tablero; (2) el nombre que se le ha dado al tablero; y, (3) una lista de ítems que pueden ser 0 o infinitos ítems añadidos.

Por último, la clase *Item*, dispone de : (1) un identificador (*id*) único que va a tener el ítem en la base de datos, (2) un atributo *i* que representa el identificador dentro de cada tablero; (3) los atributos *x*, *y*, *w* y *h* que guardan la posición y tamaño de cada ítem en la rejilla que representa el tablero; (4) los atributos *vStatic* y *moved* guardan si se puede mover el ítem y si se ha movido desde su creación. Estos últimos atributos en principio no tienen un gran uso, pero son necesarios para poder a volver a representar los ítems sobre el tablero.

En la *Ilustración 5* también vemos que la clase *Item* dispone de los atributos *name* y *color* que guardan el nombre y el color visible de cada ítem, respectivamente. Para terminar, tenemos el atributo *refreshTime*, que puede ser nulo y guarda el tiempo de refresco en segundos del ítem, en el caso de que sea necesario; y el atributo *url*, que guarda la ruta de donde se ha obtenido el documento a mostrar.

3.2.2. Implementación

Una vez estaba diseñado el modelo, procedimos a su implementación. Para la implementación de la capa de persistencia *SOINCON* utiliza siempre JPA (*Java Persistence API*) (Keith, Schincariol, & Nardone, 2018). Por tanto, una vez traducidas las clases de nuestro modelo de dominio en UML a Java, debíamos marcar dichas clases, o POJOs (*Plain Old Java Objects*), con anotaciones JPA para generar la implementación de la base de datos.

```
1.  @Entity
2.  @AllArgsConstructor
3.  @NoArgsConstructor
4.  public class Item extends AbstractDefaultEntity<Item>{
5.      private static final long serialVersionUID = 1L;
6.
7.      @Id
8.      @GeneratedValue(strategy = GenerationType.IDENTITY)
9.      private Integer id;
10.     @Column(nullable = false)
11.     private Integer i;
12.     @Column(nullable = false)
13.     private Integer x;
14.     @Column(nullable = false)
15.     private Integer y;
16.     @Column(nullable = false)
17.     private Integer w;
18.     @Column(nullable = false)
19.     private Integer h;
20.     @Column(nullable = false)
21.     private Boolean vstatic;
22.     @Column(nullable = false)
23.     private Boolean moved;
24.     @Column(nullable = false)
25.     private Integer minH;
26.     @Column(nullable = false)
27.     private Integer minW;
28.     @Column(nullable = false)
29.     private String name;
30.     @Column(nullable = false)
31.     private String url;
32.     @Column(nullable = false)
33.     private String color;
34.     private Integer refreshTime;
35.     ...
```

Ilustración 6. Clase *Item* anotada con JPA.

A modo de ejemplo, la *Ilustración* 6 muestra la clase *Item* anotada con *JPA*. En primer lugar, indicamos que la clase va a ser una entidad (línea 1), es decir, una tabla de la base de datos con identificador definido en la propia clase. A continuación, indicamos, mediante la anotación *@Id* (línea 7) que dicho identificador será el atributo *id*, y que *además* el valor de ese atributo será autogenerado, mediante la anotación *@GeneratedValue* (línea 8). Para la generación de este valor se utilizará una columna con valores auto incrementadlos en la tabla de la base de datos (*GenerationType.Identity*). El resto de los atributos son marcados como columnas y no pueden ser nulos excepto *refreshTime*.

En la otra clase, la del tablero, se dispone de otras anotaciones diferentes como *@OneToMany* en la lista contenedora de los ítems.

Una vez generadas la base de datos, para su manipulación hicimos uso del patrón *repositorio*. Para la creación de estos repositorios, por cada clase de nuestro modelo de dominio, extendíamos la interfaz de repositorio genérica que se muestra en la *Ilustración* 7.

```
1.    @NoRepositoryBean
2.    public interface IGenericDao<T, ID extends Serializable, F extends AbstractFilter> extends Serializable
      {
3.
4.        T findOne(ID id);
5.
6.        List<T> findAll();
7.
8.        T create(T entity);
9.
10.       T update(T entity);
11.
12.       T save(T entity, ID id);
13.
14.       void delete(T entity);
15.
16.       void deleteById(ID id);
17.
18.       void detach(T entity);
19.
20.       void refresh(T entity);
21.
22.       void flush();
23.
24.       List<T> findAllByFilter(F filter);
25.   }
26.
```

Ilustración 7. DAO genérico.

La implementación concreta de cada repositorio se realizó con ayuda de un código genérico ya existente en la empresa, desarrollado por un compañero. Por tanto, mi tarea en este aspecto se limitó a la correcta definición de las interfaces de acceso a datos para cada clase del modelo de dominio.

3.3. Diseño e implementación capa de negocio

3.3.1. Diseño

En esta aplicación, la capa de negocio se encarga de dos funciones diferentes: (1) conectar los diferentes *end-points*, como los de creación, borrado o editado de las entidades, con sus respectivos métodos en los repositorios; y, (2) buscar los documentos a visualizar, cargarlos, transformarlos a PDF y enviarlos a la capa de presentación. Ambas funcionalidades se han desarrollado en Java a través del framework *Spring* (Cosmina, Harrop, & Schaefer, 2017).

Respeto a la funcionalidad de gestión de las entidades de la base de datos, disponemos en los controladores de los dos recursos (*Item* y *Tablero*) diferentes llamadas, como POST, PUT, GET o DELETE, encargadas de gestionar el acceso a los datos. Por otro lado, la funcionalidad de la capa de negocio relativa a la gestión de los documentos y su procesamiento dispone de 4 *end-points* (3 POST y un GET). Para poner un ejemplo, uno de estos métodos POST es el encargado de buscar un archivo desde URL y llevarlo al servidor para su posterior tratamiento. En este caso, la llamada se realiza sobre la "/uploadFileUrl" y se le pasa un *RequestParam* con el URL del recurso que debe traer al servidor.

3.3.2. Implementación

En primer lugar, se implementó la parte de acceso a datos. Para ello, se crearon dos controladores REST, uno para la clase tablero y otro para la clase *Item*, los cuales fueron desarrollados en Java a través del framework *Spring*.

```
1.  @RestController
2.  @RequestMapping(value = "/Items")
3.  @CrossOrigin("*")
4.  public class ItemsController {
5.
6.      @Autowired
7.      private IItemsService itemsService;
8.
9.  ...
```

Ilustración 8. Cabecera controlador de ítems.

La *Ilustración 8* muestra la cabecera del controlador para la entidad *Item*, donde vemos que tiene diferentes anotaciones. Mediante la anotación *@RestController* (línea 1), indicamos que esa clase va a ser un controlador REST. A continuación, en la línea 2 indicamos a través de la anotación *@RequestMapping* cuál va a ser la URI base de acceso a ese controlador. Seguidamente, en la línea 3, la anotación *@CrossOrigin*, especifica que este controlador pueda ser accedido desde un dominio diferente al que esta desplegado, es decir, de un origen diferente.

Por último, la anotación *@Autowired* (línea 6), realiza la inyección de dependencias del servicio que gestiona el repositorio de ítems.

Dentro de este mismo controlador, tenemos diferentes *end-points*. A modo de ejemplo la *Ilustración 8* muestra el *end-point* correspondiente del método GET aplicado a un ítem concreto.

```
1.    @GetMapping(value = "/{id}")
2.       public ResponseEntity<ItemsDto> findOne(@PathVariable Integer id) {
3.
4.          ResponseEntity<ItemsDto> response;
5.          if (id == null) {
6.             response = ResponseEntity.badRequest().build();
7.             return response;
8.          }
9.          Documento entity = itemsService.get(id);
10.
11.         if (entity == null) {
12.
13.            response = ResponseEntity.notFound().build();
14.
15.         } else {
16.
17.            response = ResponseEntity.ok(ItemsDto.convertToDto(entity));
18.         }
19.
20.         return response;
21.      }
```

Ilustración 8. End-point correspondiente al GET de un ítem.

Este método, denominado *findOne*, está anotado como *@GetMapping* (línea 1) lo que indica que este método responde a una petición HTML correspondiente a la petición GET. Además, el parámetro entre paréntesis indica que la petición debe hacerse sobre la URI base del controlador (*/Items*), seguida de un identificador, el cual se corresponde, como es de esperar, con el identificado del ítem a buscar. En la cabecera del método (línea 2) observamos que este devuelve un *ResponseEntity* de un DTO *Item*. El método define un parámetro, el cual se extrae de la URL sobre la cual se realiza la petición, tal como indica la anotación *@PathVariable*.

El cuerpo del método es bastante sencillo y fácil de comprender. En primer lugar, se declara la respuesta, después se comprueba si el *id* era nulo del recurso a buscar. En tal caso, se devuelve el error que indica que se ha realizado mal la petición. En el caso de que el *id* no sea nulo, se declara y busca el ítem en la base de datos a través del servicio. Si la entidad del ítem buscado es nula quiere decir que no existe y se devuelve ese error. Por último, si existe, se convierte el ítem a DTO y se devuelve.

Por otra parte, disponemos de la API encargada del procesamiento de los documentos. Esta API contiene un único controlador con 4 *end-points*: dos encargados de cargar un documento a visualizar desde local o remoto en el servidor. Otro encargado de procesar el documento a PDF, que es el que podemos ver en la *Ilustración 9*; y, por último, el encargado de enviar los PDFs a la capa de presentación.

```
1.    @PostMapping("/pdfcreate/{fileName}")
2.       public ResponseEntity<Void> createPDF(@PathVariable String fileName) {
3.          ResponseEntity<Void> response;
4.          if (fileName == null) {
5.             response = ResponseEntity.badRequest().build();
6.          } else {
7.             Process p;
8.             try {
9.                p = Runtime.getRuntime().exec("soffice --headless --convert-to      pdf  /home/snc/dashboard-
    files"
10.                      + fileName + " --outdir /home/snc/dashboard-files/renderFiles");
11.               p.waitFor();
12.               p.destroy();
13.               response = ResponseEntity.ok().build();
14.            } catch (Exception e) {
15.               response = ResponseEntity.badRequest().build();
16.            }
17.
18.         }
19.         return response;
20.      }
```

Ilustración 9. End-point transformación a PDF.

El *end-point* de la *Ilustración 9* se corresponde a una petición de tipo POST (línea 1) que transforma el archivo cuyo nombre se indica en la URL a PDF. Este archivo debe haber sido cargado anteriormente en el servidor. Este método, devuelve una respuesta sin datos, que puede ser: (1) un código de error por petición incorrecta, si se ha pasado un nombre de archivo nulo; (2) un código de error interno si se ha producido un error durante la conversión del archivo; o, (3) un código de todo correcto.

El funcionamiento de este método se basa en un proceso que se crea en tiempo de ejecución y que lanza un comando de consola que ejecuta un módulo de *LibreOffice* para transformar documentos a pdfs. Este módulo coge el documento cuya ruta y nombre se le pasan por parámetros y lo coloca en otra carpeta donde están los documentos ya transformados para que el *end-point* que envía los datos disponga del documento procesado. Este método espera hasta que se haya terminado de procesar el documento y luego destruye el proceso creado para ello, de manera que hasta que no esté finalizado el proceso, no se retorna nada.

3.4. Diseño e implementación capa de presentación

3.4.1. Diseño

Para el desarrollo de la capa de presentación hemos utilizado *React JS*. *React JS* es un framework de desarrollo de JavaScript que proporciona una gran versatilidad a la hora de crear clientes webs, los cuales pueden ser utilizados y personalizados de manera que se adapten lo mejor posible al problema que se desea resolver. Además, *React JS* dispone de

una gran variedad de componentes que pueden ser añadidos utilizando el gestor de dependencias *Yarn* (Yarn, 2020).

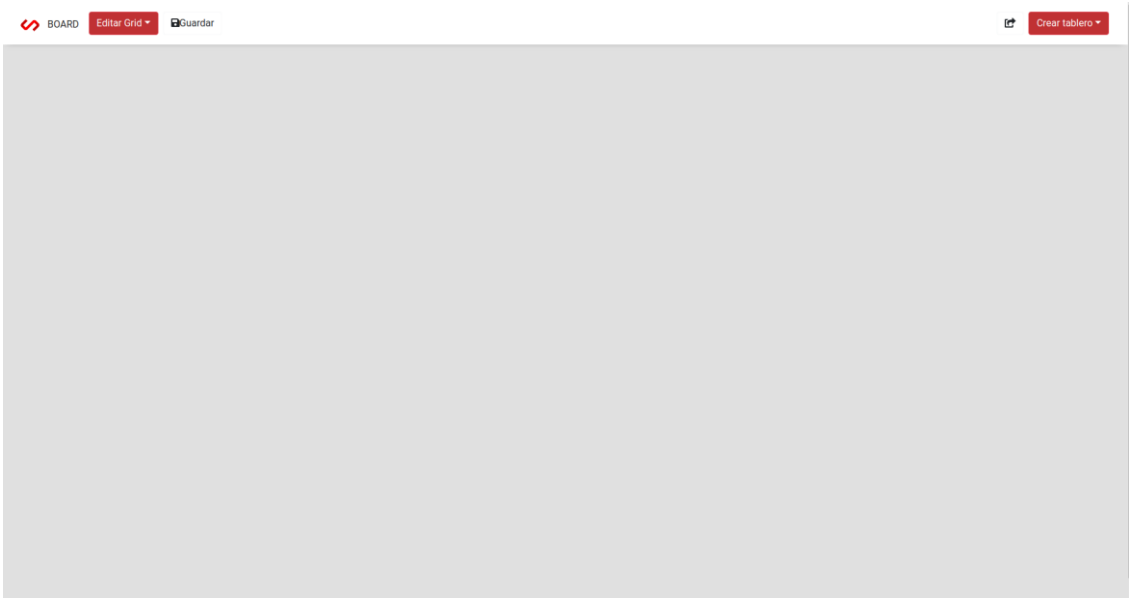

Ilustración 10. Modo edición con tablero vacío.

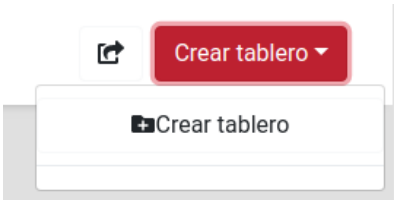

Ilustración 11. Crear tablero.

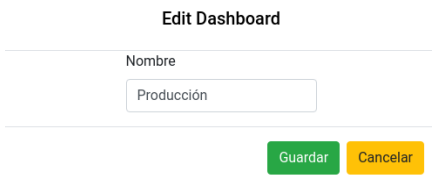

Ilustración 12. Modal de editar tablero.

El primer paso para crear nuestra capa de presentación fue definir el *layout* o apariencia estética de las diferentes interfaces que debía ofrecer el sistema y cómo sería la navegación a través de tales interfaces. Este diseño se describe a continuación.

La *Ilustración 10* muestra el modo edición de la aplicación sin ningún tablero creado. En este estado, la aplicación no permite realizar nada a excepción de crear un tablero. Pulsando sobre el botón de *"*Crear tablero*"*, que vemos más aumentado en la *Ilustración 11*, se abre un cuadro de dialogo (simular al de la *Ilustración 12*) que nos pide un nombre para el tablero a crear.

Una vez tengamos creados varios tableros, se pone como título del botón, donde antes salía crear tablero, el nombre del tablero en el que estemos actualmente, como podemos percibir en la *Ilustración 13*. Si pulsamos este botón, accedemos a la lista de tableros que dispone ese usuario, junto con dos botones que permiten editar y eliminar el tablero, respectivamente. Al final de este menú, también está la opción de volver a crear más tableros.

Una vez tenemos los tableros creados, podemos empezar a añadir documentos. Para ello debemos pulsar el botón de "Editar Grid" (*Ilustración 14)*. Si, por ejemplo, seleccionamos la opción de cargar archivo remoto, se abre un diálogo como el de la *Ilustración 15,* donde introducimos la URL o ruta donde está el archivo remoto a incorporar al tablero; y, si es necesario, se introduce el tiempo, en minutos, del refresco que va a tener ese documento.

Por último, si el documento se ha cargado correctamente y es del tipo admisible por la aplicación, se muestra un mensaje notificando que todo se ha ejecutado correctamente. Las acciones como cargar documentos, crear o borrar tableros e ítems (*Ilustración 16*).

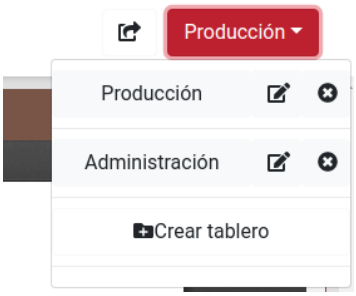

Ilustración 13. Lista de tableros.

Ilustración 14. Opciones para cargar documentos.

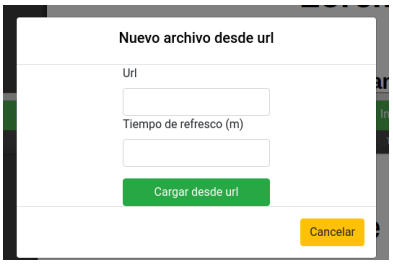

Ilustración 15. Modal cargar documento remoto.

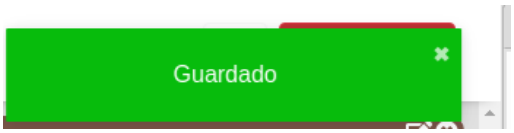

Ilustración 16. Toast de guardado exitoso.

Después de que se cargue un documento, se puede agrandar el ítem que contiene el documento, encogerlo o moverlo de forma sencilla por el *grid*. Un ejemplo del tablero con ítems ya cargados y ajustados se muestra en la *Ilustración 17*. En este tablero tenemos tres documentos cargados: (1) un *.pdf*, (2) un *.doc*; y, (3) un *.docx*. Los ítems renderizan el documento a través de *PDF.js*, una librería de *JavaScript* para el tratamiento y visualización de pdfs. Se decidió utilizar esta librería para obligar a los navegadores webs donde se fuese a visualizar la aplicación a ejecutar siempre el mismo visualizador, ya que, si no, cada explorador lo mostraría con el suyo propio. *PDF.js* también permite la visualización de pdfs en una web sin descargarlos en dispositivos Android.

Los ítems que hemos añadido al tablero se pueden eliminar pulsando sobre la "x" de arriba. También se pueden editar el nombre y el color de cada ítem pulsando en el botón adyacente a la "x". Al seleccionar la opción de editar, se abre un cuadro de diálogo como

el de la *Ilustración 18.* Una vez tenemos configurado y personalizado el tablero a nuestro gusto, lo podemos gurdar utilizando el correspondiente botón, situado en la zona superior izquierda.

Además, es posible compartir el tablero creado o mostrarlo en local en modo visualización. El modo visualización es un modo que permite visualizar el tablero con sus ítems sin que sea posible modificarlo. Este modo está diseñado para mostrar el tablero en pantallas Android de grandes dimensiones, o en proyectores, y por eso se muestra en pantalla completa. Este modo es el que vemos en la *Ilustración 20*, mientras que en *la Ilustración 19* se muestra el cuadro de díalogo que aparece a la hora de compartir el tablero.

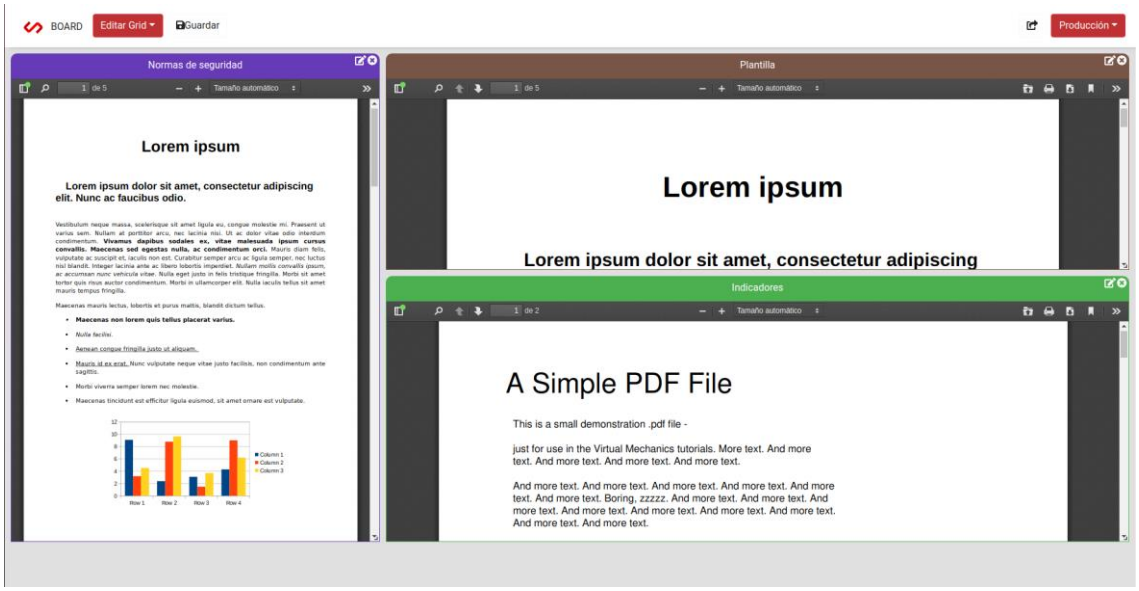

Ilustración 17. Modo edición con documentos.

Ilustración 18. Modal de editar ítem.

Ilustración 19. Modal de compartir tablero.

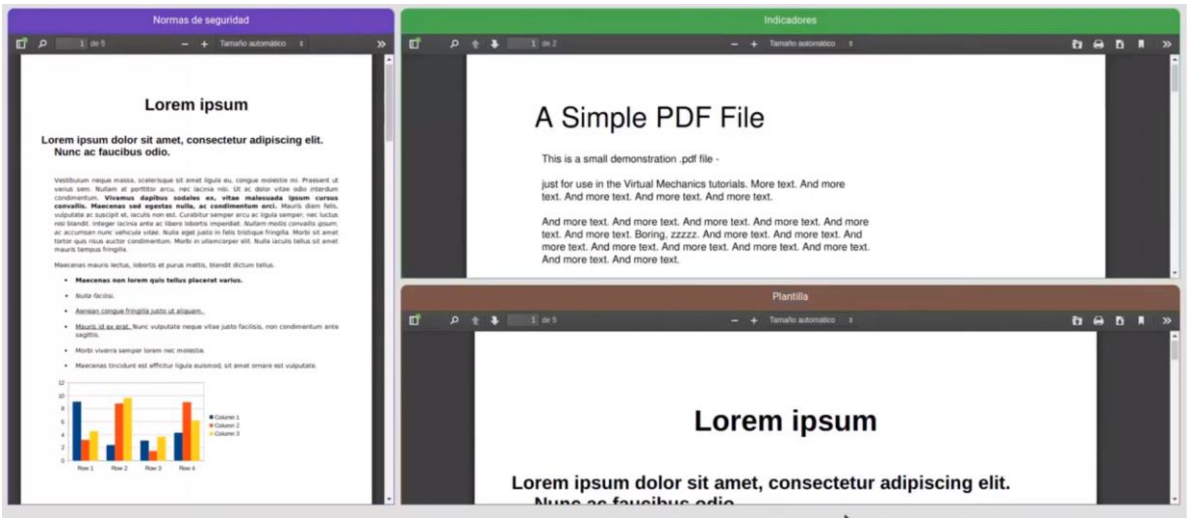

Ilustración 20. Modo visualización.

3.4.2. Implementación

Para implementar la capa de presentación, se ha hecho uso, tal y como comentamos anteriormente, de *React* y del configurador e instalador de dependencias *Yarn*. Los módulos más importantes que hemos utilizado en este proyecto son: (1) *Axio*, para realizar las llamadas AJAX; (2) *Bootstrap*, para elementos estáticos CSS; (3) *Material-UI*, para la utilización de widgets como botones o cuadros de diálogo; y, (4) *react-grid-layout*, probablemente el componente más relevante, ya que es el encargado de gestionar el *layout* de los tableros.

La aplicación consta de 15 componentes de *React* y 5 ficheros de estilo *SASS*. Un ejemplo de ficheros *SASS* es la *Ilustración 21*.

```
1.  .navbar {
2.      background-color: white;
3.      color: black;
4.      box-shadow: 0 4px 10px -10px black;
5.
6.      .navbar-brand {
7.          text-transform: uppercase;
8.          font-size: 1em;
9.          padding-right: 1em;
10.         width: fit-content;
11.     }
12.
13.     a {
14.         color:black;
15.     }
16.
17.     …
18. }
```

Ilustración 21. Parte del estilo de la barra del header.

La *Ilustración 21* muestra un trozo de código en CSS de un archivo *SASS*. En este caso se trata del estilo de la barra del *header* de las aplicaciones de "*SOINCON*" desarrollado por el departamento de diseño. En esta ilustración podemos ver el color del elemento (línea 2), el color de la fuente dentro del elemento "a", en este caso negro (línea 3), entre otras cosas.

Para ilustrar brevemente el funcionamiento de *React*, la *Ilustración 22* muestra el código asociado al cuadro de diálogo para la inclusión de un fichero remoto. Dicho cuadro se diálogo se muestra en la *Ilustración 15*.

```
1.  import React, { Component } from "react";
2.  import { ToastContainer, toast } from "react-toastify";
3.  import "react-toastify/dist/ReactToastify.css";
4.  import { IpPdfService } from "./config";
5.  import { i18n } from "./i18n";
6.
7.  class Url extends Component {
8.    constructor(props) {
9.      super(props);
10.     this.state = {
11.       selectedFile: "",
12.       loaded: 0,
13.       time: ""
14.     };
15.     this.onClickHandler = this.onClickHandler.bind(this);
16.   }
17.
18.   onChangeHandler = event => {
19.     this.setState({ selectedFile: event.target.value });
20.   };
21.   onChangeHandlerTime = event => {
22.     this.setState({ time: event.target.value });
23.   };
24.   onClickHandler = () => {
25.     this.props.setUrl(this.state.selectedFile);
26.     this.props.setTime(this.state.time);
27.     fetch(IpPdfService + "/uploadFileUrl?url=" + this.state.selectedFile, {
28.       method: "POST",
29.       headers: {
30.         Accept: "application/json",
31.         "Content-Type": "application/json"
32.       }
33.     })
34.       .then(res => {
35.         // then print response status
36.         toast.success(i18n.loadC);
37.         this.props.submit();
38.       })
39.       .catch(err => {
40.         // then print response status
41.         toast.error(i18n.loadF);
42.       });
43.   };
44.
45.   render() {
46.     return (
47.       <div class="container">
48.         <ToastContainer />
```

Ilustración 22. Código del componente URL.

```
49.           <div class="offset-md-3 col-md-6">
50.             <div class="form-group files">
51.               <label>{i18n.url} </label>
52.               <input
53.                 class="form-control"
54.                 multiple
55.                 onChange={this.onChangeHandler}
56.               />
57.               <label>{i18n.refresh} </label>
58.               <input
59.                 type="number"
60.                 pattern="[0-9]*"
61.                 class="form-control"
62.                 multiple
63.                 onChange={this.onChangeHandlerTime}
64.               />
65.             </div>
66.             <div class="form-group"></div>
67.             <button
68.               type="button"
69.               class="btn btn-success btn-block"
70.               onClick={this.onClickHandler}
71.             >
72.               {i18n.loadUrl}
73.             </button>
74.           </div>
75.         </div>
76.       );
77.     }
78.   }
79.
80.   export default Url;
```

Ilustración 23 cont. Código del componente URL.

En primer lugar, se importan las dependencias que va a utilizar este componente (líneas 1 a 5). A continuación, la implementación del componente se divide en tres secciones: (1) constructor (líneas 7 a 16); (2) funciones auxiliares o de control de respuesta eventos (líneas 18 a 43); y, (3) implementación de la función *render* (líneas 45 a 77), que es una función que debe implementar cada clase *React* que se corresponda a un componente gráfico y que especifica qué código HTML debe ser visualizado cuando se muestre dicho componente. Dentro del constructor, definimos el estado de este componente. Este estado son una serie de variables, que al cambiar ejecuta otra vez el método *render* y renderiza de nuevo el HTML, de acuerdo con un paradigma de programación parecido al de la programación reactiva (Íñigo, 2017). Esta la función *render* muestra un pequeño formulario con un campo para introducir una URL (líneas 51 a 56), el tiempo de refresco del documento (líneas 57 a 64), y un botón de aceptar (líneas 67 a 71). El botón de cancelar es introducido y gestionado por defecto por React y el tipo de componente utilizado.

A cada elemento del formulario se le asocia una función del componente, la cual se invoca siempre que se produce un cambio en estos elementos (líneas 55, 63 y 70). La función invocada, por tanto, ejerce el papel de controlador de su correspondiente elemento del formulario.

Los dos primeros *onChangeHandler* o funciones controladoras, que encontramos se encargan de actualizar los campos del formulario a la vez que los escribimos y de guardar también en el estado los valores escritos para cuando se envíe el formulario. La tercera función se ejecuta cuando se pulsa el botón de enviar (líneas 24 a 43). Esta función escribe los valores de los campos del formulario en *this.props*, variable que sirve para recoger los atributos públicos de un componente para que puedan ser leídos por otros componente. De esta forma, en nuestro caso, tras escribir la URL y la tasa de refresco, el tablero desde el cual se ha invocado este cuadro de diálogo podrá acceder a estas propiedades para saber cuál ha sido la URL cargada y cuál debe ser su tasa de refresco.

Tras ello, se envía una petición asíncrona POST a la API para la gestión de archivos remotos, solicitando la carga del fichero deseado, y con dos posibles acciones de salida. En caso de que la petición concluya con éxito, se muestra un mensaje confirmando la carga del archivo y se invoca una de *callback* función escrita por el componente llamante en las propiedades del componente. En caso de error, se muestra un mensaje notificando dicho error.

Cabe destacar que la capa de presentación incorpora un *timer* por cada ítem añadido que disponga de tiempo de refresco. Este *timer* hace las gestiones y peticiones necesarias al servicio para actualizar los documentos según el tiempo de refresco que se le haya definido.

Tal como se puede apreciar en la Ilustración 24, el componente está preparado para ser utilizado en diferentes idiomas. Para realizar esta internacionalización, se utilizó el componente *react-localization* para crear un fichero *i18n*, ya que hay clientes de *SOINCON* que tienen sede en diferentes países. La ventaja de este componente, respecto a otros, es que es capaz de detectar el idioma del navegador y del sistema operativo y renderiza la web en ese idioma, siempre y cuando exista traducción en ese idioma en el fichero de configuración del *i18n*.

```
1.  import LocalizedStrings from 'react-localization';
2.
3.  export let i18n = new LocalizedStrings({
4.    en: {
5.      editGrid: "Edit Grid",
6.      save: "Save",
7.      addLocal: "Add Local Document",
8.      ...
9.    },
10.   es: {
11.     editGrid: "Editar Grid",
12.     save: "Guardar",
13.     addLocal: "Añadir archivos local",
14.     ...
15.   }
16. });
```

Ilustración 24. *i18n* de la aplicación.

4. Pruebas y despliegue

4.1. Metodología de pruebas

Durante el desarrollo del proyecto no se ha seguido ninguna metodología de pruebas específica, sino que simplemente se diseñaron e implementaron algunas pruebas, sobre todo para las partes del *back-end* que se consideraban más críticas. También se desarrollaron unas pruebas de las versiones *alpha*. No obstante, al final de cada sprint si se relizaban una serie de pruebas de aceptación con el directivo que ejercía de *Product Owner*, y se recogían las modificaciones sugeridas.

Por otro lado, las dos APIs desarrolladas durante el proyecto fueron probadas con la herramienta Postman (Postman, s.f.). Por último, una vez que se tuvo una versión operativa y completa de la aplicación, se realizaron diferentes pruebas sobre el producto completo.

4.2. Pruebas unitarias

Durante el desarrollo del proyecto se realizaron pequeñas pruebas unitarias para comprobar el correcto funcionamiento de los métodos. A modo de ejemplo, en la *Ilustración 25*, se muestra una de las pruebas del método de servicio que cargaba documentos de forma remota al servidor. En total se realizaron pruebas para 5 métodos aproximadamente.

```
1.      @Test
2.      public void  loadFileURLTest01(){
3.          fileStorageUrlService.downloadFile(fileURL, saveDir);
4.          File f=new File(renderFile);
5.          if(!f.exists()) {
6.              fail("No se ha cargado el documento desde el URL");
7.          }
8.      }
```

Ilustración 25. Prueba unitaria de *cargar desde remoto*.

Este test es bastante sencillo. En primer lugar, se llama al método *dowloadFile* que se desea probar; se le pasa como parámetros la URL donde se encuentra el documento a buscar y la ruta de la carpeta del servidor donde se va a guardar (línea 3). A continuación, se crea un objeto de tipo *File* con la ruta del archivo (línea 4) y posteriormente se comprueba si existe, si no existe, se lanza el *fail* con el error (línea 5 y 6).

4.3. Pruebas de los servicios

Una vez tenía los dos servicios realizados, probé todos sus *end-points* con la herramienta *Postman*. *Postman* es una herramienta que permite especificar y ejecutar llamadas HTTP, así como visualizar de manera amigable al usuario el resultado de dichas llamadas. Por tanto, usando *Postman*, podemos comprobar si lo devuelto por un *end-point* de una API HTTP concuerda con lo esperado.

Usando *Postman*, probé que el servicio encargado de dar acceso a los repositorios disponía de una seguridad básica con usuario y contraseña de manera que solo pudiesen acceder a los *end-points* si se tenía las credenciales adecuadas. Obviamente, utilizando también *Postman* se probó diferentes tipos de llamadas para así ver si la respuesta del servidor era la esperada. Una prueba que se realizó es la *Ilustración* 25, que muestra la llamada al método de la API para solicitar un documento PDF, así como su resultado[3].

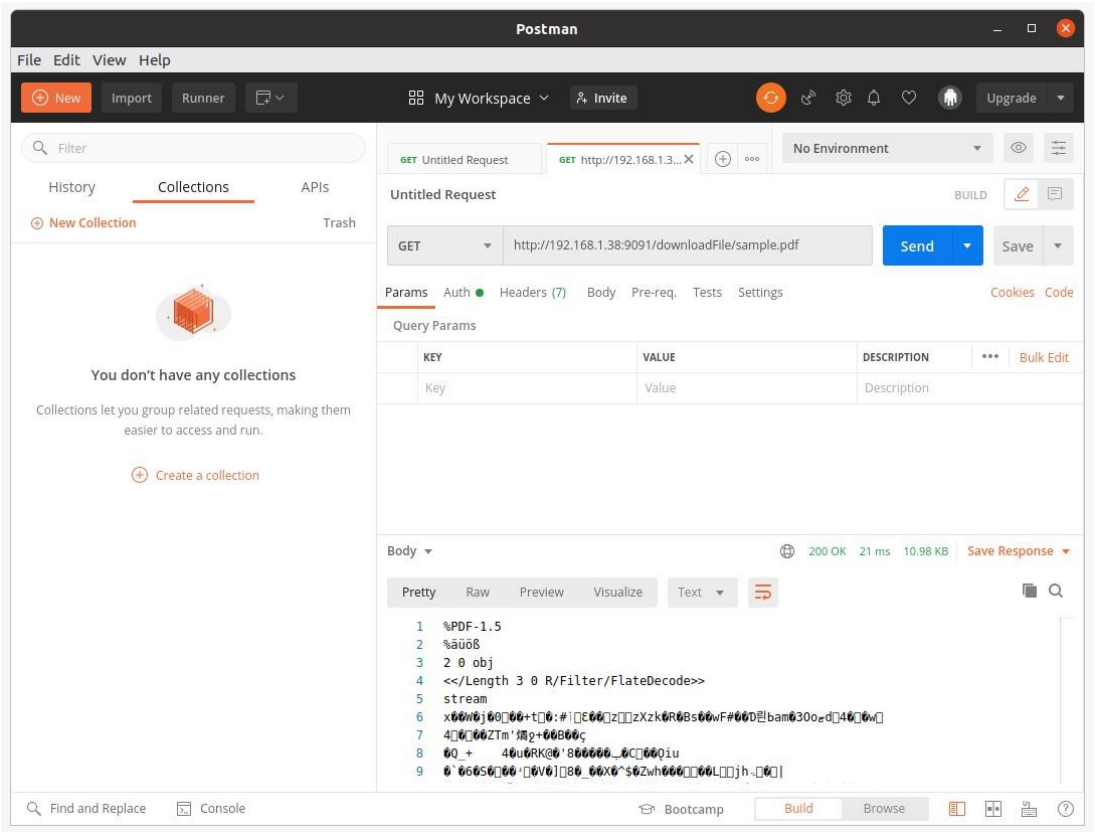

Ilustración 26. Prueba de la API con Postman.

[3] El resultado lo copiaba en un bloc de notas y le ponía la extensión PDF, para así comprobar si era el PDF esperado.

4.4. Pruebas de la versión preliminar

Cuando el producto estuvo casi terminado, se probó con un par de compañeros que no habían utilizado ni visto nunca la aplicación. Para esto, definí en un bloc de notas todas las funcionalidades a probar y como deberían actuar cada una de ellas.

Para la ejecución de las pruebas me puse en contacto con los dos compañeros citados y de forma individual les mostré la aplicación en una pantalla táctil Android de 20 pulgadas. Estos compañeros utilizaron la aplicación al principio sin ningún tipo de guía, y posteriormente bajo mi supervisión, para asegurarme de que se probaban todas las funcionalidades definidas. Durante las pruebas fui apuntando si lo que había definido cumplía con lo que se estaba realizando.

Las pruebas tuvieron un resultado exitoso, a excepción de que el primer compañero encontró un bug a la hora de borrar los ítems. Este error era bastante sencillo de solucionar, y se solucionó antes de que el otro compañero probase la aplicación.

4.5. Pruebas de aceptación

Estas pruebas servían para verificar si la aplicación se estaba desarrollando según lo que deseaba el directivo de la empresa, que es el que ejercía de *Product Owner* y principal interesado en el desarrollo del producto.

Al finalizar cada sprint, me reunía con este directivo y con un par de compañeros que aportaban ideas y revisaban lo realizado en ese sprint. Tras estas revisiones, se proponían modificaciones y sugerencias de mejora. En total se sugerían entorno a 4 mejoras por cada revisión. La mayoría de estas mejoras eran implementadas en sprints siguientes, como en el caso del *Sprint 5* que nombramos anteriormente.

4.6. Despliegue

Al ser una aplicación terminada hace relativamente poco, no se ha desplegado aún en ningún cliente, aunque la aplicación sí forma actualmente parte de *EMI Suite*, la aplicación bandera de Soincon para la Industria 4.0. La aplicación como tal, se desplegó en un principio en mi computador de la empresa. Para ello generé los *wars* de los dos servicios implementados utilizando *Maven*. Luego creé el *build* de la capa de presentación a través de *Yarn*, lo empaqueté en un *war* de forma manual y lo desplegué en Tomcat, introduciendo los *wars* en la carpeta *webapps*.

Tras esta primera prueba realicé un despliegue en el servidor de demos de *SOINCON*; de tal forma que la áplicación pudiese ser mostrada a los clientes. Para ello seguí los mismos pasos que al desplegar en local, pero instalando y comprobando que el servidor de demos tuviese la dependencia de *LibreOffice* requerida para el tratamiento de los documentos, *MySQL* para la base de datos y las carpetas necesarias para el almacenamiento de los documentos y su procesamiento.

5. Conclusiones y trabajos futuros

Este proyecto me ha aportado muchísimos conocimientos ya que he trabajador solo en este proyecto y he tenido que aprender a valerme más por mí mismo. Aunque podía pedir ayuda a otros compañeros, prefería la mayoría de las veces que no, ya que no estaban del todo en el contexto del proyecto, por lo que buscaba todo en internet o a través de foros lo que me ayudó mucho a desarrollar el autoaprendizaje.

Todo este proyecto me ha motivado hasta el punto de que he estado *enganchado* a su desarrollo y no me ha costado mucho esfuerzo anímico realizarlo. El proyecto también me ha ayudado a conocer mis límites, porque hay cosas que me han llegado a desquiciar de lo saturado que estaba, pero que al día siguiente solucionaba en minutos. Por ello aprendí a saber darme un descanso y despejar la mente.

Este proyecto me ha dado la oportunidad de aplicar los conocimientos aprendidos en mis cuatro años de carrera en un ambiente profesional y no académico, lo que supone una motivación extra. En resumen, este Trabajo de Fin de Grado me ha resultado muy gratificante en lo personal, porque he conseguido hacer algo que al principio me parecía imposible, pero que, gracias a la ayuda de mis compañeros, a los conocimientos adquiridos en el grado y a todo el equipo de *SOINCON*, al final pude conseguirlo. El proyecto también me ha ayudado a desarrollar la comunicación en equipo y a comprender mejor cómo va a ser mi vida profesional en un futuro.

Para finalizar, este proyecto tiene planes de futuro, ya que se ha pensado en añadirle más funcionalidades que pueden necesitar los futuros clientes, Por ejemplo, se ha barajado con la posibilidad de no solo mostrar documentos ofimáticos y fotos, sino también graficas dinámicas, señales o estados de máquinas en tiempo real. Otra posible mejora s es adaptar toda la aplicación a una nueva línea de diseño de la compañía que se estableció una vez terminé la aplicación. También se está pensando en cambiar la forma de procesamiento del documento, guardando los documentos ya procesados en la base de datos, en formato *Base64*, en vez de en una carpeta del servidor.

6. Bibliografía

Altadill, P. X. (2019). *Desarrollo Web con React.* Anaya.

Cosmina, L., Harrop, R., & Schaefer, C. (2017). *Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools.* Apress.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture.* Addison-Wesley Educational Publishers Inc.

Giraudel, H. (2016). *Jump Start Sass.* Sitepoint.

Íñigo, J. A. (10 de 2017). *Profile*. Obtenido de Profile: https://profile.es/blog/que-es-la-programacion-reactiva-una-introduccion/

Keith, M., Schincariol, M., & Nardone, M. (2018). *Pro JPA 2 in Java EE 8: An In-Depth Guide to Java Persistence APIs.* Apress.

Postman. (s.f.). Obtenido de Postman: https://learning.postman.com/docs/publishing-your-api/documenting-your-api/

Sutherland, J., Sutherland, J., & Gordo del Rey, V. E. (2018). *Scrum: El revolucionario método para trabajar el doble en la mitad de tiempo.* Ariel.

Yarn. (1 de 09 de 2020). *Yarn.* Obtenido de Yarn: https://classic.yarnpkg.com/en/docs/