Facultad
de
Ciencias

APLICACION WEB PARA LA GESTIÓN DE BONOS DE DESCARGO EN UNA INDUSTRIA CEMENTERA

(Web application for managing machinery unloading bonds)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Miguel Casamichana Bolado

Director: Patricia López Martínez

Junio – 2022

Resumen

Hoy en día hay numerosas empresas que están remodelando su lógica de negocio para adaptarla a las nuevas tecnologías con el fin de agilizar sus procesos internos y aumentar su productividad.

Un gran ejemplo de esto es el producido en una empresa cementera. En ésta, se realizan diariamente una serie de descargos de tensión en ciertas máquinas con el fin de dejarlas operativas para su uso. Este proceso se puede resumir en la creación de una solicitud, que debe ser aceptada y tratada por varias personas, que se realiza de forma manual y rudimentaria, e implica un gran esfuerzo en tiempo por parte de las personas que intervienen en el proceso.

El objetivo principal de este Trabajo Fin de Grado es digitalizar, y por tanto agilizar, dicho proceso específico interno de la empresa lo cual implicaría una mejora en el grado de productividad. Para ello, se ha desarrollado una aplicación web, sencilla de usar e intuitiva, la cual hace uso de un servicio REST personalizado para dar soporte a todas las operaciones que se necesiten. Además, se ha proporcionado cierta gestión de usuarios, con el fin de poder identificar de forma correcta a cada usuario que use la aplicación, y poder realizar posibles seguimientos de la maquinaria usada por los empleados durante su estancia en la empresa.

La aplicación web se ha desarrollado bajo el framework de Angular y ha sido escrita en TypeScript. Además, el servicio REST, escrito completamente en Java, ha sido implementado con el framework Spring. Por último, se optó por usar una base de datos MySQL cuyo ORM ha sido implementado mediante JPA e Hibernate.

Palabras clave: REST, Angular, Spring, Aplicación web, Bono de descargo.

Abstract

Nowadays, there are many companies that are remodeling their business logic to adapt it to new technologies in order to speed up their internal processes and to increase their productivity.

A great example of this is the one produced in a cement company. In this one, a list of voltage discharges are carried out daily on certain machines in order to put them in a operational state. This process can be summed up in a request, which must be accepted and treated by some employees, which is carried out manually and in a rudimentary way, and which implies a huge time effort by the people who are involved on it.

Therefore, the main objective of this project is to digitalize that process in order to speed up a company specific internal process which would imply in a better grade of productivity. To achieve that objective, an intuitive and simple web application has been developed, which uses a customized API REST to support every operation that is needed.

Furthermore, user management has been provided in order to identify correctly every single user of the application, and to perform some tracing of the machinery used by the employees during their time at the company.

The web application has been developed with the Angular´s framework and has been written with TypeScript. Moreover, the REST service, which has been written with Java, has been developed with the Spring´s framework. Lastly, it was decided to use a MYSQL database which ORM was implemented by the JPA and Hibernate frameworks.

Key Words: API REST, Angular, Spring, Web app, Unloading bonds.

Índice de contenidos

Índice de figuras

Índice de tablas

1. Introducción

En una empresa cementera se utiliza a diario una maquinaria, la cual antes de ser usada después de un periodo de inactividad, debe pasar por un estado de mantenimiento durante el cual se le aplica cierta tensión. Para que un empleado pueda volver a usar dicha maquinaria, es necesario descargarla, esto es, quitar la tensión, para lo cual el empleado debe acudir al cuadro de control y rellenar un formulario de solicitud de descarga o bono de descargo. En este trámite intervienen tres tipos de empleados, los cuales realizan operaciones específicas y distintas entre sí.

En la empresa para la que se desarrolla este trabajo, este proceso se realiza actualmente de manera rudimentaria y manual, por lo que el objetivo principal de este Trabajo Fin de Grado (TFG) es la automatización de este proceso. En las siguientes subsecciones se describirán detalladamente las bases del proceso mencionado y el listado de objetivos que pretende satisfacer este trabajo.

1.1. Proceso de descarga de maquinaria

Para explicar el proceso en detalle, hay que detallar primero qué es un bono de descargo, qué usuarios intervienen en el proceso y cómo se gestiona el trámite paso a paso.

1.1.1. Formulario de solicitud de descarga de maquinaria

Un formulario de solicitud de descarga es un documento en el que se recoge toda la información requerida desde que se solicita el descargo de una máquina hasta que se confirma que la máquina en cuestión se ha vuelto a poner en estado de mantenimiento.

Esta información es la siguiente: el número del bono de descargo, la identificación de la máquina a descargar en cuestión, es decir, el nombre y el número de la máquina, las instrucciones de descargo y una especificación del trabajo a realizar. Además, el bono se divide en seis partes correspondientes a las seis operaciones que se realizan durante el trámite. Por cada una de estas seis partes, se registra el nombre y firma del empleado que haya hecho esa operación, junto a la fecha y hora de ésta. Es decir, al finalizar por completo el proceso se han debido registrar seis firmas, con nombres, fechas y horas además de los datos iniciales mencionados previamente.

1.1.2. Tipos de empleados que intervienen en el proceso

En este proceso intervienen entre tres y seis usuarios, los cuales pueden tener uno de los siguientes roles:

- Solicitante: es el empleado que desea operar sobre la maquinaria de la empresa. Se encarga de solicitar al cuadro de control la descarga de maquinaria y, posteriormente, de enviar un aviso sobre la finalización del trabajo sobre una máquina. Es el rol predominante dentro de la empresa.
- Operador de cuadro: es el empleado que trabaja desde el cuadro de control. Se encarga de toda la gestión del proceso de carga/descarga de maquinaria mediante el uso del cuadro de control de SCADA. Sirve de enlace en la comunicación entre los solicitantes y los instrumentistas.
- Instrumentista eléctrico: es el responsable de quitar o añadir tensión a la maquinaria, es decir, sacar/poner los fusibles para garantizar el estado de “en uso” o “en mantenimiento” de dicha maquinaria según las indicaciones del cuadro de control.

1.1.3. Detalle del trámite

El proceso por el cual un empleado solicita el descargo de una máquina se puede simplificar en los siguientes pasos, que se resumen en la Figura 1:

1. Un solicitante o un grupo de solicitantes comienzan el trámite debido a la necesidad de sacar tensión de una máquina concreta en una fecha y hora determinada para poder comenzar a trabajar con ella. Para ello, el solicitante acude al cuadro de control con el fin de rellenar un formulario de solicitud de descarga o "bono de descargo". Una vez rellenado el formulario, éste es entregado a un operador de cuadro.

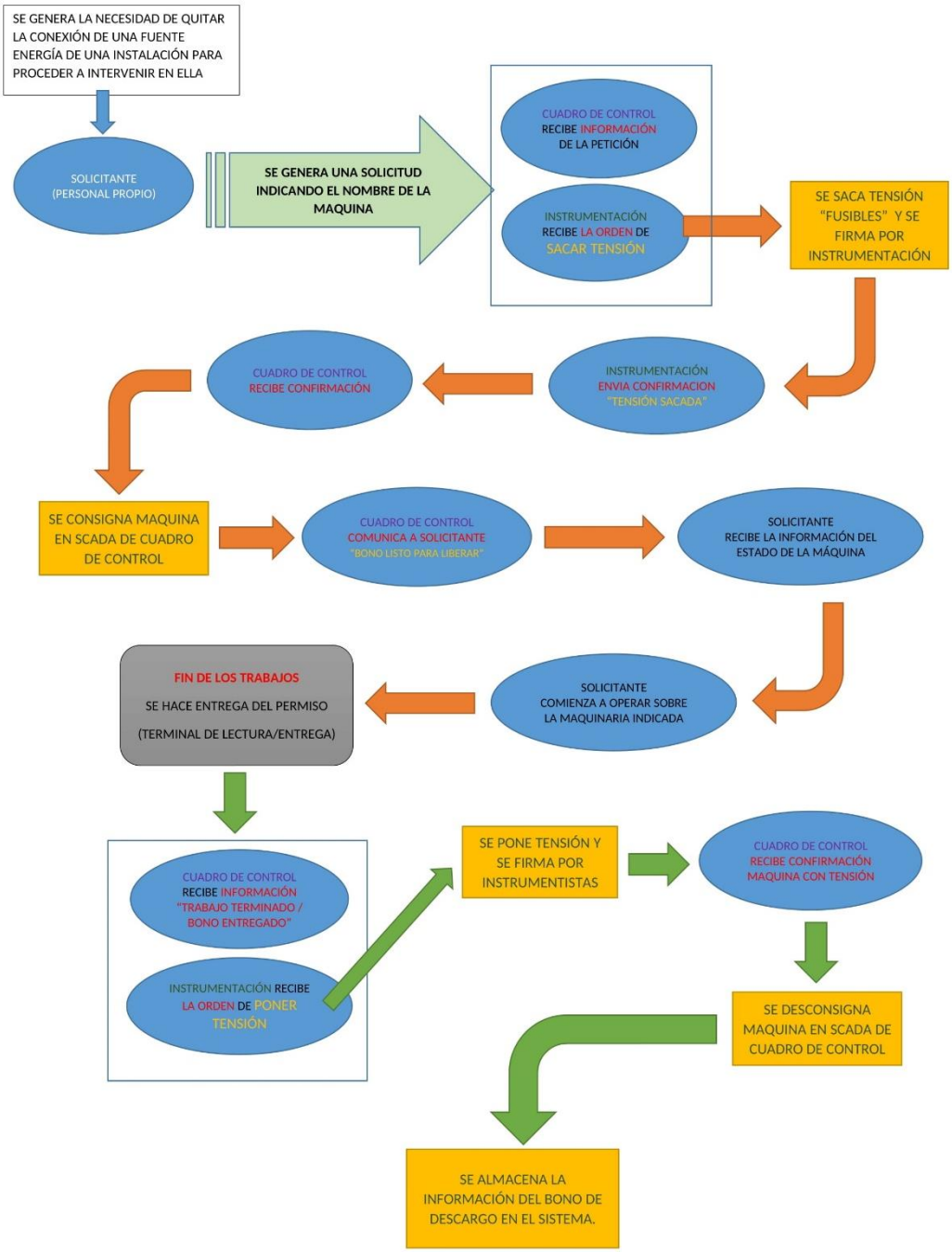

Figura 1. Flujo de pasos del proceso de descargo de maquinaria.

2. El operador de cuadro almacena el nuevo bono en el registro de bonos sin finalizar y, cuando determine que es necesario, comunica el deseo de descargo a un instrumentista eléctrico. Para ello, se cerciora previamente del estado actual de la máquina a descargar a través del sistema SCADA, ya que puede estar en uso por otro conjunto de empleados y por lo tanto no tendría sentido enviar a un instrumentista a descargar una máquina que ya se encuentra en ese estado.
3. Una vez el instrumentista quita la tensión a la máquina, comunica el estado de ésta de nuevo al cuadro de control. El operador consigna el estado de la máquina en el cuadro de SCADA y comunica al solicitante el estado de la máquina.
4. El solicitante recibe la confirmación, imprime el bono con la confirmación del descargo y comienza a trabajar.
5. Cuando finalice el trabajo en la máquina, el solicitante debe volver a acudir al cuadro de control para informar al operador de que se ha finalizado el trabajo sobre la maquinaria.
6. El operador de cuadro se cerciora de que nadie más esté usando la máquina a cargar, ya que, en caso contrario, se podrían producir accidentes mortales sobre aquellos que sigan operando sobre una máquina a la que poner tensión. En caso de que nadie más esté utilizando la máquina se envía un aviso de nuevo a los instrumentistas para que carguen de nuevo la maquinaria.
7. El instrumentista recibe la orden de cargo de maquinaria por parte del cuadro de control. Pone tensión de nuevo en la máquina en cuestión, y una vez acabado, le comunica el nuevo estado de ésta al operador que le ha contactado.
8. Por último, cuando el operador de cuadro recibe la confirmación de la puesta en mantenimiento de la máquina, consigna el estado de ésta en el SCADA, y registra el bono de descargo como finalizado.

1.2. Contexto de la aplicación

Este proceso de gestión de los bonos de descargo hace que haya mucho tráfico por el cuadro de control, lo cual en la situación generada por el COVID 19 no es aconsejable para la salud de los empleados que transitan y/o trabajan en esta zona.

Independientemente de esto, el proceso en sí es muy lento e ineficiente, ya que hace a los solicitantes acudir a dicha oficina de control, la cual puede estar situada a grandes distancias de dónde se encuentran los empleados trabajando, para realizar solicitudes de descargo, de cargos o para recibir confirmaciones de éstas. Además, cabe destacar que al realizarse este proceso en base a formularios en papel, se almacenan grandes cantidades de formularios a diario, lo que dificulta enormemente la búsqueda de bonos específicos, además de generar la necesidad de contar con un espacio físico amplio donde archivar o almacenar todos estos formularios.

Esta aplicación web surge de la idea de digitalizar este proceso, con el fin de agilizar dicho trámite, facilitar el almacenamiento de bonos de descargo y presentar nuevas funciones relacionadas con la gestión de bonos de descargo.

1.3. Objetivos

En base al contexto y proceso expuestos previamente, la aplicación debe permitir realizar todo este trámite de forma sencilla y eficaz con el fin de facilitar a los usuarios la carga y descarga de maquinaria dentro de la cementera. Para poder realizar la digitalización de este trámite, la aplicación debe dar soporte a la gestión de bonos de descargo y a la gestión de usuarios del sistema.

En cuanto a la gestión de bonos, se pretende:

- Poder tramitar paso a paso el proceso de descarga de una máquina. Es decir, realizar los pasos mencionados previamente de forma sencilla y telemática.
- Almacenar un histórico de todos los bonos de descarga (realizados, en curso y por tramitar).
- Realizar búsquedas filtradas dentro del histórico.
- Firmar digitalmente, según corresponda, las distintas partes del bono.

En cuanto a la gestión de usuarios, se pretende:

- Realizar una identificación única de cada empleado que use el sistema mediante la asignación de un usuario único e intransferible dentro del sistema con el fin de acreditar la autoría de las operaciones que se realicen durante el trámite.
- Persistir los datos de cada usuario que use el sistema mediante un registro.
- Ofrecer una gestión de roles durante la fase de registro, con el fin de asignar las funciones que sean de carácter específico a los roles que correspondan.

1.4. Organización de la memoria

Este documento se divide en ocho capítulos, los cuales, en su totalidad, tienen la finalidad de describir el proceso seguido para desarrollar la aplicación desarrollada. Los capítulos expuestos son los siguientes:

- Primero, en el capítulo 2 se describen las tecnologías, herramientas, el material utilizado por el desarrollador y la metodología de desarrollo software seguida a lo largo del proyecto.
- Posteriormente, en el capítulo 3, se definen los requisitos y las especificaciones que debe satisfacer el proyecto.
- Más tarde, se explica en el capítulo 4 el diseño software por el que ha optado el desarrollador. Se especifica el diseño de las tres capas de la aplicación: capa de presentación, capa de negocio y capa de persistencia.
- Por consiguiente, se detalla en el capítulo 5 cómo se han implementado estas tres capas y se profundiza en los conceptos más complejos encontrados.
- Una vez explicado cómo se ha implementado la aplicación, en el capítulo 6, se detallan las pruebas realizadas con las que se ha verificado el correcto comportamiento de la aplicación desarrollada.
- Por último, en el capítulo 7 se exponen las conclusiones obtenidas una vez se ha finalizado el proyecto y se enumeran una lista de posibles futuras funcionalidades nuevas a incorporar a la aplicación.

2. Tecnologías, herramientas y metodología de desarrollo

En este proyecto se ha utilizado material específico para desarrollar la capa de frontend, material específico para la capa de negocio, material específico para la capa de datos y herramientas específicas para el despliegue de la aplicación. En este apartado, se va a explicar brevemente cada elemento usado en el diseño y desarrollo de la aplicación.

2.1. Material utilizado para la capa de negocio

2.1.1. Eclipse

Eclipse es un entorno de desarrollo, comúnmente usado para el desarrollo software, que consta, entre otros, de un gestor de archivos, editores de texto, un compilador que facilita la compilación en tiempo real, asistentes para la creación de proyectos, soporte para la realización de pruebas unitarias con JUnit, etc. Además, soporta numerosos tipos de lenguajes de programación como Java o C++ y permite la instalación y uso de plugins externos.

2.1.2. Spring

Spring [1] es un framework utilizado comúnmente para el desarrollo de aplicaciones Java. Una de las funciones que más se utiliza de este framework es el patrón de inyección de dependencias [2] mediante el uso de la anotación @Autowired. El objetivo de este patrón es el de proporcionar a una clase instancias ya construidas de las clases de las que depende con el fin de no responsabilizar a dicha clase con la inicialización o construcción de éstos.

Entre otras funcionalidades, Spring proporciona una serie de anotaciones que permiten desarrollar servicios REST de forma sencilla. Además, gracias a Spring Data se puede configurar de manera sencilla la conexión con la base de datos usada por el servicio mediante un archivo de configuración y se facilita la creación de repositorios mediante el uso de interfaces que extiendan a la interfaz CrudRepository, las cuales, con tan solo especificar la entidad y el tipo de clave primaria de ésta (CrudRepository<Entity, Key>) proporcionan todas las operaciones CRUD de un repositorio. Además, este framework es compatible con numerosos frameworks como JPA o Hibernate.

2.1.3. Spring Tool Suite

Este es un entorno de trabajo, el cual está integrado en el entorno de desarrollo de Eclipse, que permite una fácil integración del framework Spring. Facilita la creación de aplicaciones Spring y permite desarrollarlas, ejecutarlas, desplegarlas y realizar tests en ellas.

2.1.4. SoapUI

SoapUI es una herramienta comúnmente utilizada para la prueba y validación de servicios REST, entre otros. Con esta se pueden realizar peticiones HTTP a los diferentes recursos que proporciona un servicio web con la finalidad de comprobar que éstos funcionan correctamente. Para ello, se debe especificar:

- El endpoint al que acceder, es decir, la URL donde se almacena el recurso.
- El tipo de operación de la petición, que generalmente es un GET, POST, PUT o DELETE.
- Las cabeceras de la petición HTTP y sus respectivos valores.

- El body, o cuerpo de la petición, en caso de cierto tipo de peticiones como las peticiones POST o PUT.

Una vez se mandan este tipo de peticiones HTTP, SoapUI permite conocer el estado de las respuestas de éstas, y el contenido de su resultado en caso de éxito.

2.1.5. JWT

JWT [3] o *Json Web Tokens* es un estándar que se basa en JSON para la creación de tokens utilizados principalmente para el envío de datos entre aplicaciones y/o servicios. Estos tokens garantizan la seguridad y validez de este tipo de intercambio de datos.

Un token JWT es una cadena de texto en formato JSON la cual está formada por tres partes:

- Header: contiene la información relacionada con el tipo de token y el tipo de algoritmo usado para realizar la firma (signature).
- Payload: contiene la información a compartir en forma de pares clave/valor.
- Signature: verifica que el token es válido, es decir, que no ha sido modificado. Este se forma con codificación Base64 del header y del payload y una clave secreta.

Estos tokens se usan comúnmente para gestionar autorizaciones y/o autenticaciones en cualquier tipo de aplicaciones o para la protección de recursos web.

2.1.6. Lombok

Lombok es una librería de Java que evita la creación de los métodos más comunes en el modelo de una aplicación como pueden ser los getters, setters o el método toString, principalmente. Para ello hace uso de anotaciones como @Getter o @Data las cuales afectan a todos los atributos de la clase afectada.

A simple vista puede parecer una librería bastante sencilla, sin embargo, facilita la legibilidad del código en clases con muchos atributos.

2.1.7. Java

Este es un lenguaje de programación muy conocido el cual se caracteriza por su simplicidad, por ser orientado a objetos, por ser multihilo y por ser distribuido.

Estas características hacen que Java sea un lenguaje universal, cuya curva de aprendizaje es menor que otros competidores. Además, puede ejecutarse en cualquier hardware siempre que se cuente con un máquina virtual.

2.2. Material utilizado para la capa de datos

2.2.1. MYSQL

MySQL es un gestor de bases de datos, es decir, es un sistema que se encarga entre otras cosas, de administrar y gestionar bases de datos. Las bases de datos MySQL son relacionales, es decir, están basadas en el modelo relacional, una forma de estructuración de datos que se caracteriza por la definición de relaciones o tablas para el almacenamiento de datos.

2.2.2. DBeaver

DBeaver es una herramienta que permite administrar bases de datos y trabajar con éstas de una forma fácil y eficaz aportando numerosas funciones que agilizan varios procesos por parte del desarrollador.

Dentro de estas funciones destacan la visualización de datos, la manipulación de datos desde la propia interfaz gráfica y el soporte a la hora de generar sentencias SQL. Además, tras realizar una consulta, los datos mostrados son modificables permitiendo realizar una actualización o borrado sin necesidad de escribir las sentencias SQL. Tan solo bastaría con hacer uso de la interfaz gráfica en la que se muestran estos resultados y escribir sobre ésta directamente.

2.2.3. JPA e Hibernate

JPA o Java Persistance API es una especificación Java utilizada para la persistencia de datos mediante la implementación de un ORM, es decir, un mapeo objeto – relacional. Este mapeo consiste en la conversión de objetos de Java, en este caso, a un formato el cual permita ser almacenable en cualquier tipo de base de datos.

Esta especificación está basada en el uso de numerosas anotaciones como @Entity para definir entidades que se mapearan a tablas de la base de datos, @Column para definir columnas de una tabla o @JoinColumn para definir relaciones entre tablas, con las cuales se realiza dicho mapeado.

Hibernate es un framework que implementa esta interfaz para desarrollar este mapeado. Por lo tanto, se podría decir que Hibernate es una implementación de JPA.

2.3. Material utilizado para la capa de frontend

2.3.1. Visual Studio Code

Esta herramienta es un editor de texto usado frecuentemente por desarrolladores en la edición de archivos. Además, consta de un depurador, de varias terminales; y presenta numerosas funciones que facilitan y agilizan el desarrollo de código, como tener atajos de teclas propios, realizar refactorizaciones, o tener a disposición del usuario una cantidad ingente de plugins descargables e instalables desde la misma herramienta.

2.3.2. Angular

Angular [4] es un framework de código abierto basado en TypeScript (TS) cuya función principal es crear aplicaciones web escalables. Se caracteriza por tener integrado un conjunto amplio de librerías que cubren las funciones principales para desarrollar aplicaciones web, como el enrutamiento (routing), la comunicación cliente – servidor, etc.; y por tener integrado un conjunto de herramientas que dan soporte al desarrollador para desarrollar, compilar y testear su código.

Este framework ofrece a su vez, un amplio conjunto de comandos con los que ejecutar la aplicación, depurarla, crear archivos compilados del proyecto con los que desplegarla, generar componentes (con sus templates, hojas de estilo y archivos de testing) automáticamente, etc. Para ejecutar dichos comandos, este framework requiere de Node.js.

2.3.3. Angular SignaturePad

Angular SignaturePad es una librería, integrable con Angular y escrita en TypeScript, que facilita la inserción de SignaturePads, u “hojas de firma”, en nuestros templates, y asociar a estos una lógica. Éste incorpora funciones para gestionar eventos en la hoja de firma, convertir las firmas digitales en imagen o string entre otros, limpiar el pad, etc.

Además, estos pads tienen una configuración editable, cuyas características modificables principales son las siguientes:

- Modificar el “pintado” de la firma: se puede asignar al lápiz un tamaño, un color, una velocidad de pintado y un parámetro que especifica cada cuanto tiempo se pinta.
- Modificar la hoja de firma: se puede especificar la anchura y altura mínima y máxima de la hoja y el color de fondo.

2.3.4. TypeScript

TypeScript es un lenguaje de programación de código abierto (open source), cuya sintaxis es completamente derivada de JavaScript. Se diferencia de éste principalmente en que añade tipos estáticos, como void o boolean entre otros, y objetos basados en clases.

JavaScript, y por lo tanto TypeScript, es un lenguaje comúnmente usado para la creación de aplicaciones web debido a que es un lenguaje nativo para los navegadores web y porque permite crear contenido interactivo y dinámico.

2.3.5. Jasmine y Karma

Jasmine y Karma, utilizados en este caso bajo el framework de Angular, son los frameworks encargados de dar soporte a la realización de pruebas unitarias sobre la aplicación web.

Mientras que el framework de Karma es necesario para gestionar la configuración de la ejecución de estos tests, Jasmine, es el encargado de ofrecer la lógica necesaria para definir y crear estas pruebas.

2.4. Material genérico

2.4.1. MagicDraw

MagicDraw es una herramienta utilizada para el modelado de aplicaciones usando UML. Durante el ciclo de vida del proyecto se ha utilizado principalmente en la fase de diseño para detallar los distintos casos de uso de la aplicación, diagramas de componentes y diagramas de clase.

2.4.2. Apache Tomcat

Esta herramienta, integrada con el propio framework de Spring, permite desplegar aplicaciones sobre un servidor de forma que sean accesibles por un cliente. Utilizada principalmente para el despliegue del servicio REST.

2.5 Metodología de desarrollo software

La metodología de trabajo seguida durante el desarrollo de la aplicación web ha sido basada en una metodología de desarrollo software incremental. Esta consiste en dividir la organización del trabajo en fases, imponiendo que la última fase corresponda con la finalización del desarrollo del software a implementar. Cada fase corresponde con nuevas funcionalidades o nuevos módulos de la aplicación; o con algún requisito, funcional o no funcional, de ésta.

Al final del desarrollo de cada fase se debe comprobar el correcto funcionamiento de estos nuevos módulos. Es decir, por así decirlo, se realizan pruebas de integración a estos bloques de código respecto al resto de la aplicación con la finalidad de que a cada nuevo módulo añadido se le pueda garantizar consistencia con el resto de la aplicación.

Es de suma importancia por parte del desarrollador, tener muy claro qué va a desarrollar y cómo va a implementar estas nuevas funciones para que no ocasionen conflictos con futuros bloques de código a añadir en la aplicación.

Esta metodología presenta varias ventajas como:

- Proporciona una visibilidad del producto desde primera fase, debido a que, en la integración de ésta, ya se implementa la aplicación con una o varias funcionalidades parciales.
- Obliga al desarrollador a realizar una planificación cuidadosa y de cierta calidad con el fin de no ocasionar conflictos en futuras integraciones de módulos.
- Al desarrollar en base a fases o bloques "pequeños", el desarrollador visualiza en todo momento qué está desarrollando, para qué y por qué. La finalidad de cada fase, es decir, la incorporación de una o varias funciones, permiten esta simplificación del desarrollo de un módulo.
- Se detecta con mayor precisión y agilidad los nuevos errores del desarrollo debido al desarrollo de bloques "pequeños" de código. Es decir, se incrementa la calidad del código desarrollado debido a que se realizan tests o pruebas de código individualmente a las funciones añadidas en cada fase. Por ejemplo, si se desarrolla en una fase la autentificación de usuarios de la aplicación web y en otra cualquier otra función del sistema, siguiendo esta metodología debería probar primero la autentificación. Posteriormente, desarrollar e implementar la otra funcionalidad, volver a autentificarme y probar dicha función. Este caso, implicaría que se estaría testeando dos veces, en lugar de una vez, la autentificación del usuario en el sistema lo que se puede traducir como una mayor calidad de código.

Por otra parte, esta metodología presenta una serie de inconvenientes como que implica una mayor dedicación al diseño de cada módulo, por parte del desarrollador, para entender cómo se ha de comportar cada módulo a implementar y cómo debe interactuar con otros módulos. Si no se cumpliese esto, podría ocasionar que el código desarrollado en una fase del desarrollo no tenga sentido o coherencia con el código de otra fase, lo que implicaría una modificación de este módulo y la realización, nuevamente, de tests de integración en la fase o fases afectadas.

3. Especificación y análisis de requisitos

En este apartado se describe el listado total de requisitos, tanto funcionales como no funcionales, que se han definido para la aplicación. También se especifican y se detallan los casos de uso que puede contemplar cualquier tipo de usuario del sistema.

Para la toma de requisitos, se han tenido presentes dos factores principalmente:

- El feedback recibido por parte de dos empleados de la cementera, mediante la realización de numerosas reuniones. Con éstas, el desarrollador ha entendido la lógica de negocio de la empresa, ha conocido la necesidad que tiene ésta y ha sido informado de la situación física que tienen los empleados, esto es, la disposición de uno o más ordenadores con conexión a Internet en el lugar de trabajo de los empleados.
- El criterio técnico del desarrollador a la hora de determinar qué requisitos se pueden satisfacer y cuáles no. Cabe destacar que los participantes en las reuniones mencionadas recientemente, no tienen conocimientos sobre el desarrollo software de aplicaciones web. Por consiguiente, se pretende satisfacer todas las necesidades posibles sin que supongan un exceso en el alcance de la aplicación.

3.1. Especificación de requisitos funcionales

Los requisitos funcionales son aquellas especificaciones de las funciones o actividades que debe implementar el sistema. Es decir, representan el comportamiento que ha de tener la aplicación independientemente de cómo se haya implementado.

A continuación, en la Tabla 1 se muestra el listado de requisitos funcionales que se han tomado, cada uno de ellos descrito mediante un identificador único y una breve descripción.

Tabla 1: Requisitos funcionales de la aplicación.

Identificador	Descripción
RF1	Cualquier usuario debe poder gestionar su propia cuenta.
RF1.1	Cualquier usuario debe poder registrarse en el sistema. Para ello deberá facilitar los siguientes datos personales: email, contraseña, DNI, nombre, primer apellido, segundo apellido y rol.
RF1.2	Cualquier usuario registrado en el sistema debe poder identificarse/cerrar sesión en la aplicación. Para ello, en el caso de iniciar sesión, se deben acreditar las credenciales del usuario.
RF2	Cualquier usuario registrado debe poder gestionar los bonos de descargo de la aplicación.
RF2.1	Cualquier usuario registrado debe poder tener acceso a cualquier bono de descargo del sistema, ya sean bonos pasados, en curso o futuros.
RF2.1.1	Cualquier usuario registrado debe poder consultar el histórico de bonos de descargo de la aplicación.
RF2.1.2	Cualquier usuario registrado debe poder aplicar filtros de búsqueda con el fin de visualizar un listado de bonos en función de los siguientes parámetros: identificador de un bono específico, nombre del usuario con rol “Solicitante” que haya intervenido como solicitante, nombre y número de la máquina que haya sido descargada o utilizada, y fecha correspondiente a la solicitud de descargo.
RF2.2	Cualquier usuario registrado, cuyo rol sea “Solicitante”, debe poder crear bonos de descargo. Para ello debe especificar la máquina a utilizar o descargar,

	la fecha en la que se desea que la máquina esté descargada para su uso, las instrucciones a seguir para el descargo de la máquina (si fuese necesario) y una breve descripción o resumen del trabajo a realizar.
RF2.3	Cualquier usuario registrado, cuyo rol sea "Solicitante", debe poder modificar el estado del bono una vez haya finalizado por completo su trabajo.
RF2.4	Cualquier usuario registrado, cuyo rol sea "Operador", debe poder modificar el estado del bono cuando quiera enviar a instrumentación una solicitud para quitar/poner tensión sobre una máquina específica.
RF2.5	Cualquier usuario registrado, cuyo rol sea "Instrumentista", debe poder modificar el estado del bono cuando quiera confirmar la descarga de tensión sobre una máquina específica.
RF2.6	Cualquier usuario registrado, cuyo rol sea "Instrumentista", debe poder modificar el estado del bono cuando quiera confirmar la puesta de tensión sobre una máquina específica.
RF2.6.1	En caso de que exista un mínimo de dos o más bonos que impliquen un trabajo simultáneo sobre la misma máquina, no se podrá poner tensión en dicha máquina, y por lo tanto, el usuario no podrá modificar el estado del bono.
RF2.6.2	En caso de que exista un mínimo de dos o más bonos que impliquen un trabajo simultáneo sobre la misma máquina, una vez finalicen todos los procesos de descarga el usuario podrá poner tensión en dicha máquina, y por lo tanto, modificar el estado del bono.
RF2.7	Cualquier usuario registrado debe firmar digitalmente tras realizar cualquier operación de modificación de estado de un bono de descargo.

3.2. Especificación de requisitos no funcionales

Los requisitos no funcionales son aquellas especificaciones del sistema que no están asociadas al comportamiento del sistema, pero que pueden influir a la hora de implementar éste. Es decir, son aquellas características de la aplicación que se han de tener en cuenta durante la fase de diseño y que imponen cierta calidad al producto.

A continuación, en la Tabla 2 se muestra el listado de requisitos no funcionales que se han tomado. Éstos tienen un identificador único, una breve descripción y una categoría sobre la que se clasifican.

Tabla 2: Requisitos no funcionales de la aplicación.

Identificador	Descripción	Categoría
RNF1	La aplicación debe ser soportada por los navegadores web más utilizados en la actualidad: Google Chrome, Opera, Internet Explorer y Mozilla Firefox.	Portabilidad
RNF2	El sistema debe soportar un tráfico constante de peticiones diariamente.	Capacidad
RNF3	La aplicación debe ser estar escrita completamente en español.	Usabilidad
RNF4	La aplicación debe ser intuitiva y fácil de usar para cualquier usuario con el fin de facilitar el uso de ésta.	Usabilidad
RNF5	El sistema debe validar la inserción de datos correctos con el fin de que cualquier dato almacenado en el sistema sea válido.	Usabilidad

3.3. Especificación de casos de uso

La especificación de casos de uso es una técnica comúnmente utilizada en el ámbito del desarrollo software para detallar las interacciones entre una aplicación y los diferentes usuarios que la utilicen. De esta manera, el objetivo principal de esta técnica es aportar información sobre qué funcionalidades pueden desempeñar cada tipo de usuario del sistema.

Como en esta aplicación se encuentran tres tipos de empleados, se ha segmentado el diagrama de casos de uso en tres partes para diferenciar claramente las operaciones que puede desempeñar cada tipo de usuario.

En la figura 2 se muestran los casos de uso correspondientes a la gestión de usuarios, referentes a usuarios no registrados en el sistema y usuarios registrados que se encuentran con la sesión cerrada, es decir, que no están autenticados en la aplicación.

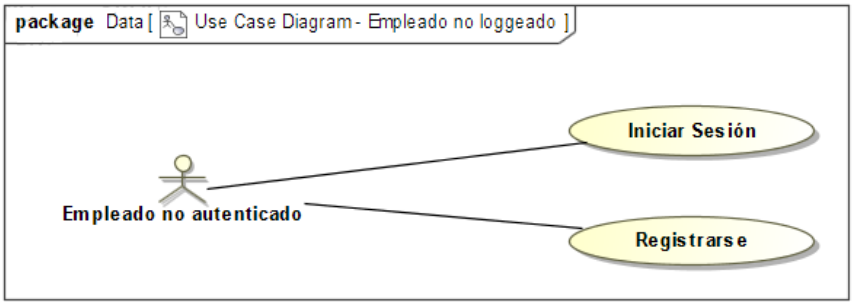

Figura 2. Diagrama de casos de uso para empleados no autenticados.

En la figura 3, se muestran los casos de uso específicos de cada tipo de usuario del sistema referidos a la gestión de los bonos de descargo. Como se puede observar, cada usuario interviene exclusivamente en ciertas partes del trámite del bono.

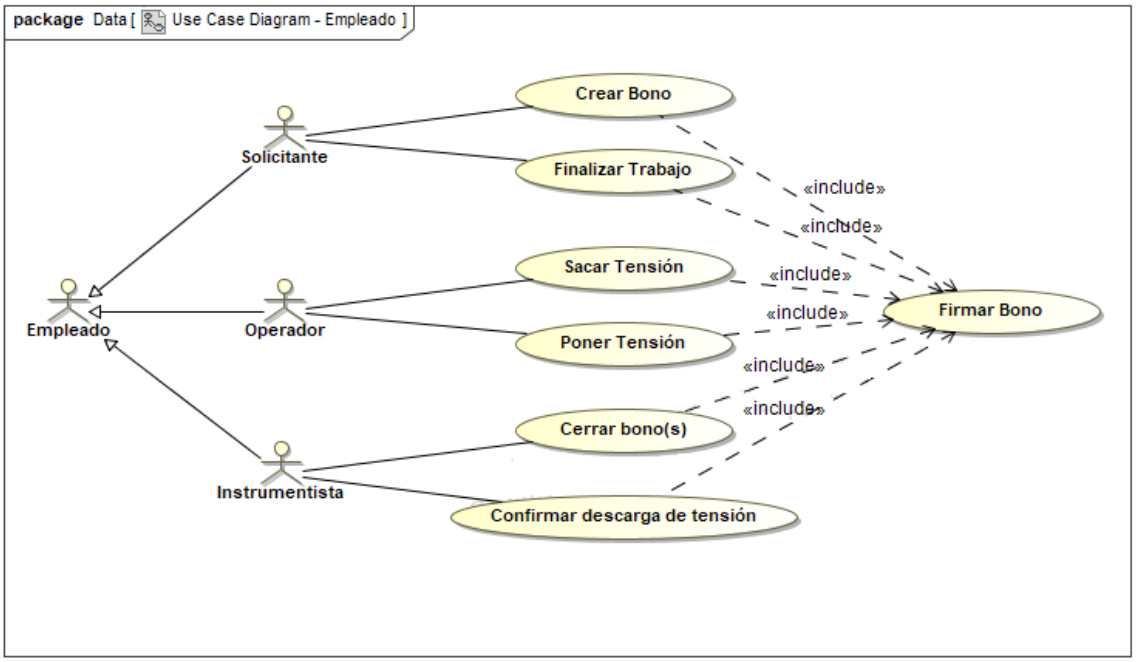

Figura 3. Diagrama de casos de uso específicos para empleados autenticados.

En la figura 4 se muestran los casos de uso comunes a todos los usuarios registrados en el sistema independientemente del rol que tengan asociado. Se encuentran tanto casos de uso referentes a la gestión de bonos de descargo como a la gestión de usuarios.

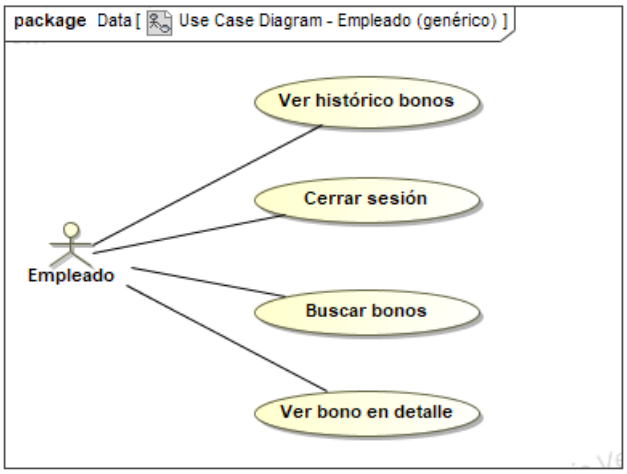

Figura 4. Diagrama de casos de uso genéricos para empleados autenticados.

3.4. Especificación de casos de uso en detalle

Una vez especificadas a alto nivel las diferentes interacciones que un usuario puede tener en la aplicación se procede a explicar con más detalle como son estas interacciones entre el usuario y el sistema. Para ello, principalmente hay que detallar el flujo de la secuencia de acciones que realiza el usuario con el sistema, qué excepciones se pueden producir, qué circunstancias se deben dar para que se produzcan y qué estado se espera de la aplicación tras finalizar el flujo del caso en una secuencia correcta.

A continuación, en la Tabla 3, se muestra la plantilla que se va a utilizar para describir los casos de uso, en la que se explica cada campo utilizado para la detallar el caso de uso. Posteriormente, desde la Tabla 4 hasta la Tabla 12 se especifican algunos de los casos de uso de la aplicación.

Tabla 3: Plantilla de un caso de uso.

CU-XX	Identificador del caso de uso.
Versión	Versión de la especificación del caso de uso.
Dependencias	Dependencias de la lista de requisitos necesarios para el caso de uso.
Precondiciones	Requisitos previos en los que se debe encontrar el sistema para poder ejecutar el caso de uso.
Descripción	Idea principal del caso de uso.
Secuencia normal	Flujo de acciones que se deben desarrollar en orden para alcanzar el estado esperado satisfactoriamente. Cada paso tiene asociado un número identificativo que determina el orden del flujo.
Postcondiciones	Estado del sistema esperado tras una ejecución correcta del caso de uso.
Excepciones	Flujo de acciones alternativo a la secuencia normal del caso de uso. En este flujo, cada acción está asociada a un paso de la secuencia habitual.
Comentarios	Comentarios del desarrollador.

Gestión de usuarios

Tabla 4: Caso de uso Registrarse.

CU-01	Registrarse		
Versión	1.0		
Dependencias	RF1, RF1.1		
Precondiciones	El usuario no ha iniciado sesión y no tiene un usuario registrado en el sistema.		
Descripción	El sistema debe confirmar que el usuario proporciona unos datos de usuario correctos y da soporte a un proceso de registro por el cual se almacena en el sistema un usuario con unas credenciales y unos datos propios de éste.		
Secuencia normal	Paso	Acción	
	1	El usuario hace click en “Regístrese”.	
	2	El usuario rellena los campos “Email”, “Contraseña”, “Repita la contraseña”, “Nombre”, “Primer apellido”, “Segundo apellido”, “DNI” y “Rol”.	
	3	El usuario hace click en “Registrarse”.	
	4	El sistema almacena el nuevo usuario en el sistema.	
	5	El sistema redirecciona la vista a la pantalla de inicio de sesión.	
Postcondiciones	Se ha almacenado en el sistema un usuario con unas credenciales y unos datos propios de éste.		
Excepciones	Paso	Acción	
	2	Si el usuario ha introducido *datos incorrectos**.	
		E.1.1	El sistema informa de la inserción de datos incorrectos o inapropiados.
		E.1.2	El sistema retorna al estado inicial del caso de uso.
	2	Si el usuario ha introducido un correo electrónico repetido, es decir, un correo existente en la cuenta de otro usuario registrado.	
		E.2.1	El sistema informa de la inserción de email repetido.
		E.2.2	El sistema retorna al estado inicial del caso de uso.
Comentarios	*Datos incorrectos: a. Email inválido. b. DNI inválido (8 números + una letra) c. “Contraseña” y “Repetir contraseña” no coinciden. d. Campos con más de 25 caracteres.		

Gestión de bonos

Tabla 5: Caso de uso Ver bono en detalle.

CU-02	Ver bono en detalle	
Versión	1.0	
Dependencias	RF2, RF2.1	
Precondiciones	El usuario ha iniciado sesión y se encuentra en la pantalla principal de la aplicación, donde se muestra la lista general de bonos.	
Descripción	El sistema muestra el detalle de un bono seleccionado.	
Secuencia normal	Paso	Acción
	1	El usuario selecciona un bono de la lista.
	2	El sistema muestra la vista en detalle del bono seleccionado con todos sus campos.
Postcondiciones	El sistema muestra los datos del bono seleccionado.	

Tabla 6: Caso de uso Buscar bonos.

CU-03	Buscar bonos		
Versión	1.0		
Dependencias	RF2, RF2.1, RF2.1.2		
Precondiciones	El usuario ha iniciado sesión y se encuentra en la pantalla principal de la aplicación, donde se muestra la lista general de bonos.		
Descripción	El sistema actualiza la lista principal de bonos de descargo en función de los filtros introducidos.		
Secuencia normal	Paso	Acción	
	1	El usuario hace click en “Filtrar”.	
	2	El usuario selecciona un filtro a introducir.	
	3	El usuario introduce un valor en función del tipo de filtro elegido.	
	4	El usuario hace click en “Ok”.	
	5	El sistema actualiza la lista de bonos según el filtro introducido. En caso de existir algún filtro previo, el sistema adhiere el nuevo filtro y mantiene los filtros aplicados previamente.	
Postcondiciones	El sistema muestra la lista de bonos filtrada en función de los parámetros de búsqueda introducidos.		
Excepciones	Paso	Acción	
	4	Si el usuario ha introducido algún filtro que no corresponde con dicho parámetro de ningún bono registrado en el sistema.	
		E.1.1	El sistema muestra una lista vacía.
Comentarios	En caso de querer borrar filtros, se debe hacer click encima del filtro. Tras esto, éste se elimina y se rehace la búsqueda en función de los filtros restantes en caso de que quede alguno.		

Tabla 7: Caso de uso Crear bono.

CU-04	Crear bono		
Versión	1.0		
Dependencias	RF2, RF2.2, RF2.7		
Precondiciones	El usuario, de rol “Solicitante” ha iniciado sesión y se encuentra en la pantalla principal de la aplicación.		
Descripción	El sistema crea y almacena un nuevo bono.		
Secuencia normal	Paso	Acción	
	1	El usuario hace click en “Crear Bono”.	
	2	El usuario rellena los campos “Fecha del comienzo de utilización”, “Instrucciones de descargo”, “Trabajo a realizar” y “Nombre de la máquina”.	
	3	El usuario hace click en “Crear Bono”.	
	4	El usuario hace click en “Firmar” e introduce su firma digital.	
	5	El sistema almacena el nuevo bono creado.	
Postcondiciones	El nuevo bono se ha almacenado y se ve reflejado en la lista.		
Excepciones	Paso	Acción	
	4	Si el usuario ha introducido *datos incorrectos**.	
		E.1.1	El sistema informa de la inserción de datos incorrectos o inapropiados.
		E.1.2	El sistema retorna al estado inicial del caso de uso.
Comentarios	*Datos incorrectos:		

	a. Fecha inválida: Fecha anterior a la fecha actual, o formato incorrecto. b. Máquina inexistente (no aparece en el desplegable). c. Campos de “Fecha del comienzo de utilización” y “Nombre de la máquina” en blanco. (Los otros dos son opcionales) d. Si no se ha insertado ninguna firma.

Tabla 8: Caso de uso Sacar tensión.

CU-05	Sacar tensión	
Versión	1.0	
Dependencias	RF2, RF2.4, RF2.7	
Precondiciones	El usuario de rol “Operador” ha iniciado sesión y se encuentra en la vista en detalle de un bono cuyo estado es el siguiente: solo los campos de la creación han sido rellenados. Además, éste previsualiza la opción “Sacar Tensión”.	
Descripción	El sistema actualiza el bono en función del estado de éste.	
Secuencia normal	Paso	Acción
	1	El usuario hace click en “Sacar Tensión” e introduce su firma digital.
	2	El sistema actualiza el bono, rellenando automáticamente los campos referentes a la fecha y hora del acuerdo del descargo, y a los datos identificativos del operador que realiza la operación (nombre, primer apellido y firma).
Postcondiciones	El bono se ha actualizado. Se muestra actualizado en la vista de detalle de éste y desaparece la opción “Sacar Tensión”.	

Tabla 9: Caso de uso Confirmar la descarga de tensión.

CU-06	Confirmar la descarga de tensión	
Versión	1.0	
Dependencias	RF2, RF2.5, RF2.7	
Precondiciones	El usuario de rol “Instrumentista” ha iniciado sesión y se encuentra en la vista en detalle de un bono cuyo estado es el siguiente: solo los campos de la creación y de descarga de tensión han sido rellenados. Además, éste previsualiza la opción “Confirmar la descarga de tensión”.	
Descripción	El sistema actualiza el bono en función del estado de éste.	
Secuencia normal	Paso	Acción
	1	El usuario hace click en “Confirmar la descarga de tensión” e introduce su firma digital.
	2	El sistema actualiza el bono, rellenando automáticamente los campos referentes a la fecha y hora de la confirmación del descargo, y a los datos identificativos del instrumentista que realiza la operación (nombre, primer apellido y firma).
Postcondiciones	El bono se ha actualizado. Se muestra actualizado en la vista de detalle de éste y desaparece la opción “Confirmar la descarga de tensión”.	

Tabla 10: Caso de uso Finalizar Trabajo.

CU-7	Finalizar Trabajo
Versión	1.0
Dependencias	RF2, RF2.3, RF2.7

Precondiciones	El usuario de rol “Solicitante” ha iniciado sesión y se encuentra en la vista en detalle de un bono cuyo estado es el siguiente: todos los campos necesarios para la descarga de tensión de una máquina han sido rellenados. Además, éste previsualiza la opción “Finalizar Trabajo”.	
Descripción	El sistema debe actualizar el bono en función del estado de éste.	
Secuencia normal	Paso	Acción
	1	El usuario hace click en “Finalizar Trabajo” e introduce su firma digital.
	2	El sistema actualiza el bono, rellenando automáticamente los campos referentes a la fecha y hora de la finalización del trabajo, y a los datos identificativos del solicitante que realiza la operación (nombre, primer apellido y firma).
Postcondiciones	El bono se ha actualizado. Se muestra actualizado en la vista de detalle de éste y desaparece la opción “Finalizar Trabajo”.	

Tabla 11: Caso de uso Poner Tensión.

CU-08	Poner Tensión	
Versión	1.0	
Dependencias	RF2, RF2.4, RF2.7	
Precondiciones	El usuario de rol “Operador” ha iniciado sesión y se encuentra en la vista en detalle de un bono cuyo estado es el siguiente: el trabajo ya ha finalizado y están rellenados los campos de acuerdo a dicho estado. Además, éste previsualiza la opción “Poner Tensión”.	
Descripción	El sistema debe actualizar el bono en función del estado de éste.	
Secuencia normal	Paso	Acción
	1	El usuario hace click en “Poner Tensión” e introduce su firma digital.
	2	El sistema actualiza el bono, rellenando automáticamente los campos referentes a la fecha y hora de la orden de poner tensión, y a los datos identificativos del operador que realiza la operación (nombre, primer apellido y firma).
Postcondiciones	El bono se ha actualizado. Se muestra actualizado en la vista de detalle de éste y desaparece la opción “Poner Tensión”.	

Tabla 12: Caso de uso Cerrar Bono(s).

CU-09	Cerrar Bono(s)	
Versión	1.0	
Dependencias	RF2, RF2.6, RF2.7	
Precondiciones	El usuario de rol “Instrumentista” ha iniciado sesión, ha accedido a un bono y se encuentra en su vista detalle, donde se muestra toda la información de dicho bono.	
Descripción	El sistema actualiza el bono en función del estado de éste.	
Secuencia normal	Paso	Acción
	1	El usuario hace click en “Cerrar Bonos” e introduce su firma digital.
	2	El sistema actualiza todos los bonos afectados, en caso de que haya más de uno, rellenando automáticamente los campos referentes a la fecha y hora de la orden de poner tensión, y a los datos

<table>
<tr><td rowspan="2"></td><td></td><td colspan="2">identificativos del operador que realiza la operación (nombre, primer apellido y firma).</td></tr>
<tr><td>3</td><td colspan="2">El sistema muestra un listado de bonos afectados.</td></tr>
<tr><td>Postcondiciones</td><td colspan="3">El bono se ha actualizado. Se muestra actualizado en la vista de detalle de éste y desaparece la opción “Poner Tensión”.</td></tr>
<tr><td rowspan="4">Excepciones</td><td>Paso</td><td colspan="2">Acción</td></tr>
<tr><td rowspan="3">1</td><td colspan="2">Si el sistema detecta que hay otros bonos en curso que especifiquen la misma máquina que la indicada por este bono y que impliquen que hay otro empleado usando dicha máquina.</td></tr>
<tr><td>E.1.1</td><td>El sistema cancela la operación.</td></tr>
<tr><td>E.1.2.</td><td>El sistema muestra un mensaje de error indicando la imposibilidad de cerrar el bono, y/o los bonos afectados.</td></tr>
</table>

4. Diseño Software

Una vez especificados y detallados los requisitos que debe satisfacer la aplicación, se puede comenzar a realizar el diseño del sistema. Este proceso es de vital importancia dentro del ciclo de vida del desarrollo de un sistema software, ya que en éste se detalla la "infraestructura" necesaria para satisfacer los requisitos. El diseño se divide dos partes: diseño arquitectónico y diseño detallado.

4.1. Diseño arquitectónico

El diseño arquitectónico de un sistema software se debe adaptar a los requisitos, funcionales y no funcionales, establecidos en la fase previa de análisis, por lo que la toma de requisitos previa influye directamente en el diseño utilizado. En este caso, se ha optado por una arquitectura en tres capas: capa de presentación, capa de negocio y capa de persistencia. A continuación, se explicará el fin y la arquitectura propia de cada una de estas capas.

4.1.1. Diseño de la interfaz web (capa de presentación)

La capa de presentación es la encargada de servir como intermediaria entre el usuario y la lógica de negocio. En ésta se encuentran todos los elementos que constituyen la interfaz web, es decir, todas las vistas, hojas de estilo, componentes, interfaces (modelo), servicios, etc. Estas interfaces de la aplicación web permiten la interacción entre el usuario y el sistema y muestran la información almacenada en el sistema.

Cómo se mencionó previamente, para el desarrollo de la interfaz web se ha usado Angular, el cual consiste en el renderizado de componentes desde una vista principal. En la arquitectura de las aplicaciones web Angular se pueden encontrar los siguientes tipos de elementos:

- Componentes de Angular: Cada componente Angular es un conjunto formado por una hoja de estilos (CSS), una plantilla (template) HTML en la que renderizar los elementos que se requieran y un archivo JS o TS con el que gestionar las interacciones entre la vista (plantilla HTML) y la aplicación.
- Modelo: Es el conjunto de interfaces definidas dentro de la propia aplicación web para definir los tipos de datos que se manejan. Pese a que Angular permite la no definición de tipos de objetos, siempre es aconsejable tener este tipo de archivos para definir conjuntos de datos de forma más rápida y consistente.
- Servicios: Es el conjunto de archivos internos de la aplicación web necesarios para hacer peticiones al servicio REST, y por lo tanto poder tratar con la información que éste proporciona.

En la Figura 5 se muestra el diagrama de componentes de la capa de presentación, donde se pueden apreciar los componentes y servicios concretos que se han definido para esta aplicación, cómo se agrupan y cómo se relacionan. Posteriormente, en la Figura 6 se especifican en detalle las interfaces de los servicios definidos cuyas operaciones mapean las diferentes peticiones que se hacen desde la capa web a los recursos del servicio. Las clases concretas que forman el modelo se mostrarán en la sección de diseño detallado.

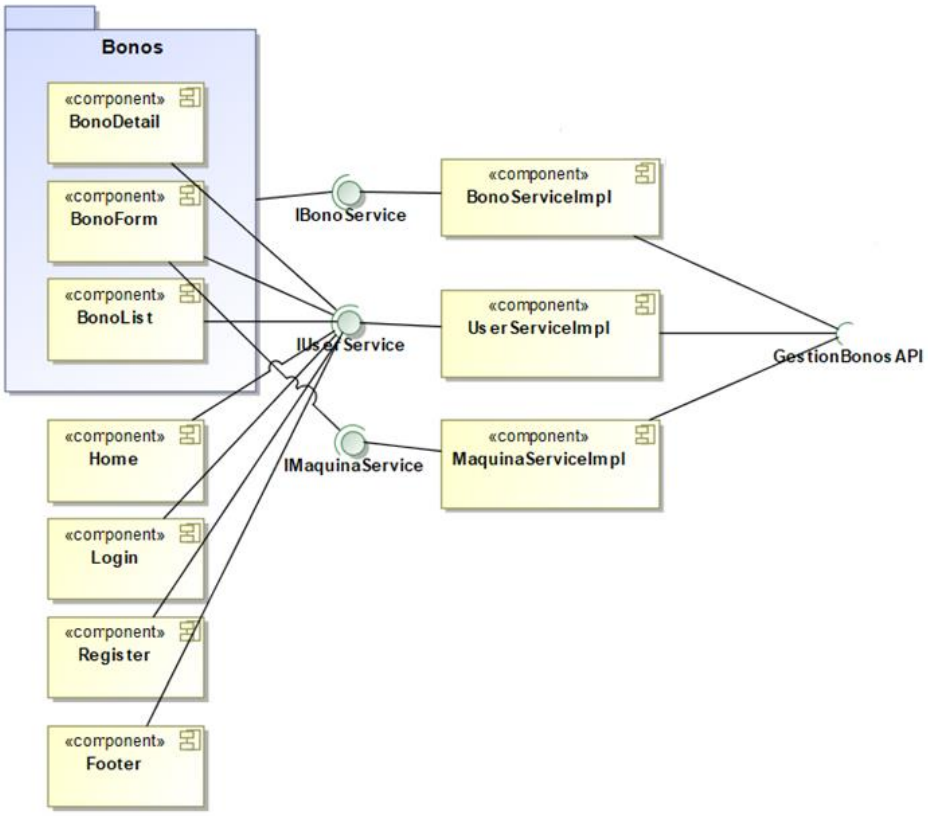

Figura 5. Diagrama de componentes de la capa de presentación.

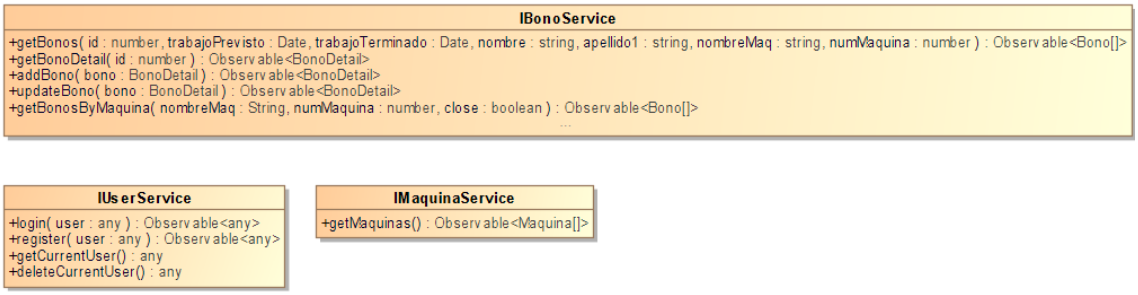

Figura 6. Especificación de las operaciones de los servicios de la interfaz web.

4.1.2. Diseño del servicio REST (capa de negocio)

Para poder gestionar las interacciones del usuario con la capa de persistencia, es decir, con la base de datos, es necesario definir un servicio REST que actúe como intermediario entre estas capas para facilitar la gestión de peticiones desde el frontend facilitando el acceso a los recursos necesarios.

En base a la fase de análisis de requisitos previa, se identificaron tres recursos a ofrecer en el servicio. Éstos son los siguientes:

- Bono: recurso que almacena toda la información de un bono de descargo de la aplicación. Es el recurso principal del servicio.
- Usuario: recurso que almacena toda la información necesaria de cada usuario del sistema.
- Máquina: recurso que almacena los datos identificativos de cada máquina del sistema.

El servicio debe ofrecer un listado de operaciones sobre los recursos especificados accesibles mediante una URI. Los tipos de operaciones principales son las siguientes:

- GET: operación con la que se obtiene la información de un recurso alojado en una URI.
- POST: operación con la que se generan nuevos recursos. También puede ser utilizado como un GET cuando es necesario incorporar en la petición un dato complejo (se hace uso en este caso del body de la petición) que no puede ser alojado en la cabecera de la operación.
- PUT: operación, que al igual que el POST, permite generar nuevos recursos y alojarles en una URI. También se utiliza este tipo de operaciones para modificar recursos existentes.

Además, las respuestas esperadas por cada operación deben estar definidas para saber qué tipo de dato deben aceptar de vuelta los servicios de la capa de presentación. Generalmente, la capa de presentación no necesita toda la información de los recursos, por lo que se generan y se devuelven DTO (Data Transfer Object) [5]. Por ejemplo, cuando un usuario se autentica, la capa de presentación no debe alojar información sobre la contraseña del usuario. Sin embargo, la base de datos sí debe alojar dicha información para permitir su verificación en futuros accesos por parte del usuario.

También se deben definir los códigos de respuesta HTTP de cada operación con los que conocer el estado de cada petición a cada recurso. Entre los más habituales, se encuentran los siguientes:

- 200: Código de éxito retornado en caso de que la solicitud se haya realizado satisfactoriamente.
- 201: Código de éxito retornado en caso de que se haya realizado una creación de un recurso satisfactoriamente.
- 403: Código de error retornado en caso de que no se tenga permisos de acceso a un recurso y se esté intentando acceder a éste.
- 404: Código de error retornado en caso de que no exista o no se encuentre un recurso específico y se esté intentado obtener éste.
- 500: Código de error retornado en caso de que haya habido algún problema interno en el servicio REST que implica la no finalización de la petición.

A continuación, en la Tabla 13, se puede visualizar el listado total de operaciones que proporciona el servicio REST para cada uno de sus recursos.

Tabla 13. Lista de recursos de la API REST.

Recurso	Path	Respuesta	Operación	Código de respuesta HTTP
Bonos	/bonos	List<BonoDTO>	GET QueryParams: id, trabajoPrevisto, nombre, apellido1, nombreMaq, numMaquina POST	200, 403, 500 201, 403, 500

Bonos de una máquina	/bonos/{nombreMaq} /{numMaquina}	List<BonoDTO>	GET QueryParams: close	200, 403, 404, 500
Bono	/bonos/{id}	BonoDetailDTO BonoDetailDTO	GET PUT	200, 403, 404, 500 200, 403, 500
Máquinas	/maquinas	List<Maquina>	GET	200, 403, 500
Usuarios	/users	UsuarioDTO	POST	201, 403, 500
Usuario	/users/{email}/token	UsuarioDTO	POST	200, 403, 404, 500

La arquitectura del servicio REST sigue, a su vez, una arquitectura de tres capas:

- Controladores: Capa que aloja las operaciones mencionadas previamente para cada recurso REST.
- Servicios: Son los intermediarios entre el controlador de un recurso y su respectivo repositorio. Pueden incorporar lógica de negocio compleja.
- Repositorios: Es la capa encargada de enlazar el servicio con la capa de persistencia.

En las figuras 7 y 8 se puede visualizar la arquitectura descrita para este servicio, con los distintos componentes, servicios y repositorios, así como las operaciones que proporcionan los servicios y repositorios.

4.2. Diseño detallado

El diseño detallado del sistema pretende dar una idea concisa de cómo se comportan las distintas clases de programación, con el fin de facilitar su futura implementación.

En la figura 9, se muestra el diagrama de clases de la aplicación en el que se especifican los datos, con sus tipos, que contiene cada clase utilizada en la aplicación.

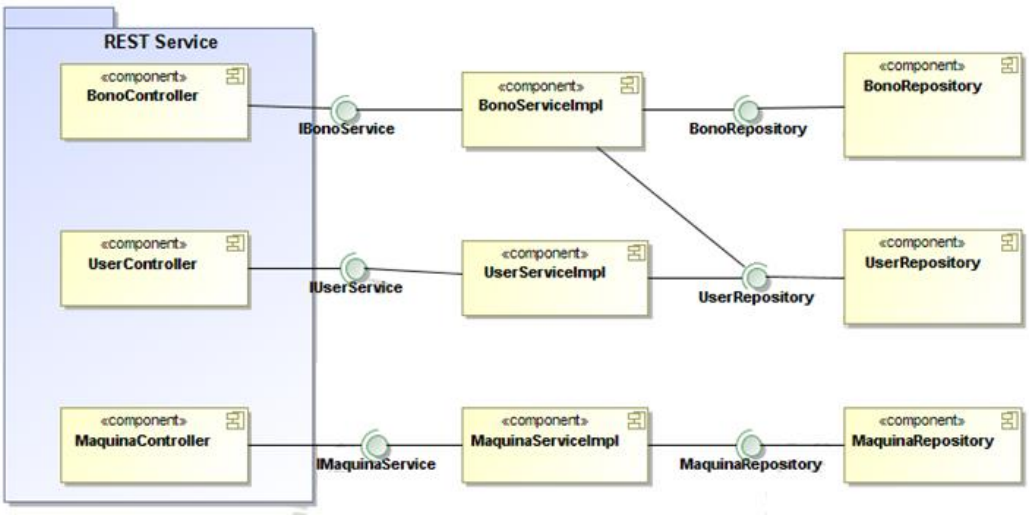

Figura 7. Diagrama de componentes de la capa de negocio.

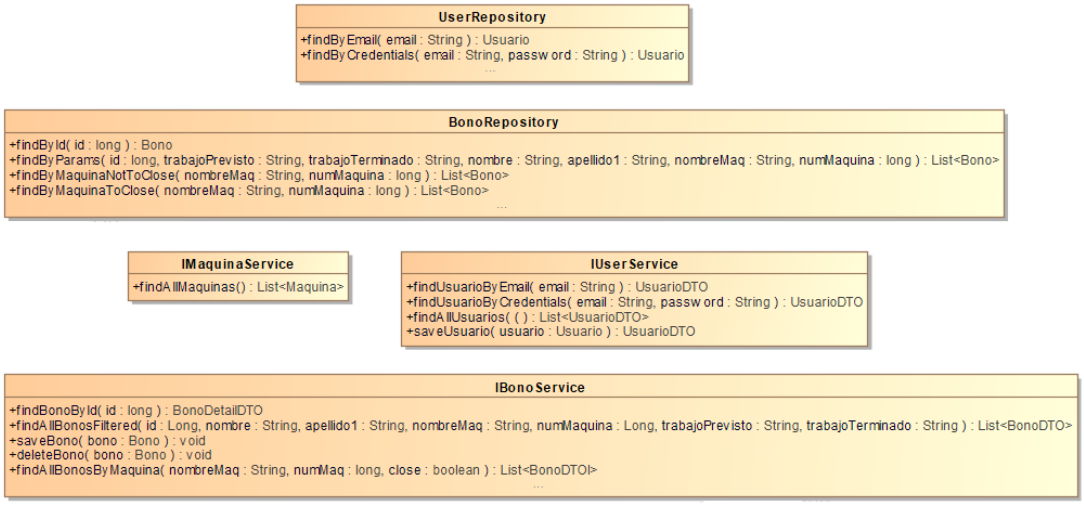

Figura 8. Especificación de las operaciones de los servicios de la capa de negocio.

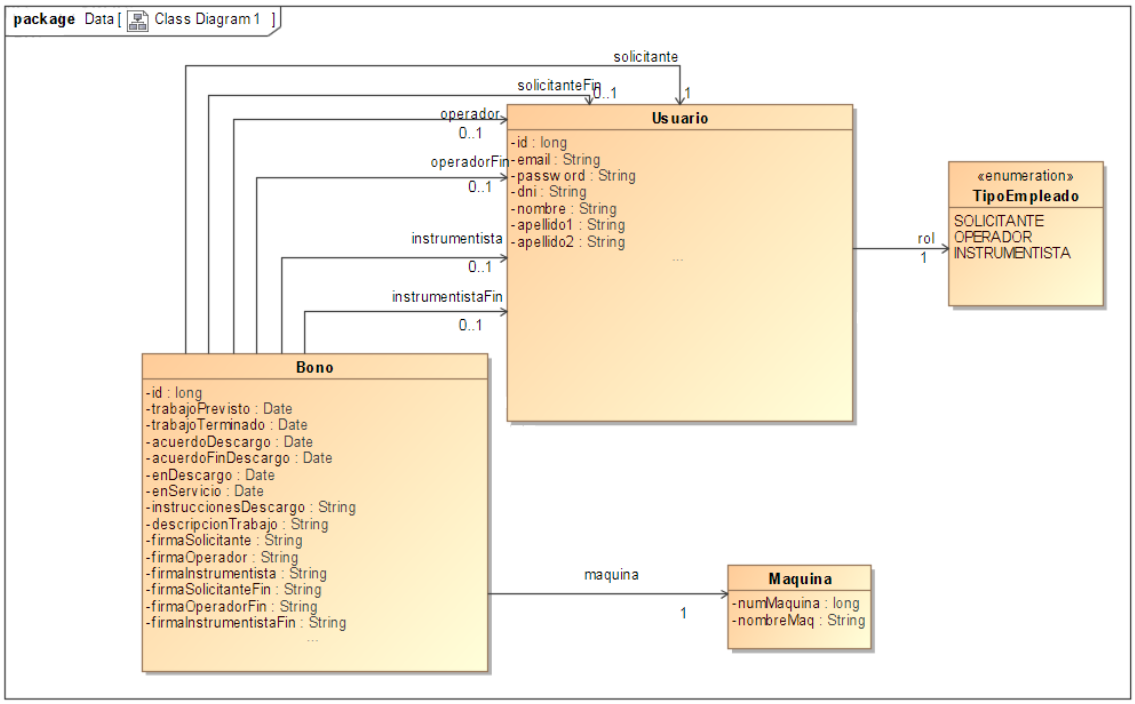

Figura 9. Diagrama de clases.

5. Implementación

Una vez se han finalizado las fases de análisis y diseño, ya se tiene la información necesaria para comenzar la fase de desarrollo del proyecto. Este apartado se divide en cuatro secciones con las que se pretende dar una idea un poco más en detalle de cómo se han implementados ciertos bloques de la aplicación

5.1. Configuración del entorno de desarrollo

En los siguientes subapartados se aclararán los pasos seguidos para configurar el entorno de desarrollo.

5.1.1. Generación del servicio REST con SpringBoot

Para la generación del proyecto del servicio REST, se hizo uso de la herramienta Spring Initializr la cual facilita enormemente la generación de proyectos de Spring. Con esta herramienta se especificaron los parámetros del proyecto de la capa de negocio como su nombre, la versión de Spring Boot, la versión de Java a usar o las dependencias a utilizar. Toda esta información se aloja automáticamente en el descriptor pom.xml del proyecto. Una vez generado el proyecto, ya se puede importar a Eclipse como proyecto Maven y comenzar a trabajar con él.

Para desplegar el servicio, se genera el archivo .jar y se ejecuta, lo cual lanza un Apache Tomcat embebido, en el que se despliega el servicio. En el archivo general de configuración del servicio se pueden definir algunas propiedades del despliegue como la dirección y el puerto dónde se debe desplegar el servicio, como se puede apreciar en la figura 10, en la que se especifica una ip y el puerto 8081. Por defecto, se utiliza el puerto 8080. Si se especifica como dirección “localhost” se despliega el servicio sobre un entorno local, por el cual los recursos de la API solo serían accesibles por el equipo que haya desplegado el servicio. Sin embargo, si se especifica una ip, la API estaría disponible para cualquier equipo que tenga acceso a dicha IP.

5.1.2. Configuración del driver de la base de datos y conexión con la API

Lo primero a tener en cuenta es la configuración de los ajustes de conexión en los que se especifican parámetros como el puerto y el nombre del servidor dónde se alojará la base de datos. Tras configurar el servidor, se debe configurar el driver del servidor al cual accederá la API para tener acceso a éste.

Una vez configurado el servidor y su respectivo driver, se puede proceder a crear la base de datos. Se optó por el uso de una base de datos MySQL debido a la facilidad de integración que tiene con el framework Spring. Para la creación de dicha base de datos, es suficiente con especificar un nombre debido a que la creación del Schema, y por lo tanto, la estructura de ésta se realizará mediante el ORM de la API.

Por último, cómo se puede observar en la figura 10, se ha de crear un archivo de configuración en la API para poder conectarse a la base de datos. Además, en dicho archivo se pueden especificar varios parámetros que influyen en el mapeado objeto-relacional, cómo “ddl-auto” con el que se especifica cómo ha de ser modificado el schema de la base de datos al inicializarla con Hibernate. En este caso, al ser “update”, se está especificando que solo se admitan adiciones o modificaciones en el schema.

```yaml
cors:
  prod:
    url: http://192.168.1.83:4200/
  local:
    url: http://localhost:4200/

server.address: 192.168.1.83 #localhost #192.168.1.83
server.port: 8081


spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/cementera_bbdd
    username: root
    password: 123456
  jpa:
    show-sql: true

    generate-ddl: true

    hibernate:
      ddl-auto: update
    properties:
      hibernate:
        dialect = org.hibernate.dialect.MySQL8Dialect
```

Figura 10. Application.yml.

5.1.3. Generación de la aplicación web

Los pasos seguidos para configurar la aplicación web se pueden resumir en especificar las opciones de "Angular Routing" y "CSS" tras ejecutar el comando "ng new *nombre del proyecto*", y en instalar las librerías necesarias mediante el comando "npm install *ruta librería*". Con la primera opción se especifica cómo se gestionará el routing de la aplicación, es decir, la forma en la que se harán las redirecciones a los templates o vistas de los componentes.

Para desplegar la aplicación web se necesita que se haya desplegado previamente la API REST ya que se sirve de dicho servicio para su funcionamiento. Si se desea desplegar localmente, la mejor opción es usar el comando "ng serve", el cual se sirve de los archivos en memoria para la construcción de la aplicación con lo que se pueden ver las modificaciones realizadas sobre el código en el mismo momento en el que se insertan. Sin embargo, si se quisiese desplegar bajo un entorno real, de producción, esa opción no sería válida. En este caso, la mejor opción es usar el comando "ng build" para generar archivos estáticos ejecutables por un servidor y desplegar dichos archivos, que por defecto se generan sobre el directorio "/dist" del mismo proyecto.

5.2. Implementación del servicio REST

El servicio REST desarrollado bajo el framework Spring es el encargado de definir el mapeado objeto relacional, con el que se crea el schema de la base de datos, y de ofrecer una serie de operaciones de acceso, eliminación, modificación o creación de los recursos que interactúan con dicha base de datos.

5.2.1. Implementación del ORM mediante JPA e Hibernate

Los frameworks JPA e Hibernate permiten crear el schema de la base de datos mediante el uso de anotaciones en las clases con las que se definen todas las tablas que tendrá dicha base de

datos, junto a sus respectivas columnas, relaciones, restricciones, etc. Las anotaciones más habituales, para las que se puede ver un ejemplo en la Figura 11, son las siguientes:

- @Entity: usada para definir una clase Java como una entidad, es decir, una clase que se mapea a una tabla de la base de datos sobre la que almacenar información según la estructura de dicha entidad. Se puede especificar el nombre de la tabla introduciéndolo en el argumento de la etiqueta.
- @Column: usada para asociar un atributo de la clase Java con una columna de la tabla. Permite ciertos parámetros en su argumento, como "nullable" para especificar si el atributo es obligatorio o no.
- @Id: usada para especificar la restricción de "Primary Key" o clave primaria sobre un atributo. Esta restricción implica unicidad y sirve como identificación de la entidad en la tabla a la que pertenezca.
- @GeneratedValue: usada para especificar el tipo de generación de ciertos campos en caso que se desee que sean autocalculados. En la figura 11, se puede apreciar cómo se especifica en el argumento el tipo de estrategia utilizado para el cálculo de dicho campo.
- @JoinColumn: usada para especificar relaciones entre entidades. Se le pueden añadir los mismos parámetros, en su argumento, que a la anotación @Column. En adición a esta etiqueta, se utilizan otros tipos de etiquetas como @ManyToOne, @OneToMany, @OneToOne o @ManyToMany para especificar el tipo de relación.
  En este caso concreto como cada bono tiene asociados seis usuarios, y es el propio bono el que debe almacenar la información del usuario asociado y no al revés, se optó por especificar una relación n:1, es decir, @ManyToOne, como se puede observar en la Figura 11. Además, como no siempre es necesaria la información de estos usuarios, se utilizó carga retardada (fetch = Lazy). Este tipo de cargado o búsqueda implica que se cargue la información de ese campo únicamente cuando se requiera, es decir, cuando se llame al getter de dicho campo.
- @Lob: utilizada en adición a la etiqueta @Column para especificar que el campo que se vaya a almacenar sea de tipo Large para el tipo de dato que sea. Por ejemplo, si se usa junto a un String, se persistirá en la base de datos una columna de tipo Text. Si se usa junto a una lista de bytes, se persistirá como Blob.

```
21@Entity(name = "bonos")
22@Table(name = "bonos")
23public class Bono {
24
25    @Id
26    @Column(name = "Id")
27    @GeneratedValue(strategy=GenerationType.IDENTITY)
28    private long id;
29
30    @ManyToOne(fetch = FetchType.LAZY)
31    @JoinColumn(nullable = false, name = "Id_Solicitante")
32    private Usuario solicitante;
33
34    @ManyToOne(fetch = FetchType.LAZY)
35    @JoinColumn(nullable = true, name = "Id_Operador")
36    private Usuario operador;
```

Figura 11. Ejemplo del mapeado ORM sobre la clase "Bono.java".

Una vez se haya especificado que estas clases del modelo del servicio sean entidades mediante el uso de JPA e Hibernate, se haya configurado previamente el archivo de configuración del servicio y se haya creado la base de datos, tras ejecutar el servicio se crearán todas las tablas, relaciones, etc. en base a las anotaciones asignadas.

5.2.2. Estructura servicio REST

Una vez se ha diseñado la tabla de operaciones y recursos REST del servicio con su respectivo diagrama de componentes, y se ha definido el mapeado ORM, ya se puede comenzar a implementar el servicio como tal. Este servicio contiene los siguientes tipos de archivos:

- Clases del modelo: estas clases alojan las entidades mencionadas en el subapartado anterior. Constan de una serie de atributos y sus respectivos métodos de obtención y modificación, es decir, sus getters y setters.
- DTOs: estos archivos representan los conjuntos de datos que se utilizarán en la capa de presentación relacionados con el modelo del servicio. Generalmente son clases similares y/o más simples que las propias entidades que representan. Constan, al igual que estas, de una serie de atributos con sus respectivos getters y setters.
- Controladores: estas clases de Java son cruciales en el desarrollo de la API. Constan de un conjunto de métodos que representan las operaciones definidas en la tabla de recursos REST. Implementadas bajo el framework de Spring, se sirven de varias anotaciones para definir con exactitud cada operación que el controlador gestione. Dentro de estas anotaciones, las usadas con más frecuencia son las siguientes:
    - @RestController: esta anotación se utiliza a nivel de clase y la identifica como un controlador REST, es decir, permite a ésta recibir peticiones de un cliente.
    - @CrossOrigin: esta anotación, utilizada junto a un argumento el cual corresponde a un URL o a un conjunto de URLs, sirve para controlar el acceso a los recursos del servicio, permitiendo solo las peticiones de las rutas que indique dicho argumento.
    - @GetMapping, @PostMapping, @PutMapping, @DeleteMapping: este tipo de anotaciones se utilizan a nivel de método para poder manejar las peticiones del cliente. La cabecera de la anotación especifica el tipo de operación que ofrece el método. Es decir, si se especifica un método con la anotación @GetMapping, se está indicando que la petición que se permite es de tipo GET. Además, en los argumentos de estas anotaciones, se debe especificar la URI que suministra e identifica el recurso al que corresponde dicho método.
    - @PathVariable: esta etiqueta acompaña a un parámetro del método de una operación REST y se utiliza para obtener información de la URI asociando el valor de la variable correspondiente a dicho parámetro del método. Para ello, como se puede observar en la figura 12, hay que utilizar el mismo nombre para la variable de la URI y el parámetro del método al que corresponde.
    - @RequestParam: esta anotación sirve, al igual que la anotación anterior, para asociar variables que llegan desde la petición del cliente a un parámetro del método correspondiente. Sin embargo, esta no va alojada en la URI, sino que se añade dentro de la cabecera de la petición HTTP. Por defecto, estas variables son obligatorias. Si se quisiese especificar que sean opcionales, bastaría con añadir en el argumento de la etiqueta "required=false".
    - @RequestBody: esta anotación, utilizada principalmente en peticiones de creación de recursos, sirve, al igual que las dos etiquetas anteriores, para asociar información de parte de la petición a un parámetro. En este caso, esta información es la correspondiente al cuerpo de la petición, que puede ser tanto un dato compuesto por datos primitivos como un objeto.

```
85    @GetMapping(value = "/bonos/{nombreMaq}/{numMaquina}")
86    public List<BonoDTO> getBonosMaquina(@PathVariable("nombreMaq") String nombreMaq,
87            @PathVariable("numMaquina") long numMaquina,
88            @RequestParam Boolean close) {
89
90        return service.findAllBonosByMaquina(nombreMaq, numMaquina, close);
91    }
```

Figura 12. Ejemplo de operación de un controlador REST.

- Servicios: estos archivos contienen la lógica de negocio de la API y sirven de enlace entre los controladores y los repositorios. Su función principal es la realizar peticiones a los repositorios cuando corresponda según su implementación, y en ese caso, devolver la información al controlador.
- Mapeadores: estos archivos auxiliares se encuentran dentro de los servicios de la API. Son los encargados de convertir DTOs en objetos del modelo y viceversa.
- Repositorios: estas interfaces, que extienden de CrudRepository [6], son las encargadas de realizar peticiones a la base de datos con el fin de obtener, añadir, modificar o eliminar información. Cada repositorio está asociado a una entidad, por lo que las peticiones que realice a la base de datos deben ser sobre la tabla referente a dicha entidad.
  CrudRepository es una especificación de Spring para definir repositorios CRUD, que por defecto aporta todo tipo de operaciones CRUD como findById(), deleteAll(), etc. Para definir un repositorio hay que indicar la entidad y el tipo de dato del id de dicha entidad como se puede ver en la definición de la interfaz de la figura 13.

```
9  public interface BonoRepository extends CrudRepository<Bono, Long> {
10
11     @Query("SELECT b FROM bonos b WHERE "
12             + "REPLACE(b.maquina.nombreMaq, ' ', '') = :nombreMaq "
13             + "AND b.maquina.numMaquina = :numMaquina "
14             + "AND b.instrumentista is not null "
15             + "AND (b.solicitanteFin is null OR b.operadorFin is null)"
16         )
17     List<Bono> findByMaquinaNotToClose(String nombreMaq, long numMaquina);
```

Figura 13. Ejemplo de definición de repositorio CRUD.

Además, como se ve en el ejemplo de la figura 13, para especificar queries o consultas más complejas no definidas por las operaciones básicas que aporta el repositorio CRUD, se puede utilizar la etiqueta @Query y definir en su argumento una consulta HQL. Estos tipos de consultas son traducidas a SQL mediante el framework de Hibernate y su uso es de especial interés cuando se requiere comprobar ciertas condiciones en éstas.

5.3. Estructura aplicación web

Como se mencionó en apartados anteriores, la capa de presentación ha sido desarrollada bajo el framework de Angular, mediante el cual se cargan todos los componentes sobre un componente principal denominado “app-component”.

Al igual que otro componente de Angular, éste consta también de su archivo dónde insertar cierta lógica en TypeScript, su propia hoja de estilos, su vista o template y, opcionalmente, un archivo de testing. En su template, como se puede ver en la figura 14 se cargan 3 vistas por defecto, las cuales van a ser visibles durante el ciclo de vida de la ejecución de la aplicación web.

Estos templates estáticos son siempre un header, una vista dinámica determinada por el archivo de enrutamiento del sistema (directiva router-outlet) y un footer.

```
<div class="principal">
    <div class="header"><app-header></app-header></div>
    <div class="content"><router-outlet></router-outlet></div>
    <div class="footer"><app-footer></app-footer></div>
</div>
```

Figura 14. Plantilla principal de la aplicación web.

5.3.1. Archivo de routing y enrutamiento de la aplicación web

El "app-routing.ts" [7] es el archivo auxiliar en el que se definen los paths o rutas de cada vista del sistema. Cada ruta asocia un componente a una URL, por lo que si en un navegador web se introduce una URL de esta lista con la aplicación en ejecución, debe cargar en la directiva router-outler el respectivo componente. Además, cómo se puede ver en la línea 12 de la figura 15, se pueden especificar redirecciones a otras rutas ya definidas. En este caso, se está especificando por defecto que cargue el componente asociado a la ruta "/login", ya que la aplicación se inicia sobre la dirección "/". La propiedad pathMatch con el valor "full" implica que debe coincidir la ruta exactamente igual para hacer el enrutado. Por defecto, su valor es el de "prefix", el cual en caso de haber introducido un fragmento de la URL de forma incorrecta, se queda con la parte correcta previa al error. Por ejemplo, si se tuviese una ruta "a/b/c" y se introdujese una URL "a/b/d", se accedería, en caso de estar definido, al componente asociado en la ruta "a/b".

```
11  const routes: Routes = [
12    { path: '', redirectTo: '/login', pathMatch: 'full' },
13    { path: 'home', component: HomeComponent },
14    { path: 'bonoDetail/:id', component: BonoDetailComponent },
```

Figura 15. Ejemplo de rutas del archivo de enrutamiento.

Para poder navegar entre vistas hay que realizar dos pasos principalmente:

1. Utilizar la directiva router-outlet sobre el template deseado para visualizar la vista deseada. En este caso particular se utiliza directamente sobre la vista del componente principal, como se explicó en la figura 14.
2. Inyectar el servicio Angular de routing "@angular/router" en cualquier componente que permita una navegación entre dicho componente y otros más. Una vez inyectado dicho servicio hay dos formas genéricas de especificar la transición:
    - Desde el archivo de TypeScript usar los métodos de navegación que ofrece el servicio, como "router.navigateByUrl(url), tal y como se puede ver en la figura 16, en la cual cuando se ejecuta el método "close()", se navega de nuevo al componente asociado a la path "/home".

```
close(): void {
  this.router.navigateByUrl('/home');
}
```

Figura 16. Ejemplo de navegación desde archivo TS.

    - Desde el propio template mediante la propiedad "router-link" a la cual se le asocia el path al que dirigirse como se puede ver en el ejemplo de la figura 17.

```
<tbody *ngFor="let bono of bonosPage">
    <tr routerLink="/bonoDetail/{{bono.bono.id}}"
```

Figura 17. Ejemplo de navegación desde template HTML.

5.3.2. Servicios

Este tipo de archivos, como bien se explicó previamente, sirven para poder acceder a recursos proporcionados por la API. Las operaciones que definen éstos consisten en peticiones HTTP que devuelven observables de cualquier tipo. Los observables en Angular son un tipo de objetos los cuales contienen información que está esperando "ser observada", es decir, son unos objetos que distribuyen información y permiten gestionar peticiones asíncronas, al igual que notificar los cambios que se produzcan en esta información.

Como se puede apreciar en la figura 18, esta petición de creación (post) tiene asociada una url, la cual es definible, un body o cuerpo, que en este caso es el bono, y unas opciones de petición http. En estas opciones, las cuales no son obligatorias, se pueden definir cabeceras con las que especificar propiedades de la petición como el tipo de contenido que se envía.

```
addBono(bono: BonoDetail): Observable<BonoDetail> {
  return this.http.post<BonoDetail>(this.bonosUrl, bono, this.getHttpOptions())
  .pipe(
    tap(_ => this.log('added bono')),
    catchError(this.handleError<BonoDetail>('addBono', ))
  );
}
```

Figura 18. Ejemplo de método de un servicio con una petición de tipo post a la API.

5.3.3. Componentes

Estos elementos, como ya se explicó en apartados previos, constan de varios tipos de archivos que componen el componente de Angular. Como se puede ver en la figura 19, al componente, definido en el archivo de código ts, se le asocia un selector con el que invocar al componente, la vista a la que está asociado y la hoja de estilos que modifica el formato de los elementos de su vista.

```
@Component({
  selector: 'app-bono-list',
  templateUrl: './bono-list.component.html',
  styleUrls: ['./bono-list.component.css']
})
```

Figura 19. Ejemplo definición de componente.

En cuanto a los templates de la aplicación, cabe destacar que son archivos escritos en HTML5, con lógica de Angular, para representar cualquier tipo de elemento sobre las ventanas de los navegadores web más utilizados hoy en día como Mozilla Firefox, Google Chrome, Opera, etc. mediante el uso de etiquetas. Angular ofrece nuevas directivas que aportan funcionalidades nuevas sobre estos templates. Durante el desarrollo de las vistas de la aplicación, las directivas más utilizadas han sido las siguientes:

- [(ngModel)] = "*attribute*": directiva utilizada para enlazar un atributo del componente a una variable definida dentro de un formulario en el template.
- *ngIf = "*condition*": directiva utilizada para imponer condiciones sobre elementos de la vista, las cuales pueden influir en el renderizado del elemento sobre el que esté asociado en función del resultado booleano de la condición expresada.

- *ngFor = “let *value* of *values*”: directiva utilizada principalmente para recorrer listas definidas en el componente.

En cuanto a las hojas de estilo, cabe destacar que son archivos CSS simples. Cada componente tiene su propia hoja de estilos, donde se definen propiedades de los elementos que aparecen en la vista las cuales afectan directamente a cómo se representan dichos elementos.

5.3.3.1. Dialogs

Los dialogs de Angular son un tipo de componente característico por mostrarse como ventanas emergentes y por la facilidad que tienen para compartir información con el componente que les invoca. Al igual que otro componente de Angular, éste tiene su propio archivo CSS, su template y su archivo TS con lógica.

Para crear un dialog, como se puede ver en la figura 20, es necesario crear una instancia de éste sobre el constructor del componente con “MatDialogRef”. De esta forma se puede referenciar dicho componente cuando se le invoque externamente.

```
constructor(public dialogRef: MatDialogRef<TextDialogComponent>,
  @Inject(MAT_DIALOG_DATA) public data: string[]) { }
```

Figura 20. Constructor del dialog “TextDialogComponent”.

Además, para inyectar información en la invocación del dialog se utiliza la directiva “MAT_DIALOG_DATA”. En caso de necesitar información cuando se cierra el diálogo, se puede redefinir el método de cierre del diálogo e introducir en su argumento cualquier dato a retornar.

Para invocar un dialog desde un componente es necesario importar el servicio “MatDialog” y crear una instancia de éste en el constructor del componente. Además, es necesario crear un método auxiliar para poder invocar el diálogo como se puede ver en la figura 21. En este método auxiliar se puede definir el formato del diálogo a invocar, los datos a compartirle, y cómo ha de comportarse cuando se cierre.

```
59    openDialog(validaciones: string[]): void {
60    const dialogRef = this.dialog.open(TextDialogComponent, {
61      panelClass: 'mat-dialog-container',
62      width:'500px',
63      height:'300px',
64      data: validaciones
65    });
66
67    dialogRef.afterClosed().subscribe(result => {
68      //En caso de que el diálogo tuviese información que devoler
69      //se podría tratar aquí en la variable result
70      return;
71    });
72  }
```

Figura 21. Invocación del diálogo “TextDialogComponent” desde el componente “Login”.

5.4. Autenticación

La API restringe el uso de sus recursos exclusivamente a usuarios que existan en el sistema mediante el uso de tokens temporales. Cualquier tipo de petición a uno de sus recursos que no sea de un usuario registrado devuelve un código de error 403, el cual indica que el recurso está

protegido, y que por lo tanto la petición no incluye en su cabecera la información de autenticación necesaria para poder acceder a dicho recurso.

El funcionamiento de este proceso de autenticación se puede resumir en el siguiente flujo:

1. El usuario, desde la pantalla de login, introduce sus credenciales (email y contraseña) e inicia sesión. Esto implica una llamada al método "login()" del servicio "UserService" de la aplicación web, con la que se hace una petición POST a la URL perteneciente al recurso "/users/*/token".
2. La API recibe esa petición y comprueba que exista un usuario en el sistema con esas credenciales. En caso afirmativo, obtiene dicho usuario, genera un token validable por el sistema de autenticación con un tiempo de expiración determinado, y lo devuelve. En caso negativo devuelve nulo.
3. La aplicación web recibe los datos del usuario y su token y los almacena temporalmente en el LocalStorage de Angular. LocalStorage es una forma de almacenamiento temporal de información en el lado del cliente propia de aplicaciones web de Angular. Es similar al uso de cookies. Cabe destacar que al cerrar sesión se limpia todo el contenido del almacenamiento local.
4. Los servicios de la aplicación web añaden el token a la cabecera Authorization de las peticiones HTTP, tal y como se muestra en la figura 22, con los que ya podría acceder a los recursos que ofrece la API. Es decir, a cada petición que haga le añade el token obtenido al iniciar sesión correctamente.

```
31    getHttpOptions(){
32      let httpOptions = {
33        headers: new HttpHeaders({
34          'Content-Type': 'application/json',
35          'Authorization': JSON.parse(this.userService.getCurrentUser()).token
36        })
37        };
38        return httpOptions;
39    }
```

Figura 22. Headers por defecto de las peticiones de los servicios de la aplicación web.

Para poder gestionar las peticiones del cliente en el back hay que tener en cuenta tres aspectos fundamentalmente: cómo se generan los tokens, cómo se validan las peticiones y cómo se habilitan y configuran los recursos de la API.

5.4.1. Configuración de Spring Security y restricción sobre los recursos de la API

Para configurar Spring Security es necesario modificar el archivo principal o main de la API. Mediante la etiqueta "@EnableWebSecurity" se habilita la seguridad de Spring definida a través del método "configure()". En este método, se deshabilita la protección por defecto de CSRF para poder implementar un sistema de autentificación personalizado sin que esta protección interfiera y produzca problemas. Se añade un filtro personalizado "JWTAuthorizationFilter" que gestionará la validación de las peticiones, entre otras cosas. Además, se restringe el acceso a todos los recursos de la API a excepción de los 2 tipos de peticiones de las líneas 61 y 62 de la figura 23. Es decir, cualquier usuario sin autentificar tiene acceso al login y al registro del sistema.

```
53  @EnableWebSecurity
54  @Configuration
55  class WebSecurityConfig extends WebSecurityConfigurerAdapter {
56
57      @Override
58      protected void configure(HttpSecurity http) throws Exception {
59          http.csrf().disable()
60              .addFilterAfter(new JWTAuthorizationFilter(), BasicAuthenticationFilter.class)
61              .authorizeRequests()
62              .antMatchers(HttpMethod.POST, "/users/*/token").permitAll()
63              .antMatchers(HttpMethod.POST, "/users").permitAll()
64              .anyRequest().authenticated()
65              ;
66          http.cors();
67      }
68  }
```

Figura 23. Segmento de código en el que se implementa la configuración de Spring Security.

5.4.2. Configuración del filtrado de peticiones

Para implementar el filtro que gestiona las validaciones de las peticiones, es necesario crear una clase auxiliar "JWTAuthorizationFilter" que extienda a la clase "OncePerRequestFilter" la cual mediante el método "doFilterInternal()" filtra cada petición HTTP que reciba la API. En esta clase se definen una cabecera, el prefijo esperado del token a recibir y una clave secreta con el que se realiza la firma del token.

Al sobrescribir el método "doFilterInternal()" se deben añadir condiciones para validar la petición recibida. La primera comprobación es que la petición tenga un header de autorización (Authorization). La segunda comprobación es que en caso de que se cumpla la condición anterior, el cuerpo de ese header comience con el prefijo definido en la clase. Si se cumple esto, quiere decir que, a priori, la petición lleva asociado un token generado por esta API, pero aún no se puede confirmar la validez de éste. La última condición para validar el token es convertir el token a "Claims". Este objeto "Claims" corresponde con las llaves del cuerpo del token, es decir, la información encriptada que aporta el token. Durante este parseo, se introduce como clave de firma el mismo secreto especificado al comienzo de la clase. Si esta validación del token se realiza correctamente, quiere decir que el token recibido se formó en esta API y que es un token que aún no ha expirado.

5.4.3. Generación de tokens JWT

Para generar el token hay que tener en cuenta cómo se han definido las condiciones de filtrado de peticiones mencionadas en el subapartado anterior para considerarles cómo válidos a la hora de pasar dicho filtrado. Teniendo esto en cuenta, el token se crea definiendo un prefijo, su fecha de creación, su fecha de expiración, su id, que en este caso es el email del usuario, y se firma usando un algoritmo predeterminado con un secreto específico. Cabe destacar que el prefijo y el secreto deben coincidir con los especificados en la clase "JWTAuthorizationFilter".

6. Pruebas

En este capítulo se describen las pruebas realizadas durante el desarrollo de la aplicación con las que se han ido identificando errores a lo largo del desarrollo. La finalidad de realizar estas pruebas es ofrecer un mínimo de calidad en el producto desarrollado y garantizar que cada módulo implementado funcione correctamente.

6.1. Pruebas unitarias

Las pruebas unitarias son aquellas que permiten comprobar el correcto funcionamiento de componentes o módulos de forma aislada, es decir, por sí solos. Para ello, en caso de tener alguna dependencia, se simulan sus interacciones con el componente a testear.

Se han implementado pruebas unitarias sobre la aplicación web mediante el uso del framework Jasmine. Al igual que otros frameworks con los que definir casos de prueba, mediante éste se hacen llamadas a funciones o bloques de códigos y se comprueba que el resultado es el esperado. En caso de tener dependencias con otros módulos, estos se pueden simular mediante el uso de spies [8] como se puede ver en la figura 24.

Los spies en Jasmine son unos elementos que tienen dos finalidades principalmente:

1. Alterar las respuestas de funciones y métodos de una clase o servicio.
2. Comprobar si se ha llamado a una función o método de una clase o servicio, y con qué parámetros, en caso de que tenga.

```
41    it('should call crearBonoVisible() - expected false', () => {
42      let userMockJSON = JSON.stringify({rol:'Operador'});
43      const spy = spyOn((component as any).userService, 'getCurrentUser')
44        .and.returnValue(userMockJSON);
45      expect(component.crearBonoVisible()).toEqual(false);
46      expect(spy).toHaveBeenCalled();
47    });
```

Figura 24. Ejemplo de un test implementado con Jasmine.

Para realizar este tipo de pruebas sobre la lógica de la capa de presentación se ha probado que cada componente de Angular creado se comporta como se espera individualmente.

6.2. Pruebas de integración

Este tipo de pruebas tienen la finalidad de comprobar que todos los módulos implementados funcionan correctamente como colectivo, es decir, que ningún módulo produce errores al interaccionar con otros módulos.

Para realizar estas pruebas sobre la aplicación web, se optó por usar la propia consola para comprobar que cada nuevo módulo implementado se integraba correctamente con el resto de módulos que eran parte de sus dependencias. Es decir, en caso de que por ejemplo un componente de Angular se comunicase con otro, como es el caso de los dialogs o las dependencias de un componente a un servicio, se comprobó manualmente que la información intercambiada entre éstos era la esperada.

Sobre la capa de backend, las pruebas de integración consistieron en realizar diversas peticiones a cada uno de los endpoints de dicha API, mediante el uso de la herramienta SoapUI, con las que

corroborar un correcto flujo por los diversos bloques del servicio REST. Para ello, se almacenaron un listado de peticiones que comunicaban con los endpoints a testear y al lanzarse, el desarrollador se sirvió en ciertos casos del depurador con el que se corroboró que la información intercambiada entre el cliente, es decir SoapUI en este caso, y el servicio era la esperada.

6.3. Pruebas de sistema

Estas pruebas son aquellas que se han seguido para probar los requisitos no funcionales definidos en la fase de requisitos.

Por ejemplo, para probar el requisito RNF5, por el cual se deben validar todos los inputs que introduce el usuario en caso de que sean erróneos en cuanto a formato. Para ello, se introducen cualquier tipo de dato no esperado esperando ver los mensajes de validación correspondientes.

6.4. Pruebas de aceptación

Estas pruebas, pese a que generalmente se han de realizar por un cliente final siguiendo un manual de usuario en caso de ser necesitado, han sido desarrolladas, por el momento, por el desarrollador. Cabe destacar, que dichas pruebas están pendientes de realizar por parte de usuarios externos con conocimiento de la lógica de negocio que define esta aplicación.

Para estas, el desarrollador ha probado cada posible escenario de uso. Primero, probando los resultados o caminos esperados de cada módulo; y segundo, probando los posibles flujos inesperados, intentando provocar comportamientos no deseados.

7. Conclusiones y trabajos futuros

Este apartado se divide en dos secciones. Una primera en la que se recogen las conclusiones obtenidas una vez finalizado el proyecto. Y una segunda sección, en la que se recogen un conjunto de posibles funcionalidades o nuevos módulos que se podrían desarrollar en un futuro.

7.1. Conclusiones

El objetivo principal de este proyecto desde un inicio ha sido adaptar una lógica de negocio utilizada en una empresa real, la cual se realiza manualmente, a un producto software, específicamente una aplicación web, la cual facilite la gestión de un trámite por parte de los empleados de dicha empresa.

Una vez finalizado el proyecto, se puede estar satisfecho con el producto final desarrollado. En un principio se han podido implementar todas las funcionalidades principales que afectan a la gestión de bonos y de los usuarios. Además, no ha habido muchos inconvenientes para integrar correctamente el servicio REST con la aplicación web, al igual que tampoco ha supuesto un gran inconveniente ofrecer una persistencia de las entidades definidas en el servicio. Cabe destacar que aún queda pendiente la fase de pruebas de aceptación y la aceptación por parte de la empresa para poder poner la aplicación en producción.

Uno de los aspectos más complejos a tener en cuenta durante la realización de este proyecto, ha sido la integración de Spring Security para ofrecer control y gestión de recursos del servicio mediante tokens JWT. Esto se debe a la pobre documentación que se aporta en los sitios oficiales referidos a estos elementos, lo cual ha implicado un mayor trabajo de investigación por parte del desarrollador.

Desde un punto de vista personal, he de decir que estoy muy satisfecho con lo que ha supuesto este reto para mí, ya que no solo he puesto en práctica los conocimientos adquiridos durante la carrera, sino que también he adquirido un gran conocimiento sobre varios frameworks y lenguajes de programación. Siempre me he considerado una persona que tiene más conocimiento de backend que de frontend, por lo que el desarrollo de una aplicación web, con un framework tan utilizado hoy en día como es Angular, ha supuesto un gran desarrollo en mi persona a nivel profesional. Por último, me gustaría recalcar que el aprendizaje de estos frameworks nuevos, al igual que el de alguna otra librería, me han servido de motivación para continuar adquiriendo más conocimientos referidos al mundo del software.

7.2. Trabajos futuros

Cabe destacar que esta aplicación web es una primera versión inicial limitada a un alcance específico. Me hubiese gustado tener más tiempo para integrar nuevas funcionalidades como las que cito a continuación:

- Realizar el registro de usuarios exclusivamente bajo la VPN de la empresa específica.
- Integrar Autofirma para el firmado de documentos.
- Adaptar la aplicación web a aplicación móvil (para Android).
- Facilitar una impresión de bonos dentro del detalle de éstos.

Referencias

[1] VMware Inc. *Documentación del framework de Spring*. Último acceso realizado el día 17-06-2022 en https://docs.spring.io/spring-framework/docs/current/reference/html/overview.html#overview

[2] Baeldung. *Documentación sobre el patrón de Inyección de dependencias*. Último acceso realizado el día 17-06-2022 en https://www.baeldung.com/spring-dependency-injection#:~:text=Dependency%20Injection%20is%20a%20fundamental,managing%20components%20onto%20the%20container.

[3] Auth0 Inc. *Documentación sobre los tokens JWT.* Último acceso realizado el día 17-06-2022 en https://jwt.io/introduction

[4] Google LLC. *Documentación del framework de Angular*. Último acceso realizado el día 17-06-2022 en https://angular.io/guide/developer-guide-overview

[5] Baeldung. *Documentación sobre el patrón DTO (Data Transfer Object).* Último acceso realizado el día 17-06-2022 en https://www.baeldung.com/java-dto-pattern

[6] VMware Inc. *Documentación sobre los CrudRepository de Spring*. Último acceso realizado el día 17-06-2022 en https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html

[7] Google LLC. *Documentación sobre el redireccionamiento o routing de Angular.* Último acceso realizado el día 17-06-2022 en https://angular.io/guide/router

[8] Pivotal Labs. *Documentación sobre los Spies de Jasmine*. Último acceso realizado el día 17-06-2022 en https://jasmine.github.io/tutorials/your_first_suite#section-Spies